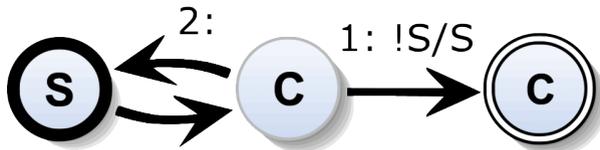


SCCharts

Language and Interactive Incremental Compilation

Dipl.-Inf. Christian Motika



Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2016

Kiel Computer Science Series (KCSS) 2017/2 v1.0 dated 2017-10-11

URN:NBN urn:nbn:de:gbv:8:1-zs-00000334-a7

ISSN 2193-6781 (print version)

ISSN 2194-6639 (electronic version)

Electronic version, updates, errata available via

<https://www.informatik.uni-kiel.de/kcss>

The author can be contacted via <http://www.motika.de>

Published by the Department of Computer Science at Kiel University

Real-Time and Embedded Systems Group

Please cite as:

- ▶ Christian Motika. *SCCharts — Language and Interactive Incremental Compilation*. Number 2017/2 in Kiel Computer Science Series. Dissertation, Faculty of Engineering, Kiel University, Germany, 2017.

```
1 @book{Motika17,  
2   author   = {Christian Motika},  
3   title    = {SCCharts — Language and Interactive Incremental Compilation},  
4   publisher = {Department of Computer Science},  
5   year     = {2017},  
6   isbn     = {9783746009391},  
7   series   = {Kiel Computer Science Series},  
8   number  = {2017/2},  
9   doi      = {10.21941/kcss/2017/02},  
10  note     = {Dissertation, Faculty of Engineering,  
11             Kiel University, Germany}  
12 }
```

© 2017 by Christian Motika

Herstellung und Verlag:

BoD – Books on Demand, Norderstedt, Germany

ISBN 978-3-7460-0939-1

About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at the Christian-Albrechts-Universität zu Kiel. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit <http://www.informatik.uni-kiel.de/kcss> for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

1. Gutachter: Prof. Dr. Reinhard von Hanxleden
Kiel University, Kiel, Germany
2. Gutachter: Prof. Florence Maraninchi
VERIMAG Institute, Grenoble, France

Datum der mündlichen Prüfung: 21. April 2017

Zusammenfassung

Sicherheitskritische Systeme sind eine Unterklasse von reaktiven Systemen, welche heutzutage eine der wichtigsten und größten Klasse von Computersystemen darstellt. Solche Systeme kontrollieren die Airbags unserer Autos, die Landeklappen eines Passagierflugzeugs, Kernkraftwerke oder Herzschrittmacher. Software für solche Systeme muß absolut zuverlässig sein. Daher werden Computersprachen und Werkzeuge benötigt, die es erlauben, zuverlässige Softwaremodelle zu erstellen und zu warten. Weiterhin braucht es zuverlässige Compiler, die aus solchen abstrakten Modellen korrekten maschinenlesbaren und ausführbaren Code erzeugen.

Mit SCCharts präsentiert diese Arbeit eine zustandsmaschinenbasierte und synchrone Modellierungssprache für den Entwurf und zur Implementierung sicherheitskritischer Systeme. Es wird betrachtet, warum sich dafür eine kontrollflußorientierte und synchrone Sprache besonders gut eignet und welche Wahl inkrementeller Sprachbestandteile die Lernkurve senken können. Die Arbeit zeigt, wie ein als SLIC bezeichneter, interaktiver, inkrementeller und auf Modelltransformationen basierender Kompilierungsansatz sowohl dem Modellierer dabei helfen kann, zuverlässige Modelle zu erstellen, als auch den Werkzeugentwickler darin unterstützt, einen zuverlässigen Compiler bereit zu stellen. Es wird ein auf SLIC basierender SCCharts Compiler inklusive seiner high-level Modelltransformationen vorgestellt. Weiterhin wird der vorgestellte Ansatz mit Hilfe der beispielhaft umgesetzten KIELER SCCharts Sprach- und Werkzeugimplementierung auf seine Praktikabilität hin überprüft.

Abstract

Safety-critical systems are a subclass of reactive systems, a dominating class of computer systems these days. Such systems control the airbags in our cars, the flaps of an aircraft, nuclear power plants or pace makers. Software for these systems must be reliable. Hence, a language and tooling is needed that allows to build and maintain reliable software models. Furthermore, a reliable compiler is required to obtain decent machine-understandable and executable code from highly abstract models.

This thesis presents SCCharts, a Statecharts-based synchronous and visual modeling language for specifying and designing safety-critical systems and for deriving their implementations. It elaborates on why a control-flow oriented and synchronous language is desirable and how incremental language features are chosen to flatten learning curve. It presents an interactive incremental model transformation based compilation approach termed SLIC. It shows how SLIC helps in supporting both, the modeler and the tool smith for building reliable models and maintaining a reliable compiler, respectively. A SLIC-based compiler for SCCharts including its high-level model transformations is presented. Furthermore, practicality aspects of the KIELER SCCharts language and tooling implementation complete the considerations to validate the proposed approach.

Contents

1	Introduction	1
1.1	Contributions of this Thesis	9
1.2	Related Publications	10
1.2.1	Major Publications	10
1.2.2	Other Publications	13
1.2.3	Supervised Theses	15
1.3	Outline	18
2	Background and Related Work	19
2.1	Synchronous Languages	19
2.2	Statecharts	22
2.2.1	Argos	23
2.3	SyncCharts	24
2.3.1	The <i>hello world</i> of synchronous programming (ABRO)	26
2.3.2	Advanced SyncCharts Features	26
2.4	Other Statechart Dialects	28
2.4.1	SCADE State Machines	30
2.4.2	UML Statecharts	32
2.4.3	Stateflow	34
2.4.4	Modechart	35
2.5	Code Generation from Statecharts	35
2.6	Sequential Constructiveness	36
2.6.1	Synchronous Programming Classes	41
2.7	The Sequentially Constructive Graph (SCG)	41
2.7.1	Explicit Data Dependencies	42
2.7.2	Abstract Syntax	43
2.7.3	SCG Example	44
2.8	Model-Based Compilation and Compiler Infrastructure	45

Contents

3	The SCCharts Language	47
3.1	Basic Concepts	48
3.1.1	General Structure	48
3.1.2	Termination	49
3.1.3	Synchronous Signals	50
3.1.4	Declarations	51
3.1.5	Instantaneous Communication and Preemption	51
3.1.6	Transitions	51
3.1.7	Actions	52
3.2	Visual Syntax and Semantics	52
3.2.1	Termination Transitions	54
3.2.2	Transition Priorities	54
3.2.3	Weak Abort Transitions	58
3.2.4	Entry Actions	62
3.2.5	Exit Actions	63
3.2.6	During Actions	63
3.2.7	Initializations	66
3.2.8	Connectors	67
3.2.9	Suspensions	68
3.2.10	Count Delays	68
3.2.11	The Pre Operator	69
3.2.12	History Transitions	70
3.2.13	Complex Final States	72
3.2.14	Static Variables	72
3.2.15	Deferred Transitions	73
3.3	The Textual Syntax: SCT	75
3.3.1	Grammar	76
3.3.2	Identifier Name vs. Label	76
3.3.3	Priorities	78
3.3.4	Host Code	79
3.3.5	Other Elements	80
3.4	Abstract Syntax	81
3.4.1	States and Regions	83
3.4.2	Transitions	83
3.4.3	Actions	84

3.4.4	Expressions	84
3.4.5	Syntax Validation	84
4	Interactive Incremental Compilation	87
4.1	Single-Pass Language-Driven Incremental Compilation (SLIC)	89
4.1.1	SLIC Foundations	91
4.1.2	Deriving a SLIC Schedule	95
4.1.3	SLIC Key Questions	98
4.1.4	SLIC Order for SCCharts Compilation	99
4.1.5	SLIC Language and Sublanguage	101
4.1.6	SLIC Design Challenges	103
4.2	Interactive Compilation	104
4.2.1	Traditional User Story	104
4.2.2	Interactive User Story	106
4.2.3	Element Tracing	109
4.3	Interactive Incremental SCCharts Compilation	111
5	Compiling SCCharts	113
5.0.1	High-Level Compilation Overview	113
5.0.2	Low-Level Compilation Overview	115
5.1	SyncCharts Compilation	116
5.2	High-Level Compilation	123
5.2.1	Compiling ALDO to Core SCCharts	124
5.2.2	Compiling ALDO to SCG	131
5.2.3	Transformation Dependencies	134
5.2.4	Connector	135
5.2.5	Entry Action	137
5.2.6	Exit Action	140
5.2.7	Initialization	144
5.2.8	Abort	146
5.2.9	Complex Final State	176
5.2.10	During Action	181
5.2.11	Signal	190
5.2.12	Pre	193

Contents

5.2.13	Suspend	199
5.2.14	Count Delay	201
5.2.15	History	206
5.2.16	Static Variables	210
5.2.17	Deferred	211
5.2.18	Weak Suspend	215
5.2.19	Normalization	221
5.2.20	Constructing the SCG	231
5.3	Pseudocode for High-Level Transformations	232
5.3.1	Connector	233
5.3.2	Entry Action	235
5.3.3	Exit Action	236
5.3.4	Initialization	236
5.3.5	Abort	239
5.3.6	Complex Final State	244
5.3.7	During Action	246
5.3.8	Signal	247
5.3.9	Pre	249
5.3.10	Suspend	251
5.3.11	Count Delay	252
5.3.12	History	253
5.3.13	Static Variables	255
5.3.14	Deferred	256
5.3.15	Weak Suspend	257
5.3.16	Normalization	260
5.4	Priority-Based Low-Level Compilation	263
5.4.1	S	265
5.4.2	S Code Generation	269
5.4.3	SJL/SCL Code Generation	271
5.5	Circuit-Based Low-Level Compilation	273
5.5.1	S Code Generation	273
5.5.2	Java/C Code Generation	275
5.6	Compilation Design Choices	276
5.6.1	Circuit vs. Priority	278
5.6.2	RISC vs. CISC	279

5.6.3	Defaults for KIELER SCCharts Compiler Implementation	283
5.7	SCCharts Targets	283
5.7.1	Software	285
5.7.2	Hardware	286
6	SCCharts Tooling	303
6.1	Related Tools	303
6.1.1	Ptolemy	303
6.1.2	SCADE	307
6.1.3	Esterel Studio	308
6.1.4	Berry Esterel v5.92 Compiler and Simulator	311
6.1.5	The Columbia Esterel Compiler (CEC)	312
6.2	Eclipse	312
6.2.1	Eclipse Plugins	313
6.3	The KIELER Simulation Infrastructure	316
6.3.1	KIELER Execution Manager (KIEM)	316
6.3.2	Data Table	318
6.3.3	Synchronous Signals View	320
6.3.4	Benchmark Component	320
6.4	SCCharts Editor Implementation	322
6.4.1	KIELER ThinKCharts Editor	324
6.4.2	The Yakindu SCCharts Editor	324
6.4.3	The SCCharts Textual Editor	326
6.4.4	Graphical vs. Textual vs. <i>Textical</i> Modeling	327
6.5	Interactive SLIC Compiler Implementation	332
6.5.1	KIELER Compiler (KiCo) Implementation	332
6.5.2	KiCo 1.0 Prototype	333
6.5.3	Disadvantage of KiCo 1.0	335
6.5.4	The KiCo 2.0 Advanced Prototype	337
6.5.5	SLIC Implementation Notes	343
6.5.6	Interactive Compilation GUI: KiCo.UI	344
6.5.7	Command Line and Online Compilation	347
6.6	Automatic Validation	353
6.6.1	Regression Tests	353

Contents

6.6.2	KIEM JUnit Test Runner	356
6.6.3	Esterel Input Output Trace (ESO) File Creation	358
6.6.4	Benchmark Regression Tests	360
7	Practicality & Validation: The Model Railway Demonstrator	365
7.1	Introduction	366
7.2	SCCharts Model Railway Project	367
7.2.1	SCCharts Topology Scenario	367
7.2.2	Complex System Modeling with SCCharts	368
7.2.3	SCCharts Compilation Performance	372
7.3	SCCharts Survey	375
7.3.1	SCCharts vs. Other Languages	376
8	Related Projects	381
8.1	Ptolemy and KIELER	381
8.1.1	Ptolemy	381
8.1.2	KIELER leveraging Ptolemy (KlePto)	386
8.2	KIELER Esterel Integration	393
8.2.1	Esterel	394
8.2.2	Meta Model	396
8.2.3	Simulation and Debugging	397
8.2.4	Sequentially Constructive Esterel (SCEst)	403
8.3	Esterel to SyncCharts Transformation	405
8.4	Synchronous Java (Light)	407
8.4.1	Coroutines and Deterministic Concurrency	408
8.4.2	Synchronous C	409
8.4.3	Synchronous Java — Concept and Realization	409
9	Conclusion and Future Work	417
9.1	Conclusion	417
9.1.1	Reliable Modeling	418
9.1.2	Reliable Compilation	418
9.1.3	Practicality	419
9.2	Future Work	419
9.2.1	SCCharts	420

Contents

9.2.2	SCEst and Esterel	425
9.2.3	Related Projects	426

Abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CEC	Columbia Esterel Compiler
CFG	Control-Flow Graph
CISC	Complex Instruction Set Computer
COTS	Commercial Off-The-Shelf
CPU	Central Processor Unit
CT	Continuous Time
DE	Discrete Events
DSL	Domain-Specific Language
ELK	Eclipse Layout Kernel
EMF	Eclipse Modeling Framework
ESO	Esterel Input Output Trace
FPGA	Field Programmable Gate Array
FSM	Finite-State-Machine
GMF	Graphical Modeling Framework
GUI	Graphical User Interface
HDL	Hardware Description Language
HTTP	Hypertext Transfer Protocol

Contents

HW	Hardware
ID	Identifier
IDE	Integrated Development Environment
I/O	Input/Output
ISA	Instruction Service Architecture
ISE	Integrated Synthesis Environment
IUR	Initialize-Update-Read
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KCSS	Kiel Computer Science Series
KEP	Kiel Esterel Processor
KiCo	KIELER Compiler
KIEL	Kiel Integrated Environment for Layout
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KIEM	KIELER Execution Manager
KITS	KIELER Textual SyncCharts
KlePto	KIELER leveraging Ptolemy
KlePto	KIELER leveraging Ptolemy
KLighD	KIELER Lightweight Diagrams
LED	Light-Emitting Diode
M2M	Model-to-Model
MDE	Model Driven Engineering

MDSD	Model Driven Software Development
MoC	Model of Computation
MOML	Modeling Markup Language
NXT	NeXT Generation
OS	Operating System
PC	Personal Computer
PHP	PHP Hypertext Preprocessor
PN	Process Networks
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RTL	Real-Time Logic
SC	Synchronous C
SCADE	Safety Critical Application Development Environment
SCCharts	Sequentially Constructive Statecharts
SCEst	Sequentially Constructive Esterel
SCG	Sequentially Constructive Graph
SCL	Lightweight Synchronous C
SCT	SCCharts Textual Language
SDF	Synchronous Data-Flow
SJ	Synchronous Java
SJL	Lightweight Synchronous Java
SLIC	Single-Pass Language-Driven Incremental Compilation

Contents

SR	Synchronous Reactive
SSA	Static Single Assignment
SSM	Safe State Machines
STG	State Transition Graph
SW	Software
TCP	Transmission Control Protocol
UI	User Interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier
USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit Hardware Description Language
WCRT	Worst Case Reaction Time
WTO	Write Things Once
WWW	World Wide Web
WYSIWYG	What You See Is What You Get
XML	Extensible Markup Language

List of Figures

1.0.1	Computer system classes	1
1.0.2	Reactive system cyclic control loop	3
2.1.1	Synchrony Hypothesis	20
2.1.2	A Statecharts example	22
2.2.1	Argos example	23
2.3.1	SyncCharts notation and example	25
2.3.2	A <i>Reactive Cell</i> of SyncCharts	27
2.3.3	Statechart semantic problem example	28
2.4.1	A simple SCADE automaton	29
2.4.2	UML Statecharts	33
2.4.3	Stateflow example	34
2.6.1	Advantages and limits of SCChart’s sequentially construc- tive semantics	37
a	Concurrent causality	37
b	Sequential causality	37
2.6.2	Synchronous program classes	40
2.7.1	Elements of an SCG	42
2.7.2	Different types of SCG dependencies	42
2.7.3	SCG meta model	44
2.7.4	SCG for the AO example	45
3.0.1	KIELER SCCharts <i>textical</i> modeling tool	47
3.1.1	ALDO SCCharts example	49
3.2.1	Overview of the SCCharts visual syntax	53
3.2.2	Hiding complexity using extended SCCharts features	55
a	ALDO Extended SCChart	55
b	ALDO Core SCChart	55
3.2.3	Transition priorities as a tie breaker	56

List of Figures

3.2.4	Weak abort transitions triggered from the inside	58
3.2.5	Causally wrong use of a strong aborts	60
3.2.6	Weak abort or termination do not induce any further constraints	61
3.2.7	Visibility pitfall when replacing during actions	64
3.2.8	Example use-case of count delay	69
3.2.9	Example use-case of history transition	70
3.2.10	Example use-case of complex final state	71
3.2.11	Example use-case for static keyword	73
3.2.12	Example use-case of deferred transition	74
3.3.1	Host code usage suggestion example	80
3.4.1	SCCharts meta model (simplified)	82
4.0.1	High-level interactive model-based compilation user story in KIELER tool	88
4.1.1	Single-Pass Language-Driven Incremental Compilation as an incremental model-based compilation strategy	91
4.1.2	Example SLIC features	92
4.1.3	Example SLIC transformations	97
4.1.4	Example SLIC schedule	97
4.1.5	SCCharts transformation dependency details	99
4.1.6	SCCharts transformation dependencies	101
4.1.7	SCCharts transformation dependencies without groups	102
4.2.1	Traditional vs. interactive model-based compilation user story	105
a	Traditional compilation	105
b	Interactive compilation	105
4.2.2	Element tracing with Single-Pass Language-Driven Incremental Compilation (SLIC) and usage	110
a	Building tracing tree for model elements	110
b	Utilizing tracing tree by convergecasting information	110
4.2.3	SLIC concept mapped onto KIELER SCCharts tool	112
5.0.1	SCCharts compilation tree (from [MSvH14])	114
5.0.2	Five patterns for Normalized (Core) SCCharts and their direct mapping to SCG elements	115

List of Figures

5.1.1	The SyncCharts-S compiler of KIELER 0.8.0 release	117
a	SyncCharts-S compiler	117
b	SyncCharts compiler transformations	117
5.1.2	SCCharts compilation tree	123
5.2.1	KiCo compiler selection for ALDO	124
5.2.2	High-level compilation from Extended to Core ALDO I/II . .	126
5.2.3	High-level compilation from Extended to Core ALDO II/II .	127
5.2.4	High-level compilation from Normalized to SCG ALDO . . .	132
a	Normalized SCChart	132
b	Sequentially Constructive Graph (SCG)	132
5.2.5	KiCo compiler selection showing all high-level transforma- tions	134
5.2.6	Statecharts feature transformations	135
5.2.7	Connector feature expansion transformation	136
5.2.8	Entry feature expansion transformation	137
5.2.9	Entry action combined with a strong abort	139
5.2.10	Exit feature expansion transformation	141
5.2.11	Exit action combined with a strong abort	143
5.2.12	Initialization feature expansion transformation	144
5.2.13	Initialization combined with existing entry action	145
5.2.14	Simple abort example	147
5.2.15	Weak abort of cyclic action — before transformation	148
5.2.16	Weak abort of cyclic action — after transformation (wrong)	148
5.2.17	Weak abort of cyclic action — after correct transformation .	149
5.2.18	Abort: Ease down-stream compilation	151
5.2.19	Abort transformation for delayed weak aborts	153
5.2.20	Abort transformation for delayed weak aborts (similar ex- ample)	154
5.2.21	Abort transformation termination elimination	156
5.2.22	Abort transformation — further optimizations (I/II)	157
5.2.23	Abort transformation — further optimizations (II/II)	158
5.2.24	General case: Abort transformation (WTO)	159
5.2.26	Abort transformation: Dealing with the (dynamic) transient forking state	161

List of Figures

5.2.27	Abort: Further ease down-stream compilation and reduce complexity	162
5.2.28	Abort (non-WTO): Mixing immediate and delayed aborts shall not lead to an earlier termination.	165
5.2.29	Abort (non-WTO): Mixing delayed strong aborts with other transformation types shall not lead to priority inversion.	167
5.2.30	Abort (non-WTO): A final state but no termination shall not lead to an undesired leaving of a state.	168
5.2.31	General case: Abort transformation (non-WTO)	170
5.2.32	Enhanced abort transformation with correctly omitted control region (non-WTO)	171
5.2.33	Nested abort example (original and non-WTO)	172
5.2.34	Abort feature expansion transformation with connectors	173
5.2.35	Scheduling problem for self-loops	174
5.2.36	Complex final state feature expansion transformation	176
5.2.37	Complex final state feature expansion transformation for two complex final states	177
5.2.38	Complex final state feature expansion transformation in the hierarchical case	178
5.2.39	Complex final state feature expansion transformation optimizing regions with final states only	179
5.2.40	Complex final states in the root state	180
5.2.41	During action feature expansion transformation	182
5.2.42	During action examples with termination	183
5.2.43	During action examples with termination	184
5.2.44	Simple during transformation and advanced during transformation	186
5.2.45	During action should continue as long as state S1 is active and not aborted	187
5.2.46	Simple during action original vs. new transformation	189
5.2.47	SyncCharts feature transformations	190
5.2.48	Signal feature expansion transformation	190
5.2.49	Valued signal feature expansion transformation	192
5.2.50	Pre operator feature expansion transformation, wrong (b), correct (c), and alternative (d)	194

5.2.51	Pre operator for states with a termination transition	196
5.2.52	Nested pre operator feature expansion transformation	197
5.2.53	Pre operator feature expansion with signals	197
5.2.54	Pre operator feature expansion with valued signals	198
5.2.55	Suspend feature expansion transformation	200
5.2.56	First proposed count delay transformation	202
5.2.57	Current count delay feature expansion transformation	202
5.2.58	Count delay combined with suspend feature expansion	204
5.2.59	Count delay feature with during action	205
5.2.60	SCADE / QUARTZ / Esterel v7 feature transformations	206
5.2.61	History feature expansion transformation	206
5.2.62	Deep history feature expansion transformation	207
5.2.63	Mixed deep history and shallow history features	208
5.2.64	Static feature expansion transformation	210
5.2.65	Deferred feature expansion transformation	212
5.2.66	Shallow deferred feature expansion transformation	213
5.2.67	Deep deferred feature expansion transformation	214
5.2.68	Weak suspend expansion transformation	216
5.2.70	Weak suspend expansion and scoping	219
5.2.71	Weak suspend and hierarchy	221
5.2.72	Normalization transformations	221
5.2.73	Five allowed patterns for Normalized (Core) SCCharts	222
5.2.74	Sandwich SCCharts example and S code generation	222
5.2.75	Trigger/effect normalization transformation examples	225
5.2.76	Surface/depth normalization transformation and optimization	227
5.2.77	<i>Apparent priority inversion</i> when normalizing SCCharts	230
5.2.78	AO example as Extended SCChart, Normalized SCChart and SCG	231
a	AO example as Extended SCChart	231
b	Mapping between Normalized SCCharts pattern constructs and SCG elements	231
5.4.1	S intermediate language in the context of SCCharts and SyncCharts SW compilation	264

List of Figures

5.4.2	SCCharts compilation tree (cf. Figure 5.0.1 on page 114): Priority low-level synthesis part	265
5.4.3	S intermediate language example	265
5.4.4	S intermediate language meta model	268
5.4.5	Simulation with the SCCharts compiler predecessor	270
5.4.6	Priority-based code generation from S	272
5.5.1	SCCharts compilation tree: Circuit-based low-level synthe- sis part	273
5.5.2	Circuit-based code generation from S	274
5.5.3	Sequentialized SCG examples	276
a	Sequentialized SCG for AO	276
b	Sequentialized SCG for ALDO	276
5.6.2	Circuit vs. priority SCCharts compilation: Code size	278
5.6.3	Comparison of Extended SCCharts (CISC) with equivalent Core SCCharts (RISC)	280
5.6.4	Comparison of C code synthesis paths for SCCharts	281
5.7.1	SCCharts SW and HW targets	284
5.7.2	Arduino microcontroller platform	287
5.7.4	Targeting the Arduino platform from KIELER	293
5.7.5	The SCCharts-Arduino demonstrator: ABROINO	294
5.7.6	SCCharts to Esterel during CEC simulation	295
5.7.7	Running SCCharts on FPGAs: The ISE tool	297
5.7.8	AO SCChart visualized as HW circuit inside the KIELER SCCharts tool during simulation	300
5.7.9	Simulation and visualization of SCCharts compiled to elec- trical HW circuits	301
6.1.1	Ptolemy graphical model editor “Vergil”	304
6.1.2	Simulating a Ptolemy model	305
6.1.3	ABRO synchronous state machine within the SCADE suite and its model simulation GUI	308
6.1.4	Esterel Studio GUI	309
6.1.5	Esterel v5.92 simulator running ALDO	310
a	Tick 0	310
b	Tick 1	310

c	Tick 2	310
d	Tick 3	310
6.2.1	Eclipse IDE with editors and views	314
6.2.2	Eclipse extension point mechanism	315
6.3.1	GUI of the KIELER simulation infrastructure while simulating the ALDO SCCharts model	316
6.3.2	Schematic overview of KIEM	317
6.3.3	GUI for input/output and debugging: KIELER Data Table and Synchronous Signals view	319
6.3.4	Benchmarking user story for SCCharts	321
6.4.1	GMF dashboard for graphical editor generation	322
6.4.2	Evolution of KIELER SCCharts editor	323
a	KIEL tool (SyncCharts)	323
b	ThinKCharts editor (SyncCharts)	323
c	Yakindu SCCharts editor	323
d	Xtext and KLighD-based SCCharts editor (current)	323
6.4.3	KIELER ThinKCharts editor	325
6.4.4	Yakindu statecharts tools overview	326
6.4.5	KIELER SCCharts modeling tool implementation	329
6.5.1	KiCo 1.0 prototype implementation overview	333
6.5.2	KiCo 1.0 advanced selection algorithm example	335
6.5.3	KiCo 2.0 advanced prototype implementation overview	338
6.5.4	KiCo 2.0 advanced selection algorithm example	341
6.5.5	Example of final SLIC order for processing transformations in a compilation run	343
6.5.6	The graphical user interface plugin infrastructure of KiCo	344
6.5.7	Different compiler selection views for the user and the developer of KiCo	346
6.5.8	KiCo server infrastructure	348
6.5.9	KiCo server GUI	350
6.5.10	KiCo command line usage	351
6.5.11	KiCo online compiler: http://www.sccharts.com	353
6.6.1	KIEM JUnit integration running regression tests	354
6.6.2	KIEM JUnit integration schematics	357
6.6.3	Creating ESO files manually (left) or automatically (right)	359

List of Figures

6.6.4	KIEM JUnit integration schematics enhanced with benchmarking	361
6.6.5	Benchmark output during SCCharts regression tests	362
6.6.6	KIEM schedule for SCCharts regression tests	363
7.0.1	Model railway demonstrator	365
7.1.1	Conceptional view to the communication of the 4th generation compared to the 3th generation	367
7.1.2	SCCharts controller topology scenario	368
7.2.1	SCCharts train controller vs. Harel Wristwatch	369
a	Harel's Wristwatch example	369
b	Top layer of SCCharts train controller	369
7.2.2	SCCharts SLIC-based compilation approach turns out to be practically usable	371
a	Hiding complexity by using Extended SCCharts features	371
b	Tickets as opened and closed in the bug tracker	371
7.2.3	Performance of SCCharts SLIC-based compiler	373
7.2.4	Performance of SCCharts SLIC-based compiler	374
7.3.1	The SCCharts language compared to other languages	376
8.1.1	Ptolemy actor model example	382
8.1.2	A hierarchical and heterogeneous Ptolemy model	386
8.1.3	SCChart (left) and an automatically generated Ptolemy model (right)	388
8.1.4	Generated Ptolemy model with optimized inputs and outputs	389
8.1.5	Traffic Light model automatically generated by KIELER leveraging Ptolemy (KlePto)	390
8.1.6	Transformation and execution scheme of the Ptolemy-based SyncCharts/SCCharts KlePto simulation	391
8.1.7	New immediate transitions in Ptolemy	393
8.2.1	Eclipse EMF Esterel meta model (simplified)	395
8.2.2	External compiler integration and simulation visualization concept	396
8.2.4	External (blackbox) compiler integration and simulation visualization concept	398

List of Figures

8.2.5	The KIELER Esterel simulator with visualization is running ALDO by leveraging the CEC.	399
8.2.6	ALDO simulation with KIELER SCEst compiler integration as a CEC alternative	403
8.2.7	Size and speed comparison of compilation with SCEst and the CEC	404
8.3.1	Interactive transformations for visual models	406
8.4.1	Comparison of Lightweight Synchronous Java (SJL) and coroutine thread concepts	408
8.4.2	SCChart of SJL example	412
8.4.3	State diagram of an SJL program's life cycle	414
8.4.4	Worst-case runtimes, SJL vs. Synchronous Java (SJ) vs. standard Java threads	415
9.1.1	SCCharts reliable modeling and compilation	417

Listings

3.3.1 ALDO example as SCCharts Textual Language (SCT)	76
3.3.2 SCCharts feature overview visualized in Figure 3.2.1 as its textual SCT equivalent	77
3.3.3 SCT Xtext based grammar excerpt	78
5.1.1 SyncChartsSSimulationDataComponent.java: Snippet of simulation component applying “SyncCharts extended” feature transformations before calling the SyncCharts to S core compiler Synccharts2S.	119
5.2.1 Snippet from SyncCharts2S compiler, handling transformation for the depth of a state	224
5.4.1 S intermediate language Xtext grammar	267
5.7.1 Arduino code generated from AO SCCharts example	290
8.2.1 ALDO example as an Esterel program	394
8.2.2 Esterel source-to-source visualization transformation for a statement P	397
8.2.3 Simple Esterel IO.strl example before simulation visualization transformation	397
8.2.4 Transformed version IO.simviz.strl after applying the visualization transformation to IO.strl	397
8.4.1 tick() method of SJL example from Figure 8.4.2	412

List of Tables

2.6.1	SCCharts transition trigger and effect examples	39
4.2.1	Comparing transitional and interactive model-based compilation user stories	108
5.1.1	Advantages and drawbacks of SyncCharts/SCCharts compilers	120
5.2.1	During actions in root state	188
5.6.1	Design choices for SCCharts compilation	277
5.7.1	Advantages of using SCCharts for modeling Arduino software vs. programming the C-like code directly	288
5.7.2	Comparing the predecessor SCL2VHDL with the current SCG2Circuit SCCharts circuit project	299
6.1.1	Ptolemy Vergil vs. KIELER Execution Manager (KIEM) simulation features	307
6.4.1	Graphical vs. Textual vs. Combined <i>Textical</i> Modeling	328
6.6.1	Semantic vs. syntactic validation	356
8.4.1	Similarities and differences of coroutines and SJL cooperative thread scheduling	410

Introduction

Nowadays, computer systems are present all around. There are different classes [PBEB07] of computer systems which often overlap and are hard to differentiate. Figure 1.0.1 illustrates these classes and gives examples.

PCs already can be found on many desks and are used to program and run simulations, calculations, and to store or query data. Such systems are termed *transformational systems*.

Transformational systems get some inputs from an *environment* which is a user or another program. From these inputs, the transformational system computes its outputs by applying a computation function to the inputs.

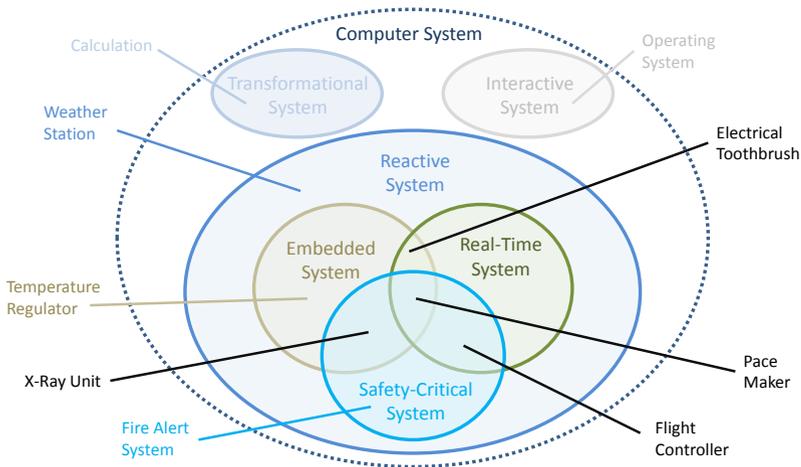


Figure 1.0.1. Computer system classes (based on [PBEB07])

1. Introduction

Transformational systems typically do not run continuously. They also have no time constraints imposed by the environment. Examples for such systems are calculators or weather simulations.

1.0.1 Definition (Environment). An *environment* of a system is the source of inputs and/or the target of outputs of a system.

A system interacts with its environment using its system interface.

1.0.2 Definition (System Interface). A *system interface* is the part of internal state variables that are manipulatable and/or visible from the outside (environment). Manipulatable state variables are called *inputs*, visible state variables are called *outputs*.

To ease programming of a computer, abstraction layers were invented, namely the Instruction Service Architecture (ISA) and operating systems (OS) [Tan09]. These abstraction layers made programs independent from a concrete hardware. Computer programs were also written in higher-level programming languages instead of using non-portable and hardly maintainable assembly code. Using programming languages has the benefit of portability and maintainability. However, this comes with a drawback: Since computers only understand low-level assembly code, a program written in a higher-level programming language has to be translated into a computer-understandable machine language. Hardware and OS-specific compilers [ASU07] were needed for this task and these compiles also have to be maintained.

Operating systems are an example of a different class of systems that became common: *Interactive systems*. They typically handle multiple executing programs at the same time and manage resources. Simultaneously, they often allow interaction with the user, i. e., their environment.

Like transformational systems, interactive systems compute a function, have inputs and outputs and do not have any time constraints imposed by the environment. However, they typically are non-terminating and continuously interact with their environment. Examples are flight booking systems, database systems in general, or operation systems.

Computers have spread in all manners of daily life. Today, computers control traffic lights, they control electronic toothbrushes, they even control

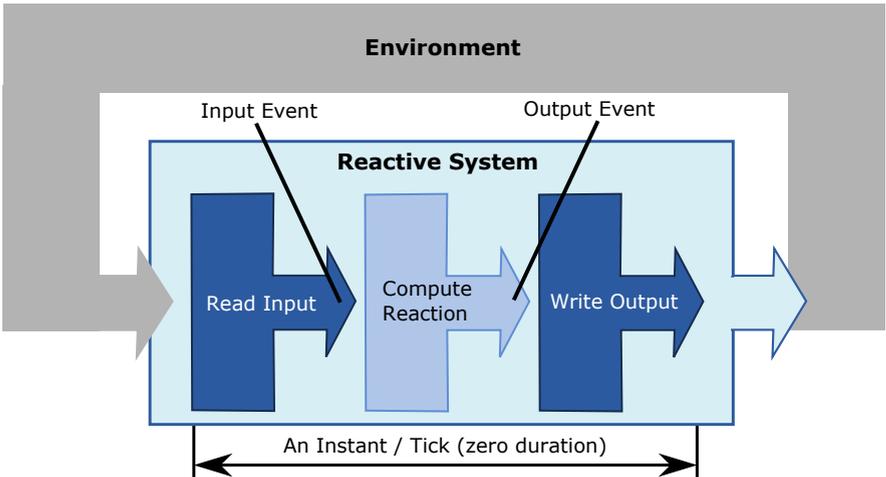


Figure 1.0.2. Reactive system cyclic control loop and synchronous tick abstraction (figure based on [MvH14])

our refrigerator and smart homes. A computer may not always be visible in the sense of a typical desktop PC or laptop. Still, a microprocessor is present to do computational tasks which are often control tasks.

Reactive Systems: Reactive systems [HP85] are today’s largest class of computer systems. According to Berry [Ber15], 98% of all manufactured CPUs are not built into a conventional PC but run in (often embedded) *reactive systems*. Such systems compute functions and have inputs and outputs. Like interactive systems, reactive systems also continuously interact with their environment. However, in contrast to interactive systems, reactive systems have to fulfill their computations in a certain amount of time which is typically imposed by the environment. Reactive systems often not only interact with but also control their environment or at least parts of it by their computed outputs. Inputs for a reactive system typically are sensor information. Examples for reactive systems are temperature regulators but also traffic light controllers, airbag controllers, pace makers,

1. Introduction

flight controllers, dome light controllers in cars, or electrical toothbrushes.

Key properties of reactive systems are: 1. Reactive systems compute a function, 2. reactive systems run continuously, and 3. reactive systems control their environment by their reaction. Reactive systems often have time constraints derived from the dynamics/physics of their controlled environment.

Figure 1.0.2 visualizes the control loop of a reactive system. In a cyclic fashion it reads inputs from the environment, computes a reaction by the given inputs, and writes the reaction, i. e., the outputs to the environment. The reactive system reacts with computed outputs, i. e., the reaction, to given inputs. Adapting the synchronous approach [BCE⁺03], we call one reaction computation a *tick* which is an abstraction of physical time and hence logically assumed to be computed in zero time (cf. Section 2.1 on page 19).

1.0.3 Definition (Tick). A *tick* is one reaction computation of a reactive system based on inputs and a potential internal state.

Embedded Systems: Embedded systems are a subclass of reactive systems that have been in use since the 1970s [LS11]. A simplified but common definition is that *embedded systems are computers that are not perceived as such*. More specifically, these are systems that are tightly embedded into their environment in order to control it. Typically, these systems have lots of interfacing with sensors and actuators of their environment and must work under harsh conditions such as dust, bad weather, under water, or extreme temperatures. Examples for these systems are temperature regulators, many controllers in cars, or mars robots.

Real-Time Systems: Real-time systems are another subclass of reactive systems. Real-time systems often are embedded systems, i. e., they are embedded into their environment. A *correct* real-time system must 1. compute the correct value and 2. additionally meet its timing requirements. Examples for real-time systems are traffic light controllers, airbag controllers, or flight controllers.

Safety-Critical Systems: Safety-critical systems are reactive systems that control environments whose overall correct behavior is crucial. The failure of such a system or parts of it typically means extreme danger to human lives or the environment. It is essential that these systems do not fail and guarantee safety. Hence, many real-time and embedded systems also are safety-critical systems. Examples for safety-critical systems are airbag controllers, pace makers, or flight controllers.

Safety-Critical System Development: Since computers are mostly programmed in higher-level programming languages, compilers are around to translate the human readable computer language into bytes that a micro-processor is able to execute.

Especially when it comes to safety-critical systems, it must be ensured that there are no errors in computer software that could lead to system failure. This typically means that safety and liveness properties must be ensured. Safety properties specify that *nothing bad ever happens*, e. g., failure states are unreachable. Liveness properties specify that *something good eventually happens*, e. g., a sent communication message is eventually received. [Lyn96, p. 216]

Liveness properties for synchronous systems often become (inverted) safety properties because *something good must happen in bounded time*. This is much stricter than *eventually* and hence it is observable and decidable whether *something good* has happened or not happened before a concrete point in time. Model checking helps to verify that safety properties hold.

This is the reason why software or at least critical parts of software for safety-critical systems is often model checked [BBF⁺01]. Verification of properties using model checking or similar techniques is time consuming and requires significant effort. Hence, validation and well-organized careful testing using simulations is often the only practical alternative in reality for most parts of a system.

Most reactive systems are still programmed in C today [Rus10, p. 23]. In order to be able to verify or validate safety and/or liveness properties, usually a model of the software or at least from certain software parts is required.

1. Introduction

Modeling is a process of gaining a deeper understanding of a system through imitation. [LS11]

Speaking about models of a software systems, the above statement also holds in the following sense: In order to define safety or liveness properties, usually only certain aspects of software, e. g., invariants, loop bounds, or value ranges are considered. As a result of focusing on certain parts, one automatically abstracts from other parts of software like interface code, glue code, low-level driver parts, or network communication. Summarizing, a software model emphasizes certain aspects or parts of software only. Principally, there are two scenarios for retrieving a software model:

1. Extracting a model afterwards from software [CDH⁺00] or from parts of a software implemented in a programming language. This is out of the scope of this thesis.
2. Starting with a model in the first place and deriving software code from the model using code generation techniques.

The general process of starting from models in the first place when designing and implementing systems is also termed Model Driven Engineering (MDE) or more specifically w. r. t. engineering/developing software components, Model Driven Software Development (MDSD).

MDSD has the advantage that the software development for a safety-critical system can start at a very high abstraction level which is also very close to the (control) problem that the software is supposed to solve in the end. The higher the abstraction level of a model, the more it is usually understandable also for non-computer scientists such as engineers or domain experts. Additionally, the model for the software is already available at the beginning of the development process and certain (safety) properties can already be checked to be met by the model.

Computer simulations [Fis95] are an established means to analyze the behavior of a system. There are basically two categories of computer simulations: On the one hand one wants to be able to predict and better understand physical systems and train humans to better interact with them (e. g., weather forecasts or flight simulators). On the other hand one aspires to emulate computer systems themselves prior to their physical integration in order to increase safety and cost effectiveness (e. g., airbag controller

or mars robot). Software models can also be used for a simulation in order to gain a deeper understanding of the dynamics of the software. Additionally, software simulations of a reactive system can be combined with a co-simulation of a model of the controlled environment that will be used to embed the reactive system later on.

States and Deterministic Concurrency: Reactive systems often not only consider inputs from an environment but have also an internal state that impacts on the computed reaction. Such system states should be expressed in a modeling language that is designed for reactive systems.

Additionally, reactive systems typically fulfill more than one concurrent task. E. g., a mars robot moves forwards while concurrently searching for obstacles to avoid getting stuck or an electrical tooth brush is concurrently observing its on/off button state and additionally a timer to stop brushing. A modeler should be able to express this inherent system concurrency also directly in the software model they build.

Reactive and especially safety-critical systems are expected to work reliably. One important factor for a reliable system is determinism. I. e., a system in any defined and reachable internal state S should always react to the same inputs I with the same outputs O and by changing to the same possibly other internal state S' . Specifying and implementing deterministic concurrency is a hard task as shown for threads by Lee [Lee06]. A modeling language for reactive and safety-critical systems should also be able to maintain determinism for specified concurrency.

Motivation: As discussed before, reactive systems require implementations with a deterministic behavior even if concurrency is involved as it is often the case. Since *synchronous languages* [BCE⁺03] cope very well with specifying and implementing deterministic concurrency, it seems reasonable to apply the main concepts from synchronous languages such as the separation of timing and functionality. Using a synchronous language for specifying and implementing software for reactive systems, one advantage is to be able to check such a program against system properties (cf. Section 1.0.3).

1. Introduction

Since reactive systems often involve states, a *statechart-based* [Har87] synchronous language, where the modeler is able to reflect system states directly with states of the modeling language, is reasonable. An advantage of a graphical modeling language is that models can even be read and understood by non-computer scientists such as domain experts. *SyncCharts* [And96] is one major representative of a synchronous and graphical statechart dialect.

Typically, synchronous languages have a common drawback. This is a steep learning curve especially for the majority of programmers that are used to imperative languages such as C or Java. These imperative languages naturally allow to specify sequential parts such as :

```
... if (!done) { ...; done = true; } ...
```

In contrast, synchronous languages tend to restrict the usage of variables/signals in the sense that for each tick a consistent value must exist. E. g., the code above would often be conservatively rejected because *done* would not be allowed to have both values false and true in one and the same tick. *Sequential constructiveness* [vHMA⁺13c] overcomes this typical drawback by allowing multiple value assignments to a variable per tick if these are ordered sequentially (non-concurrent). This seems naturally more appealing and intuitive for programmers that have experience in classical imperative languages and to still preserves the deterministic nature of synchronous languages.

For the above reasons, a statechart dialect such as SyncCharts is a reasonable choice for modeling reactive systems. However, a sequential constructive semantics is preferable in order 1. to accept more models, 2. to lower the learning curve, and 3. to be more intuitive for the majority of imperative language programmers.

Reactive systems often are safety-critical systems that require reliable software. Hence, reliable models need be build in the first place. A modeling environment, i. e., the tooling, should support the modeler in (a) building reliable models

- ▶ by providing abstraction mechanisms,
- ▶ by supporting the modeler to understand the language, its features, and whole models,

1.1. Contributions of this Thesis

- ▶ by providing a simulation for analyzing, understanding, and validating dynamic behavior,
- ▶ by offering optimizations to make models more compact and more readable, and
- ▶ by supporting fine-tuning to optimize the speed or size of the final code.

Such a tool chain must also be practically usable. Additionally, a compiler for a modeling language which targets safety-critical systems needs to (b) be reliable itself:

- ▶ It should be well-structured,
- ▶ it should be easy to understand every single part of it,
- ▶ it should be extensible such that new functionality can be added easily, and
- ▶ it should be maintainable such that parts that break, can be fixed in isolation, i. e., without breaking other parts.

Any new modeling, tooling, and compilation concepts that lead towards (a) a reliable tool chain and (b) a reliable compiler should be evaluated for a concrete modeling language. This language should include well known and common features in order to validate the practicality and the reasonableness of the overall approach.

1.1 Contributions of this Thesis

The major contributions of this thesis are threefold:

1. A novel synchronous statechart dialect termed *SCCharts* which for the first time leverages the sequentially constructive semantics and all of its benefits in a ready-to-use modeling language (cf. Chapter 3). *SCCharts* focus the development of safety-critical systems, since this primary use-case was considered when the language and its features were designed.
2. A novel interactive incremental compilation strategy based on model transformations termed *Single-Pass Language-Driven Incremental Compilation (SLIC)* (cf. Chapter 4). *SLIC* is specifically designed to facilitate

1. Introduction

building reliable compilers and models as required in the context of safety-critical system development.

3. An application of SLIC for compiling SCCharts language features. These language features are common and well known from other synchronous languages. SLIC-based model transformations were defined and are studied in this thesis by giving relevant examples and pseudocode (cf. Chapter 5).

The KIELER SCCharts tool (cf. Chapter 6) is implemented in order to validate the SCCharts language, the SLIC approach, and the proposed model transformations. The language, a generic interactive incremental compiler, and the model transformations are parts of this implementation. To show the practicality of the proposed approach, the reference implementation for the SCCharts modeling tools and compiler were used and validated in the context of medium-sized, real world safety-critical systems (cf. Chapter 7). Related projects (cf. Chapter 8) deal for example with integrating other synchronous languages such as Esterel into the KIELER SCCharts tools for validation purposes or with providing a Java-based runtime for a software-targeted compilation path of SCCharts.

1.2 Related Publications

The following publications contain parts of this thesis. A brief summary of each publication and its relation to this thesis is given. Additionally, a list of student theses advised by the author is given. The publications are sorted by importance w. r. t. this thesis.

1.2.1 Major Publications

The following listed publications are stronger connected to this thesis.

[MSvH14] Christian Motika, Steven Smyth, and Reinhard von Hanxleden.
Compiling SCCharts— A Case-Study on Interactive Model-Based Compilation.

1.2. Related Publications

In Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014), volume 8802 of LNCS, page 443–462, Corfu, Greece, October 2014

This paper introduces the novel interactive incremental model-based compilation approach termed *SLIC*. It further exemplifies *SLIC* for the use-case of incrementally compiling *SCCharts*. This paper summarizes the main results of this thesis that are discussed here in Chapter 4 and Chapter 5.

[*vHDM⁺14*] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquin Aguado, Stephen Mercer, and Owen O’Brien. *SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications*. In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’14), Edinburgh, UK, June 2014

This paper introduces the *SCCharts* language and its incremental features (cf. Chapter 3). It further summarizes the circuit-based and the priority-based low-level compilation paths for *SCCharts* (cf. Chapter 5).

[*vHDM⁺13c*] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquin Aguado, Stephen Mercer, and Owen O’Brien. *SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications*. Technical Report 1311, Kiel University, Department of Computer Science, December 2013

This technical report gives details on categorization of *SCCharts* features. It further explains how these features are incrementally compiled. This is described as an application of model transformations which are presented by examples. These transformations, in some cases an updated version, are discussed here in Chapter 5.

[*SMSR⁺15*] Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, Nis Boerge Wechselberg, Carsten Sprung, and Reinhard von Hanxleden. *SCCharts: The Railway Project Report*. Technical Report 1510, Kiel University, Department of Computer Science, August 2015

This technical report is a publication of the experiences when trying to solve realistic medium-sized real world problems with *SCCharts* and

1. Introduction

its tool chain. Results of this model railway practical student course are presented in Section 7.2 on page 367.

[RSM⁺16] Francesca Rybicki, Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. *Interactive model-based compilation continued — interactive incremental hardware synthesis for SCCharts*. In Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016), LNCS, Corfu, Greece, October 2016.

This paper reports on how to leverage the interactive incremental SLIC compilation for hardware synthesis. The interactive incremental synthesis of electrical circuits from SCCharts is part of the discussion in Section 5.7.2 on page 286 about reasonable targets. It further validates that the SLIC approach is not limited to software code generation.

[MvHH12] Christian Motika, Reinhard von Hanxleden, and Mirko Heinold. *Synchronous Java: Light-Weight, Deterministic Concurrency and Preemption in Java*. Technical Report 1213, Kiel University, Department of Computer Science, October 2012

[MvHH13] Christian Motika, Reinhard von Hanxleden, and Mirko Heinold. *Programming Deterministic Reactive Systems with Synchronous Java (Invited Paper)*. In Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2013), IEEE Proceedings, Paderborn, Germany, 17/18 June 2013

[MvH14] Christian Motika and Reinhard von Hanxleden. *Light-weight Synchronous Java (SJL) — An Approach for Programming Deterministic Reactive Systems with Java*. *Journal of Computing, Special Issue on Software Technologies for Embedded and Ubiquitous Systems*, 97(3):281–307, 2014

These three papers present *Synchronous Java* (SJ) and its lightweight successor SJL, which both bring language concepts and constructs well known from synchronous programming to Java. This can help achieving deterministic concurrent Java programs either as a target for code generation or as a language for programming deterministic (embedded) Java software directly. Synchronous Java is discussed in Section 8.4 on

page 407 as a software target for the priority-based SCCharts compilation path (cf. Section 5.4 on page 263).

1.2.2 Other Publications

The following listed publications are also partially connected to this thesis.

[vHMA⁺13c] Reinhard von Hanxleden, Michael Mendler, Joaquin Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. *Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation*. Technical Report 1308, Kiel University, Department of Computer Science, August 2013

[vHMA⁺13a] Reinhard von Hanxleden, Michael Mendler, Joaquin Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, and Owen O'Brien. *Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation*. In Proc. Design, Automation and Test in Europe Conference (DATE'13), page 581–586, Grenoble, France, March 2013

[vHMA⁺14] Reinhard von Hanxleden, Michael Mendler, Joaquin Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. *Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation*. ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design, 13(4s):144:1–144:26, July 2014

The above three papers introduce the sequentially constructive model of computation and compare it to other synchronous models of computation. It is the semantical foundation of SCCharts (cf. Section 2.6 on page 36).

[SSM⁺13] Miro Spönemann, Christoph Daniel Schulze, Christian Motika, Christian Schneider, and Reinhard von Hanxleden. *KIELER: Building on Automatic Layout for Pragmatics-Aware Modeling (Showpiece)*. In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'13), San Jose, CA, USA, September 2013

1. Introduction

This paper focuses advantages of *textical modeling*, i. e., textual modeling combined with a graphical view by utilizing automatic layout as a key-enabler. This kind of modeling is also proposed for modeling SCCharts (cf. Section 6.4.4 on page 327) and an integral part of the interactive compilation user story (cf. Section 4.2 on page 104).

[MFvH10] Christian Motika, Hauke Fuhrmann, and Reinhard von Hanxleden. *Semantics and Execution of Domain-Specific Models*. In 2nd Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2010) INFORMATIK 2010, GI-Edition - Lecture Notes in Informatics (LNI), page 891-896, Leipzig, Germany, September 2010

This paper reports on the results of my diploma thesis [Mot09]. This work dealt with defining semantics for high-level modeling languages based on model transformations and about using the Ptolemy tool to practically realize this for arbitrary semantics (*KlePto*). It further described how this technique can be leveraged to execute models. The simulation infrastructure discussed in Section 6.3 on page 316 and more specifically the *Execution Manager* discussed in Section 6.3.1 on page 316 are strongly rooted in that work. Also the extensions of the *KlePto* project described in Section 8.1.2 on page 386 are based on that work.

[MFvHL12] Christian Motika, Hauke Fuhrmann, Reinhard von Hanxleden, and Edward A. Lee. *Executing Domain-Specific Models in Eclipse*. Technical Report 1214, Kiel University, Department of Computer Science, October 2012

This paper reports on extensions to the *Execution Manger* and the *KlePto* project. Sections 6.3.1 and 8.1.2 summarize these extensions.

[RSM⁺15] Karsten Rathlev, Steven Smyth, Christian Motika, Reinhard von Hanxleden, and Michael Mandler. *SCEst: Sequentially Constructive Esterel*. In Proceedings of the 13th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE'15), Austin, TX, USA, September 2015

The paper on *SCEst* describes a conservative semantical extension of classical Esterel in the sense of the sequentially constructive model of computation. *SCEst* is addressed briefly in Section 8.2.4 on page 403 mainly as an alternative Esterel compiler for validation purposes.

1.2. Related Publications

[vHLMF12a] Reinhard von Hanxleden, Edward A. Lee, Christian Motika, and Hauke Fuhrmann. *Multi-View Modeling and Pragmatics in 2020 — Position Paper on Designing Complex Cyber-Physical Systems*. In *Pre-Proceedings of the 17th International Monterey Workshop on Development, Operation and Management of Large-Scale Complex IT Systems*, Oxford, UK, 19–21 March 2012

This paper conjectures about modeling in a few years from now, incorporating all known current state of the art techniques such as auto layout, model transformations, or visualizations of particular artifacts in transient views. These ideas in general have influenced the tooling of SCCharts as discussed in Chapter 6.

[RMvH11] Ulf Rüegg, Christian Motika, and Reinhard von Hanxleden. *Interactive Transformations for Visual Models*. In *3rd Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2011)* at conference INFORMATIK 2011, GI-Edition - Lecture Notes in Informatics (LNI), Berlin, Germany, October 2011

This paper reports on interactively transforming Esterel models to SyncCharts, the predecessor of SCCharts. The idea of utilizing interactive model transformations as used for compiling SCCharts (cf. Section 4.3 on page 111) was partly inspired by this contribution. The Esterel tooling (cf. Section 8.2 on page 393) is partly based on the implementation of this work. Results of this contribution are summarized in Section 8.3 on page 405.

1.2.3 Supervised Theses

Additionally, this thesis is based on parts of the following student theses supervised by the author:

[Ryb16] Francesca Rybicki, *Interactive Incremental Hardware Synthesis for SCCharts*, March 2016

This thesis deals with leveraging the interactive incremental SLIC approach for synthesizing hardware circuits (cf. Section 5.7.2 on page 298).

1. Introduction

[Nas15] Stanislav Nasin, *Transformation from SCCharts to Esterel*, October 2015

The topic of this thesis was to try transforming graphical SCCharts to textual Esterel programs (cf. Section 5.7.2 on page 296).

[Pei15] Lars Peiler, *Modeling Simulations of Autonomous, Safety-Critical Systems*, September 2015

In this thesis, SCCharts was evaluated as a language for building environment simulations.

[And15] Lewe Andersen, *Quadrocopter Flight Control Design using SCCharts*, September 2015

This thesis successfully utilized SCCharts for building a flight controller for a quadrocopter.

[Mac15] Felix Machaczek, *Collision Avoidance of Safety-Critical Real-Time Systems*, September 2015

The contribution of this thesis was a collision avoidance algorithm implemented in SCCharts for the quadrocopter project.

[Wec15] Nis Wechselberg, *Model Railway 4.0 - A Demonstrator for Interactive Timing Analysis*, March 2015

The topic of this thesis was to realize the renewal proposal (cf. Figure 7.1.1 on page 367) for the model railway demonstrator that is also used to validate the practicality of the SCCharts tooling.

[Rat15] Karsten Rathlev, *From Esterel to SCL*, March 2015

This thesis implemented the SCEst compiler by re-using large parts of the low-level SCCharts compilation chain (cf. Section 8.2.4 on page 403).

[SR14] Alexander Schulz-Rosengarten, *Framework zum Tracing von EMF-Modelltransformationen*, March 2014

This theses dealt with the tracing of model transformations as it can be used together with SLIC (cf. Section 4.2.3 on page 109).

[Joh13] Gunnar Johannsen, *Hardwaresynthese aus SCCharts*, October 2013

This thesis firstly evaluated how to synthesize hardware circuits from SCCharts and execute these on FPGAs (cf. Section 5.7.2 on page 298).

1.2. Related Publications

[Smy13] Steven Smyth, *Code Generation for Sequential Constructiveness*, July 2013

This thesis studied the low-level circuit-based compilation path and provided an implementation (cf. Section 5.5 on page 273) that is still used as an integral part of the SCCharts compiler.

[Har13] Wahbi Haribi, *A SyncChart-Editor based on Yakindu*, March 2013

The first Eclipse-based prototype SCCharts editor was the result of this thesis (cf. Section 6.4.2 on page 324).

[Dud12] Björn Duderstadt, *A Statechart Dialect With Sequential Constructiveness*, December 2012

This thesis studied and implemented parts of an industrial prototype for the SCCharts language including parts of a first sample compilation path.

[Rue11] Ulf Rüegg, *Interactive Transformations for Visual Models*, March 2011

The topic of this thesis was to interactively, stepwisely transform Esterel programs into graphical SyncCharts (cf. Section 8.3 on page 405).

[Car10] John Carstens, *Datenvisualisierung in grafischen Modellen*, September 2010

This thesis studied practical possibilities to visualize data in graphical models.

[Hei10] Mirko Heinold, *Synchronous Java*, September 2010

In this thesis, a first prototype of Synchronous Java was developed.

[Klo10] Paul Klose, *Beispiel Management in KIELER*, September 2010

This thesis contributed an example management for our Eclipse-based SCCharts tooling.

[Ame10] Torsten Amende, *Synthese von SC-Code aus SyncCharts*, May 2010

Part of this thesis was an implementation of the first priority-based compilation path for SyncCharts, the predecessor of SCCharts (cf. Section 5.4.2 on page 270).

[Han10] Sören Hansen, *Configuration and Automated Execution in the KIELER Execution Manager*, March 2010

1. Introduction

This thesis contributed extensions for the simulation infrastructure used in our Eclipse-based SCCharts tooling.

1.3 Outline

Chapter 1.3 introduces into synchronous programming. It outlines similar synchronous languages, related work, the sequentially constructive model of computation (MoC), and the intermediate control-flow graph representation (SCG) used for compiling SCCharts. Chapter 3 introduces the SCCharts language, its features, and its semantics. It presents the visual syntax as well as a textual one. An abstract syntax is finally introduced, which is the basis for the model transformation-based compilation approach. Chapter 4 introduces the Single-Pass Language-Driven Incremental Compilation (SLIC) approach that is based on model transformations. It compares the traditional and the interactive compilation user story including a brief summary of element tracing benefits for the SLIC approach. Chapter 5 gives details on the concrete SCCharts compilation process including the high-level model transformations for eliminating extended SCCharts features. Various examples are discussed and pseudocode for these transformations is given. It further illustrates design choices also for the low-level priority and circuit-based synthesis. Chapter 6 introduces the KIELER SCCharts implementation and its tooling that is used to study the language, the SLIC concepts, and the concrete model transformations. Chapter 7 reports on how SCCharts have been used in practice to validate the language and the compiler. A model railway demonstrator is introduced, which is a reactive embedded real-time system. A survey evaluation based on a student project which had the topic to build an SCCharts-based multi-train controller concludes the evaluation of practicality aspects. Chapter 8 summarizes other closely related projects that influenced the work on SCCharts and its KIELER implementation. Chapter 9 concludes and gives outlook on future work for SCCharts and SLIC-based interactive incremental compilation.

Background and Related Work

2.1 Synchronous Languages

This chapter presents the basics on synchronous programming and synchronous languages [BCE⁺03] in general which influenced the work on SCCharts. It also introduces the sequentially constructive MoC and a control-flow representation named SCG that is used as an intermediate representation to compile SCCharts.

Synchrony Hypothesis: All synchronous languages are based on the *synchrony hypothesis* [BC84] which assumes that a reactive system computes its reactions conceptually infinitely fast. Under this assumption, reaction intervals become single instances in time. These instances are called *ticks* as defined in Section 1.0.2 on page 3.

In short, the synchrony hypothesis states the following:

1. Time is divided into discrete ticks.
2. There is a zero computation time for a reaction, i. e., outputs are present instantly together with the inputs for each tick.

The rationale for this hypothesis is discussed in the following paragraphs.

Ticks: As shown in the lower part of Figure 2.1.1, physical time is divided into logical ticks. Hence, a tick logically is a point in time and time logically advances only from one tick to the next tick. The synchrony hypothesis states that inputs and outputs are present at the same point in time (tick) because of zero computation time. Of course this is not a practical

2. Background and Related Work

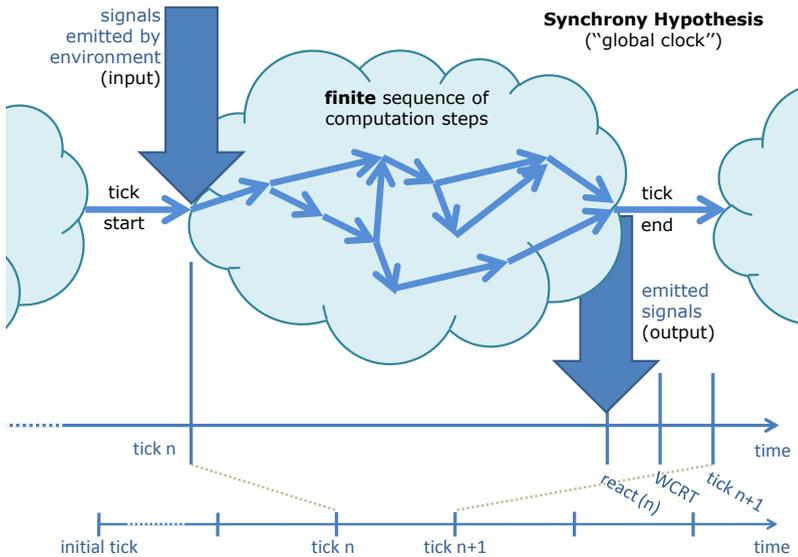


Figure 2.1.1. Synchrony Hypothesis: Discrete ticks and conceptually zero computation time illustration (figure based on [LM01])

assumption for the real system. In reality, outputs have to be computed first to become available as shown in the upper part of Figure 1.0.2 on page 3. Computation time is not zero in reality. Thus, the logical point in time is stretched as shown in Figure 2.1.1 to become a time interval. A tick is often also referred to as a *reaction cycle*.

The logical concept of synchrony is mapped into reality by requiring that the computed output, i. e., the reaction, is available strictly before the *next* tick. Based on this assumption, the interval time may be abstracted to represent a single point in time whenever the inputs and their computed reaction-outputs become available. This simplification is a key benefit for synchronous programs and it makes them far more easy to verify or validate.

The separation of functionality and timing is most significant for making good abstractions from the actual target hardware. This abstraction can be

2.1. Synchronous Languages

done because the assumption for any target hardware is that one tick can be computed *in time* before the next tick. In order to achieve regular clocked ticks, “in time” means that a bound on reaction computation time must be found that holds for every single tick. In Figure 2.1.1, this bound is denoted as the Worst Case Reaction Time (WCRT). The actual reaction computation of tick n , i. e., $\text{react}(n)$, is always less or equal to this bound. The WCRT can be derived by looking at the longest path of transitions (cf. Figure 2.1.1) which could be taken to compute a reaction.

The purpose of *WCRT analysis* is to find this bound or verify that a bound holds. Thus, on the one hand, this bound imposes constraints on the minimal tick time, i. e., it sets an upper bound how frequent a reactive system can compute consecutive reactions. On the other hand, if there are physical time constraints of the controlled environment, e. g., for an engine control, then this may limit the WCRT and has to be considered in the system implementation step or even already at design time. Timing analysis therefore is crucial for developing synchronous reactive programs. It is studied in various recent projects [MvHT09, FBSvH14].

Synchronous Languages: SyncCharts [And96] is a graphical statechart dialect with a synchronous semantics. It is introduced in more detail in Section 2.3. Esterel [Ber02] is a control-flow oriented textual synchronous language that has been already used for developing embedded hardware and software utilized in a commercial tool called *Esterel Studio*¹. Esterel can be seen as a textual variant of SyncCharts. It is discussed in Section 8.2 on page 393.

There are similar textual synchronous languages which focus data-flow semantics such as Lustre [CPHP87] or Lucy-n [MPP10]. The Safety Critical Application Development Environment² is a textual and graphical synchronous language, which was based on Lustre but was extended to express control-flow as well as data-flow. It is successfully used, e. g., in the aerospace industry and its qualified code generator satisfies highest safety standards like DO-178B [Rie13] level A. It is a commercially used synchronous language and tool suite for developing safety-critical systems.

¹<http://www.synopsys.com>

²<http://www.esterel-technologies.com>

2. Background and Related Work

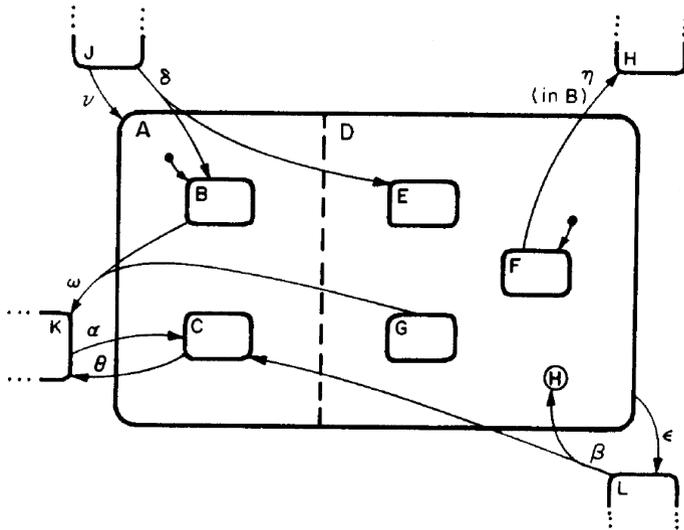


Figure 2.1.2. A Statecharts example from David Harel [Har87]

2.2 Statecharts

Statecharts [Har87] were invented by David Harel in 1987. Basically, Statecharts are *Mealy machines* [Mea55] with hierarchy, orthogonality, and broadcast-communication. Figure 2.1.2 shows an example Statechart.

Hierarchy: A state may contain further behavior inside, which is expressed as an inner Statechart itself.

Orthogonality: Orthogonality is expressed as a dashed line that separates concurrent Statecharts.

Broadcast: Broadcast (communication) events that may occur inside parts of the Statechart are visible inside their entire scope.

In contrast to SCCharts, Statecharts do not have a synchronous semantics. Additionally, Statecharts allow inter-level transitions, which are forbidden for SCCharts in favor of a clear semantics.

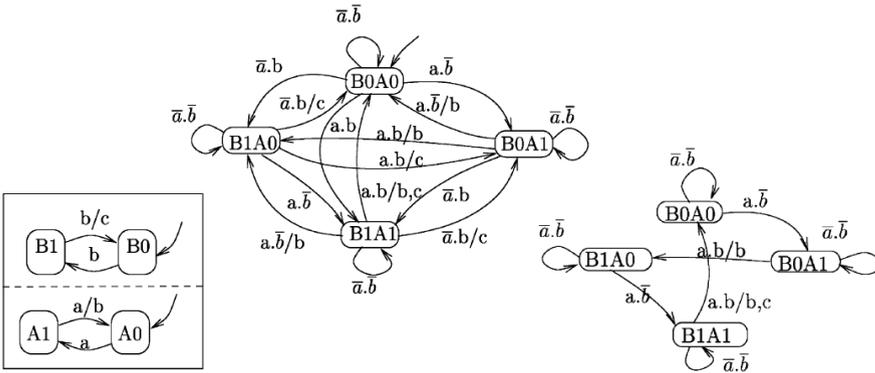


Figure 2.2.1. Argos parallel composition example of two-bit counter (left), its semantics (middle), and encapsulation (right) of communication signal b (from [MR01])

The Statecharts semantics and its STATEMATE tool implementation is intensely studied by Harel and Naamad [HN96].

2.2.1 Argos

The synchronous language Argos³ [Mar92] is considered one of the major predecessors of SyncCharts. Hence, Argos plays an historically important role also for SCCharts.

Argos basically consists of operators which allow hierarchical and parallel composition of mealy machines. Argos models can appear in a graphical syntax like Statecharts (cf. Figure 2.2.1). In contrast to Statecharts, Argos models do not have inter-level transitions and also other Statecharts features are missing. Some of these missing features could be described by combining primitive Argos operators. This is quite similar to SCCharts, which also are based on a small set of core features, while Extended SCCharts features are defined as a combination of these core features.

Figure 2.2.1 shows an Argos two-bit counter example for the paral-

³<http://www-verimag.imag.fr/~altisen/DSTAUCH/ArgosCompiler>

2. Background and Related Work

lel composition of two boolean mealy machines (left) and its semantics (middle). In Argos, parallel composition is used to describe the composition of *independent* concurrent parts which does not involve any implicit synchronization as it is the case for SCCharts.

When concurrently being in states A1 and B1, and the input a is present but b is not, then the transition from A1 to A0 is taken which emits b. But the concurrent part will remain in state B1 and not react to this emission b as it would be the case for SyncCharts and SCCharts.

Note that Argos is also capable of specifying instantaneous communication by using encapsulation of dedicated synchronization signals. This is done on the right side of Figure 2.2.1 for the communication signal b where certain transitions have been removed. Only those transitions are left for which the synchronization described in the encapsulation applies to. Details on Argos can be found elsewhere [Mar92, MR01].

2.3 SyncCharts

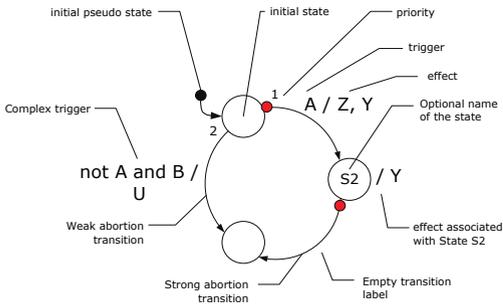
SyncCharts [And96] were invented by Charles André in 1996. SyncCharts are basically Statecharts with a synchronous semantics. SyncCharts often are seen as a graphical variant of Esterel. This is quite a good approximation because of an extremely similar set of language features [PTvH06] and the same MoC.

SyncCharts can be seen as a predecessor of SCCharts. SCCharts borrow most of their visual syntax from SyncCharts and extend their semantics. Hence, every valid SyncChart is a valid SCChart with the same⁴ meaning.

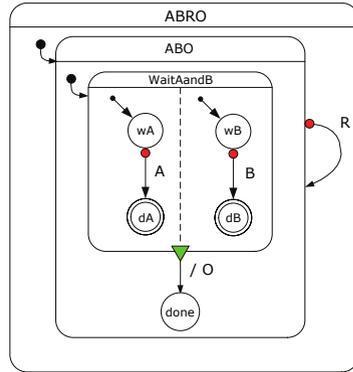
SyncCharts were previously integrated into KIELER with different compiler implementations which also served as a basis for ideas of today's SCCharts interactive incremental compiler. The syntactical and semantical details of SyncCharts are introduced using the ABRO example.

⁴To be precise, there is a difference due to current design decisions for aborts and entry actions: Entry actions in SCCharts are not preempted while for SyncCharts they can be preempted, see discussion in Section 5.2.5 on page 140. But as the discussion shows, it would be simple to change or extend the semantics of SCCharts regarding this issue.

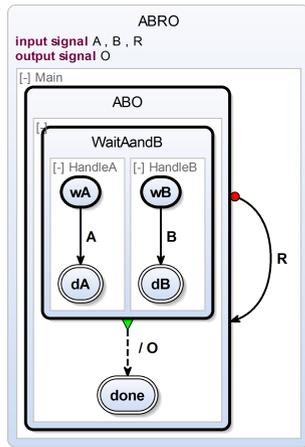
2.3. SyncCharts



(a) SyncCharts FSM notation



(b) ABRO SyncChart



(c) ABRO SCChart

Figure 2.3.1. SyncCharts notation and example from Charles André (partly from [And03])

2. Background and Related Work

2.3.1 ABRO

SyncCharts consist of states, initial and final states, transitions, signals (events), hierarchy, concurrency, and modularity. An example SyncChart, the ABRO *hello world* of synchronous programming is shown in Figure 2.3.1b next to some self-explaining basic concrete syntax excerpt of SyncCharts (cf. Figure 2.3.1a). Figure 2.3.1c shows the SCCharts version of ABRO using a very similar concrete graphical notation. ABRO shows the hierarchy and concurrency concept borrowed from Statecharts. Final states have a double border, initial states have an *initial connector* (or a bold border in SCCharts). A, B, and R are inputs and O is an output. All are synchronous signals that can be either present or absent in a tick.

The behavior of the ABRO SyncChart (cf. Figure 2.3.1b) is as follows: Concurrently, ABRO waits for A and B to become present in states *wA* and *wB*. Once both signals have become present, which can also happen in the same tick, ABRO ends up in the “done states” *dA* and *dB*. The transition with the green triangle is called *normal termination*. It is triggered implicitly and immediately when all concurrent regions of the originating state have reached a final state. Hence, when *dA* and *dB* are entered, immediately the normal termination transition is taken. It emits the output signal O and leads to the done state. ABRO can be reset by the reset signal R.

Note that the reset transition marked with a red dot and triggered by R is a *strong abort*. This means that all (immediate) behavior of the originating state is preempted in the tick when this strong abort transition is taken. For example this means for ABRO being in states *wA* and *dB* with present input signals A and R that the state change from *wA* to *dA* is preempted and O is also not emitted. Furthermore, the reset transition leads to re-entry of state ABO and hence ABRO afterwards waits in states *wA* and *wB* again for the signals A and B.

2.3.2 Advanced SyncCharts Features

Numerous basic SyncCharts features are shown in Figure 2.3.1a and in the ABRO example (cf. Figure 5.1.1a on page 117). However, SyncCharts has more advanced features such as *Entry* action, *Exit* action, or *Suspension* as

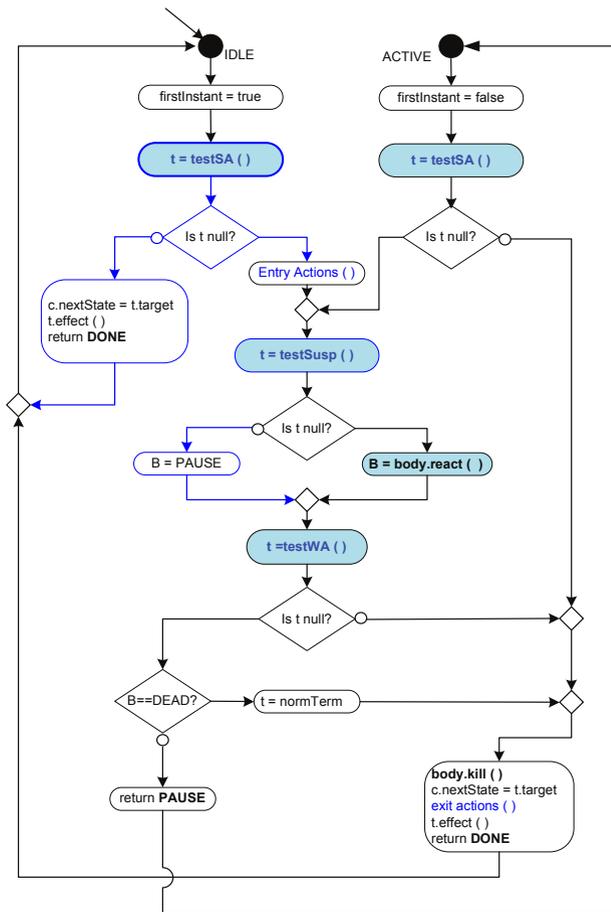


Figure 2.3.2. A Reactive Cell of SyncCharts visualizing semantics of several advanced SyncCharts features (adapted from [And03]).

introduced in detail by André [And03]. Figure 2.3.2 gives an overview of how these features integrate with other features such as weak and strong abort and a possible body (cf. Figure 2.3.2). It visualizes the inner behavior of

2. Background and Related Work

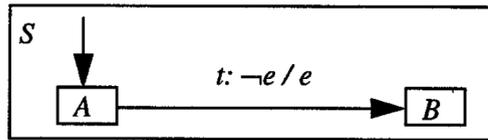


Figure 2.3.3. Statechart semantic problem example: Possible inconsistency between trigger and action (from [vdB94])

a SyncCharts superstate, when being executed. Superstates are sometimes also referred to as *macrostates*. Inner states of a superstate are often called *substates*.

A *Reactive Cell* represents a state in a so called *State Transition Graph* (STG). The structure of any SyncChart is an alternation of superstates and STG. Each STG consists of typically several Reactive Cells and is graphically represented by a concurrent region.

Hence, a superstate has a non-empty set of STGs. The schematics of a Reactive Cell is shown in Figure 2.3.2. One Reactive Cell per STG is marked as initial. A Reactive Cell has possibly but not necessarily outgoing transitions of type weak abort, strong abort, or normal termination. It also has a body that can either be a *simple state* or a superstate.

In an STG there may be many Reactive Cells that initially are IDLE but can become ACTIVE for a while when they are entered by an outgoing transition from another Reactive Cell. The figure shows details under which conditions for example the body or Entry Actions are executed. t is an outgoing transition of the Reactive Cell. All outgoing transitions are tested in the following order: 1. Test for strong aborts ($\text{testSA}()$), 2. execute the body if no strong aborts occurred ($\text{body.react}()$), 3. test for weak aborts ($\text{testWA}()$), 4. test for normal termination if also no weak aborts occurred ($B == \text{DEAD}?$). More details are covered in a technical report [And03].

2.4 Other Statechart Dialects

Von der Beek [vdB94] compares various different Statecharts semantics and studies 19 specific problems including approaches to solve these. To name

2.4. Other Statechart Dialects

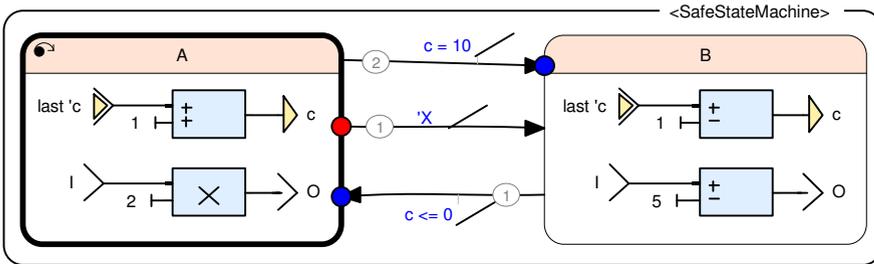


Figure 2.4.1. A simple SCADE automaton combining control-flow with data-flow (from [Tra10])

just some interesting examples these include perfect synchrony hypothesis, inter-level transitions, state references, negated events, self-termination, instantaneous states, and determinism. SCCharts deals with most of the problems in one or the other proposed way. For example SCCharts follow the synchrony hypothesis, SCCharts forbid inter-level transitions incorporating self-termination. SCCharts have a reference state feature, allow the negation of trigger events, support instantaneous states (also called “transient states”), and come with deterministic concurrency.

An interesting problem studied is the one sketched in Figure 2.3.3 where von der Beek identified that Statecharts dialects often have a problem with inconsistencies between a trigger and effects of a transition. SCCharts overcomes this problem by taking profit of its underlying sequentially constructive semantics. In the shown example, the trigger $\neg e$ is ordered strictly before the effect e of the transitions. This means that e is allowed to be written in the effect (sequentially) *after* it has been read in the trigger. Hence, sequentiality is given precedence to solve these kind of problems if they are of non-concurrent nature. The SCCharts behavior is classified as *local consistency* by von der Beek.

As pointed out by Maraninchi [MR01] there is an overwhelming number of statechart dialects around and the comparison of SCCharts to all of them is not reasonable. Hence, in the following only a few dialects are considered.

2. Background and Related Work

2.4.1 SCADE State Machines

The Safety Critical Application Development Environment (SCADE) is a tool for graphically modeling synchronous data-flow. Lustre [HCRP91] served as the underlying data-flow language in the beginning where the SCADE language was a graphical representation of Lustre [CPP05]. Later, SCADE evolved to an own, independent language *textual SCADE*, where control-flow and data-flow parts can be arbitrarily mixed with each other. Another specialty of SCADE is its certified code generator. It produces C code, which for example is commercially used in the aerospace industry.

For modeling control-flow, SCADE offers a synchronous state machine dialect that is partly similar to SyncCharts and hence to SCCharts. One specialty of SCADE's state machines is that these not only allow to model control-flow inside a state but also data-flow. In Figure 6.1.3 on page 308, for example, the state `ABRO_state` contains more control-flow, namely the `ABO_SM` region with the `ABO_state`. In Figure 2.4.1, for instance, the state `A` contains data-flow that writes to the outputs `c` and `o`.

Comparing SCADE state machines with SCCharts, the following specialties and limitations exist according to the SCADE language reference manual [Est11]:

Synchro transitions: Concurrent control-flow, i. e., several concurrent regions as in `ABO_state` (cf. Figure 6.1.3 on page 308) can be joined by so called *synchro transitions*. Syntactically, synchro transitions have a green triangle. A synchro transition is the semantic counterpart in SCADE to SCCharts's termination transition or to SyncCharts's normal termination.

Initial and final states: Initial states are syntactically marked by a small *arrow connector* in the upper left corner of a state. Initial states can also be final states. There is exactly one initial state per state machine. In SCCharts, initial states are marked by a bold border. Initial states can also be final states and there is also exactly one initial state per region in SCCharts.

Final states and termination: Final states are *useless* if there exists no outgoing synchro transition in the direct upper hierarchy level. The same is true for SCCharts: If there is an outgoing termination transition and one of the direct inner regions has no final state, the termination transition can never be taken.

2.4. Other Statechart Dialects

Root termination: A final state is useless in the root level, i.e., the top level automaton. In contrast, an SCChart's root state always has an implicit termination transition that leads to termination of the SCChart. Hence, a final state in the root level may trigger this implicit termination transition.

Empty termination: If a synchro transition appears on a state with no state machine in it, then the synchro transition will always be fired. In SCCharts, a termination transition is only allowed for superstates not for simple states.

Strong self-termination: Triggers of weak transitions are allowed to reference variables and signals that are written in the scope of their body. For triggers of strong transitions this is not allowed. The same is true for SCCharts and SyncCharts where weak transitions are called weak abort transitions or simply *weak aborts* and strong transitions are called *strong aborts*.

Reachability: All states should be reachable by at least one path of transitions from some initial state. The same is true for SCCharts.

Priorities: There is a textual order of outgoing transitions in a SCADE state machine that defines the order in which transitions are checked. Additionally, strong transitions are checked first, then weak transitions, and then synchro transitions. Syntactically, this *priority* is visible as a circled number arrow start decorator (cf. Figure 2.4.1). In SCCharts, the priority is derived from the textual order, which in contrast is directly visible in the SCT model file. Also in SCCharts, strong aborts should have the highest priorities over weak aborts and terminations (cf. Section 3.2.3 and Section 3.2.3). However, because weak and strong abort transitions are Extended SCCharts features, this is only enforced by validation rules and there is no implicit *group order* like there is in SCADE. Furthermore, SCCharts terminations are allowed to have a higher priority than weak aborts.

Dynamic semantics: A SCADE state machine distinguishes between *selected* and *active* states. The selected state is the state where strong terminations are examined and the active state is the state where the body and weak transitions are examined. Typically, the selected and the active state is

2. Background and Related Work

one and the same state. Starting with the selected state, there is at most one active state in a synchronous tick. That is the selected state if no strong transition was taken. Otherwise, it is the target state of a strong transition. In SCADE only one transition can be taken per tick. If a strong transition has been taken, the target state's body can be executed but the target state cannot be left in the current and only in the next cycle. This is only possible in consecutive cycles. This differs from SCCharts and SyncCharts where a *finite* number of immediate transitions are allowed to be taken during a tick reaction computation. Hence, in SCCharts and SyncCharts more than one (immediate) transition can be taken per tick and more than one state can be active.

Another difference is, when taking weak aborts in SCCharts, then instantaneous behavior such as immediate transitions or target state actions may be taken or executed in the same tick. This is not the case in SCADE and the target state of a taken weak transition is selected only for the next tick. Syntactically, this is denoted as a blue dot transition arrow end decorator (cf. Figure 2.4.1).

This *deferred* behavior of weak abort transitions in SCADE gave inspiration for the *deferred transition* feature in SCCharts where also immediate behavior of the target state is preempted in the tick when entering this state. Because the semantics is not exactly the same, e. g., as SCADE does not have immediate transitions/actions, we decided to use a red dot transition arrow end decorator instead of a blue one because a red dot denotes preemption in SCCharts.

There exist further differences and specialties to/of SCADE as for example a special *restart* transition. Regarding the SCADE portion, the interested reader is referred to the SCADE language reference manual [Est11].

2.4.2 UML Statecharts

UML Statecharts [Dou99] represent the behavior of a graphically modeled system. Figure 2.4.2 shows a UML Statecharts example⁵ that represents

⁵<http://www-sop.inria.fr/members/Charles.Andre/CAdoc/ESINSA/Chap7-StateMachine.pdf>

2.4. Other Statechart Dialects

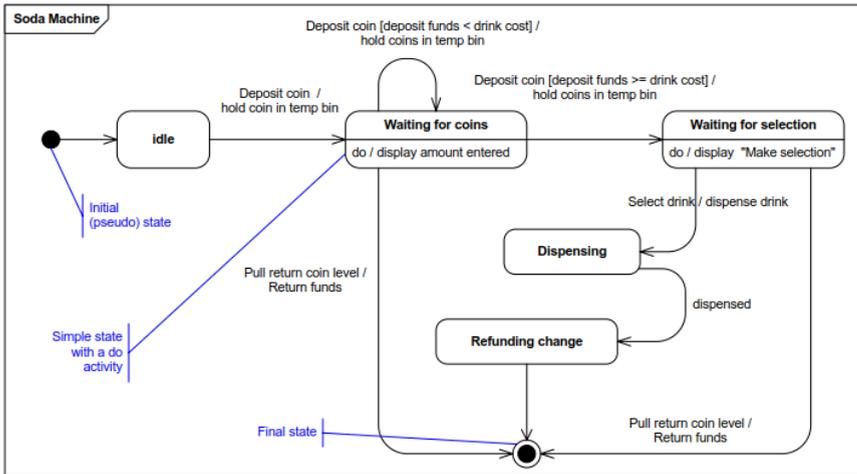


Figure 2.4.2. UML Statecharts example (from Charles André)

a soda machine system. UML state machines are also based on Harel's Statecharts formalism. As both SCCharts and UML state machines are based on Statecharts, many language constructs are comparable.

However, the semantics of UML state machines fundamentally differs to the one of SCCharts. UML state machines do not follow the synchronous MoC. In contrast, UML specifies an *event pool* and a so called *run-to-completion* semantics [Obj15]. All events are 1. detected, 2. dispatched, and 3. processed, one at a time. The order in which this happens to events is not specified. However, *run-to-completion* means that the next event dispatch (2.) must wait until the previous event is fully processed (3.). Unfortunately, UML state machines have been found to have numerous *semantical variation points* [FSKdR05] which in practice hampers the usage as a clear and unambiguous behavioral system specification language.

2. Background and Related Work

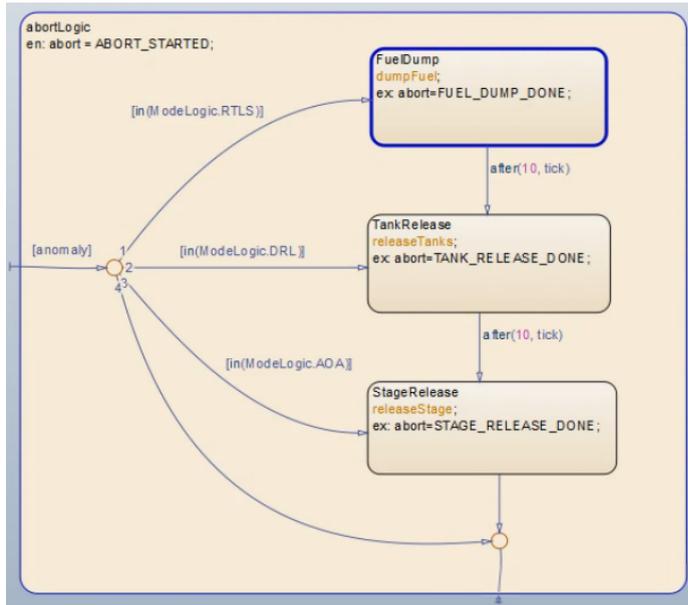


Figure 2.4.3. Stateflow example (from Mathworks website)

2.4.3 Stateflow

Stateflow⁶ is a commercial state machine modeling and simulation environment, which is typically used by control engineers. Stateflow is closely coupled with MATHLAB Simulink⁷ which is a data-flow modeling language for various kind of systems. Simulink is widely used in the automotive domain. An operational semantics for Stateflow was studied by Hamon [HR04]. Figure 2.4.3 shows a rocket example modeled in Stateflow as taken from the vendor's website.

Like SCCharts, Stateflow also supports basic statechart elements including entry, during, and exit actions for superstates. Contrary to SCCharts, Stateflow supports inter-level transitions and does not have a synchronous

⁶<http://www.mathworks.com/products/stateflow>

⁷<http://mathworks.com/products/simulink>

2.5. Code Generation from Statecharts

semantics. Instead, the Stateflow semantics is event-based. A Stateflow chart always has an active state. In case an event occurs in the environment, the active state of the chart is executed. During execution of a state, a transition may be taken which may make another state active.

2.4.4 Modechart

Modechart [JM94] is a statechart-based specification language for real-time systems with a graphical implementation. Modechart has a semantics based on Real-Time Logic (RTL). Modecharts can be translated into RTL formulas in order to reason about system properties. However, in contrast to SCCharts, Modechart does not follow the synchronous abstraction and its separation of timing and functionality.

2.5 Code Generation from Statecharts

For SyncCharts there exists a dedicated experimental compiler called SCC⁸. This monolithic compiler follows a common structure having a frontend for an XML representation of SyncCharts and, e. g., a backend for C. Other compilation paths for SyncCharts involve the translation to the synchronous Esterel language as presented by André [And95].

Graphical SCADE models, including state machines, are compiled to textual SCADE, a variant of Lustre. From there, C code is produced that can be cross-compiled for the desired target platform. For each data-flow node the generated C code always contains a reset and a step function. These functions also exist on the top-level for the whole system in order to interact with the environment. SCADE generates two data structures, an input and an output structure. The reset function initializes the generated output data structure with the default values. Before calling the generated step functions, the input data structure should be filled. According to these values, the step function will calculate the outputs in a typically nested call to all hierarchically contained nodes.

⁸<http://julien.boucaron.free.fr/i3s>

2. Background and Related Work

Similarly to the Core SCCharts features, Argos as well builds upon a simple core language with few features which may make Argos also suitable to apply an interactive incremental SLIC-based compilation. The currently available Argos compiler produces Lustre code.

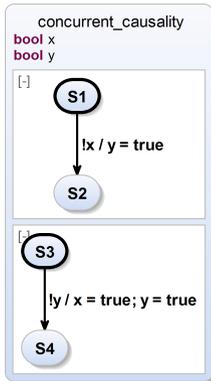
Code generation for general statechart models is often based on UML state machines. It is studied by Ali and Tanaka [AT00], Allegrini [All02], or Pinter [PM03]. E.g., Pinter proposes to use a *state pattern* to derive Java code from UML Statecharts. Agrawal, Simon, and Karsai [ASK04] present a proposal to use graph transformations to turn Stateflow charts into hybrid automata. The $\langle HOE^2 \rangle$ action language [LCFH14] is meant to extend hierarchical state machine dialects and to equip them with *data parallelism* and operation on compound data. The purpose is to preserve expressiveness of the state machine dialect and to capture data organization including a path to synthesize efficient low-level imperative code.

2.6 Sequential Constructiveness

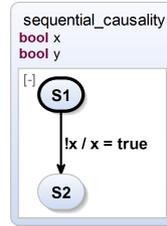
As the name Sequentially Constructive Statecharts (SCCharts) suggests, SCCharts are inherently based on the *sequentially constructive* semantics. In short, sequential constructiveness extends the “Berry constructive” semantics of SyncCharts or Esterel by deterministically letting sequential ordering dominate non-concurrent writes to variables/signals. SCCharts borrow the visual syntax from SyncCharts where all language elements keep their meaning (modulo different design decisions, cf. footnote on page 24). Since sequential constructiveness is a conservative extension to the constructive MoC of SyncCharts (or Esterel), all valid SyncCharts are valid SCCharts with the same semantics and input/output behavior.

The following will only briefly introduce the sequentially constructive MoC. A deeper and more formal explanation can be found elsewhere [vHMA⁺13c]. Figure 2.6.1 illustrates the advantages of this semantic extension that was motivated by accepting (like Figure 2.6.1b) more models as valid and unambiguously schedulable than under the (Berry) constructive semantics. Hence, fewer models are rejected (like Figure 2.6.1a) as being (still) not schedulable under the sequentially constructive semantics.

2.6. Sequential Constructiveness



(a) Concurrent causality cycles cannot be unambiguously scheduled: Both the SyncCharts and the SCCharts compiler must reject these models.



(b) Sequential causality can be unambiguously scheduled by giving precedence to sequentiality: A SyncCharts compiler will reject but an SCCharts compiler will accept these models.

Figure 2.6.1. Advantages and limits of SCChart’s sequentially constructive semantics as a conservative extension to Esterel’s and SyncChart’s constructive semantics (from [MSvHM13])

Note that for all SCCharts that are not schedulable and rejected under the sequentially constructive semantics, their SyncCharts counterparts likewise are rejected under the (Berry) constructive semantics.

Concurrent Causality

In Figure 2.6.1a, x is read in the upper thread and written in the lower thread. At the same time, y is read in the lower thread and written in the upper thread. This is a *concurrent dependency cycle*, i. e., a dependency cycle across concurrent regions that cannot be unambiguously scheduled. Hence, such models are rejected in both cases, under the constructive semantics of SyncCharts/Esterel and under the sequentially constructive semantics of SCCharts.

2. Background and Related Work

Sequential Causality

In contrast to Figure 2.6.1a, Figure 2.6.1b shows a model where x is read and written in the same thread. This is typically still considered a dependency cycle but it is not a concurrent dependency cycle. Hence, it is possible to break the cycle by giving precedence to sequentiality, i. e., the read of x occurs sequentially *before* the write to x and both can be unambiguously scheduled. Informally speaking, sequentiality can impose scheduling constraints with the effect of accepting more models under sequentially constructive MoCs that would otherwise have been rejected due to dependency cycles appearing without these constraints under the constructive MoCs. Models such as the one shown in Figure 2.6.1b are therefore accepted as valid and deterministic SCCharts but rejected as cyclic SyncCharts or cyclic Esterel programs.

Mixed Causality and Confluence

Note that also in the model of Figure 2.6.1a, y is both read and written in the lower region. This alone is sequentially causal again and the read and write to y can be unambiguously ordered by sequentiality such that this is not a problem under the sequentially constructive semantics. However, a concurrent write to y typically is challenging also under a sequentially constructive MoC with an exception of *confluent* writes. These are writes that can be executed in any order leading to the same result. In this example both concurrent writes to y set y to true and hence are confluent such that these concurrent writes are also no problem under sequential constructiveness. As explained earlier, the only problem here is the concurrent dependency cycle.

The Initialize-Update-Read (IUR) Protocol

The Initialize-Update-Read (IUR) protocol is an integral part of the sequentially constructive MoC [vHMA⁺13c]. For sequential parts of an SCChart the modeler explicitly models the control-flow (schedule). For concurrent parts of an SCChart the modeler only implicitly models the control-flow due to data dependencies. The scheduler produces an explicit control-flow

2.6. Sequential Constructiveness

Table 2.6.1. SCCharts transition trigger and effect examples inducing (concurrent) data dependencies considered for IUR scheduling w. r. t. one variable (adapted from [MSvHM13, vHMA⁺13c])

Example	Type	Scheduling
... / L = false	Init: Absolute Write	1. Absolute writes w. r. t. a variable are scheduled first. Multiple absolute writes typically result in a scheduling conflict unless they are confluent, i. e., the same value is written.
... / L = L true	Update: Relative Write	2. Relative writes w. r. t. a variable are scheduled after absolute writes and before reads. Relative writes require the <i>combination function</i> (here OR) to be associative and commutative. Hence, multiple relative writes are always confluent and do not result in scheduling conflicts.
L / ...	Read	3. Reads w. r. t. a variable are scheduled last but as the value is assessed already the order is not important and still leads to deterministic behavior.

(schedule) by respecting these data dependencies in a deterministic way as follows.

In concurrent parts of an SCChart, the IUR protocol simplifies reasoning about the concurrent scheduling that will take place. As the name suggests, the IUR protocol determines the order of initializations, updates, and reads w. r. t. a variable. Typically, in the constructive semantics (Esterel or SyncCharts) writes are scheduled before reads. This is also true for the sequential constructiveness. However, writes are sub-categorized into initialization-writes and update-writes. Hence, the IUR protocol additionally ensures that all initializations are always scheduled before any updates to that variable. Initializations are also called *absolute writes* and updates are also called *relative writes*. Examples for absolute and relative writes are given in Table 2.6.1.

2. Background and Related Work

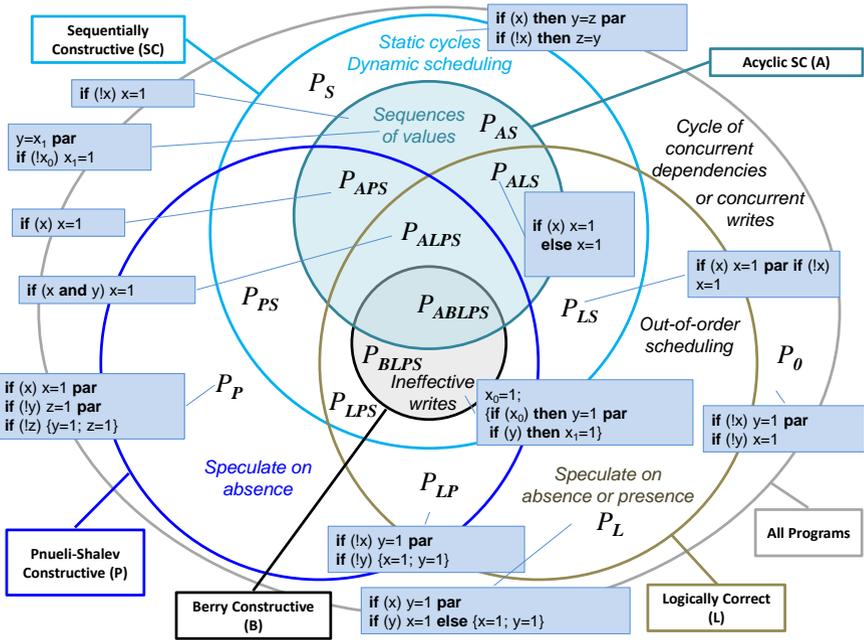


Figure 2.6.2. Relationships of synchronous program classes (adapted from [MSvHM13, vHMA⁺13c])

Non-confluent, concurrent absolute writes (such as multiple absolute writes with different values) lead to rejection of the SCChart because no distinct deterministic execution schedule can be derived. Confluent, concurrent absolute writes (such as multiple absolute writes with the same value) can be arbitrarily scheduled and executed with deterministic result and hence do not lead to rejection. Non-concurrent absolute writes (confluent or non-confluent) are sequentially scheduled as given by the modeled control-flow.

Relative writes require a combination function that needs to be associative and commutative. E. g., this could be AND or OR for boolean data types and ADD or MULT for integers. Hence, the order of execution of

2.7. The Sequentially Constructive Graph (SCG)

relative writes with a combination function does not affect the resulting value. This means that relative writes are always confluent.

Concurrent reads are scheduled after all absolute and relative writes. As reads do not change the value, multiple reads are allowed (in any order) and do not cause any scheduling conflicts.

2.6.1 Synchronous Programming Classes

The previous sections made clear that the class of sequentially constructive programs, e. g., the class of valid SCCharts, is larger than the class of Berry constructive programs, e. g., the class of valid SyncCharts. Figure 2.6.2 visualizes this fact by comparing different synchronous semantics and showing their relations. SCCharts are subsumed under Sequentially Constructive (SC) and SyncCharts or Esterel are subsumed under Berry Constructive (B). Note that B is fully contained in SC. This represents the fact that all valid SyncCharts are valid SCCharts and that any invalid SCChart cannot be a valid SyncChart. Note that the current KIELER SCCharts compiler is limited to handling acyclic SCCharts, i. e., SC(A). A more detailed comparison and description of Figure 2.6.2 is given elsewhere [vHMA⁺13c, vHMA⁺13b].

2.7 The Sequentially Constructive Graph (SCG)

For compiling SCCharts a control-flow graph representation, the Sequentially Constructive Graph (SCG) [vHDM⁺14, MSvHM13], was chosen as an intermediate format to ease down-stream compilation.

An SCG is a pair (N, E) , where N is a set of statement nodes and E is a set of control-flow edges. The node types are *entry* and *exit* connectors, *assignments*, *conditionals*, *forks* and *joins*, and *surface* and *depth* nodes that jointly constitute tick-boundaries. The edge types are *flow* edges (solid edges), which denote instantaneous control-flow, *pause* tick boundary edges (dotted lines), and *dependency* edges (dashed lines), added for scheduling purposes.

Hence, an SCG consists of only five basic constructs that are shown in Figure 2.7.1.

2. Background and Related Work

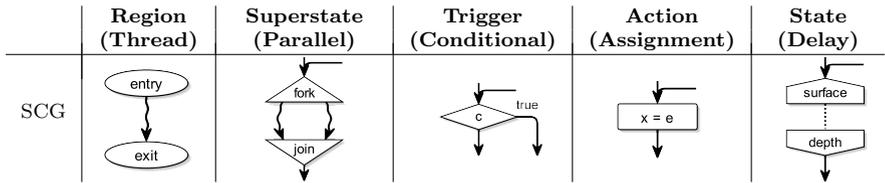


Figure 2.7.1. Elements of a Sequentially Constructive Graph (SCG) (from [vHDM⁺14])

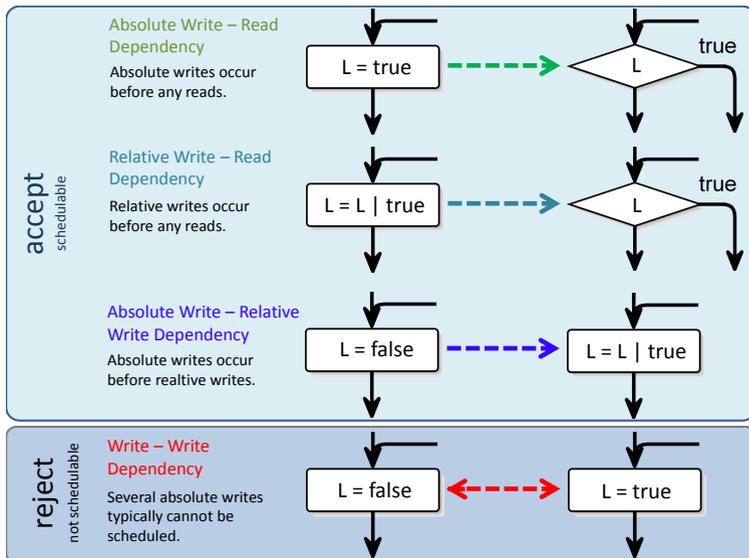


Figure 2.7.2. Different types of SCG dependencies for concurrent read and/or write accesses (based on [MSvH14])

2.7.1 Explicit Data Dependencies

Typically, data dependencies exist in concurrent parts of an SCChart (cf. Section 2.6). During compilation, these data dependencies also emerge in the derived SCG. These dependencies are essential for statically ordering SCG elements, e. g., for the circuit-based low-level synthesis (cf. Sequentialize

2.7. The Sequentially Constructive Graph (SCG)

SCG in Figure 5.0.1 on page 114). The dependencies are also essential for calculating priorities for the priority-based low-level synthesis. Additionally, these dependencies are helpful to detect dependency cycles for rejecting not schedulable SCCharts/SCGs.

In short, there are four types of concurrent write/read access dependencies as depicted in Figure 2.7.2: (1) absolute write – read dependencies, (2) relative write – read dependencies, (3) absolute write – relative write dependencies, and (4) write – write dependencies. These dependencies are already encoded in the model implicitly when constructing the SCG from an SCCharts model as transition triggers and effects (cf. Section 2.6). However, making these data dependencies explicit in the SCG model facilitates further compilation steps, e. g., the scheduling or calculation of priorities.

The first three types of dependencies shown in Figure 2.7.2 can easily be ordered according to the IUR protocol introduced earlier in Section 2.6. The last, a (concurrent) write – write dependency, may lead to rejection of the model because the behavior of the model depends on the resolution of this race condition which is clearly is a non-deterministic choice. Note that this observation does not hold for confluent write – write dependencies where the same value is written because the order is unimportant in this special case. Also, recall that write – write dependencies are also not a problem for sequentially ordered, non-concurrent parts.

2.7.2 Abstract Syntax

Figure 2.7.3 shows the meta model of SCGs. An SCGraph contains an unlimited number of nodes that are of type Node. The following subtypes of nodes exist: (1) Assignment, (2) Surface, (3) Depth, (4) Conditional, (5) Fork, (6) Join, (7) Entry, and (8) Exit. All of these nodes can contain ControlFlow, which itself is a Link to another node. Dependencies of type Dependency are also represented as Links. There are four subtypes of Dependency that represent all possible dependencies between SCG nodes, namely: (1) AbsoluteWrite_Read, (2) RelativeWrite_Read, (3) AbsoluteWrite_RelativeWrite, and (4) Write_Write.

2. Background and Related Work

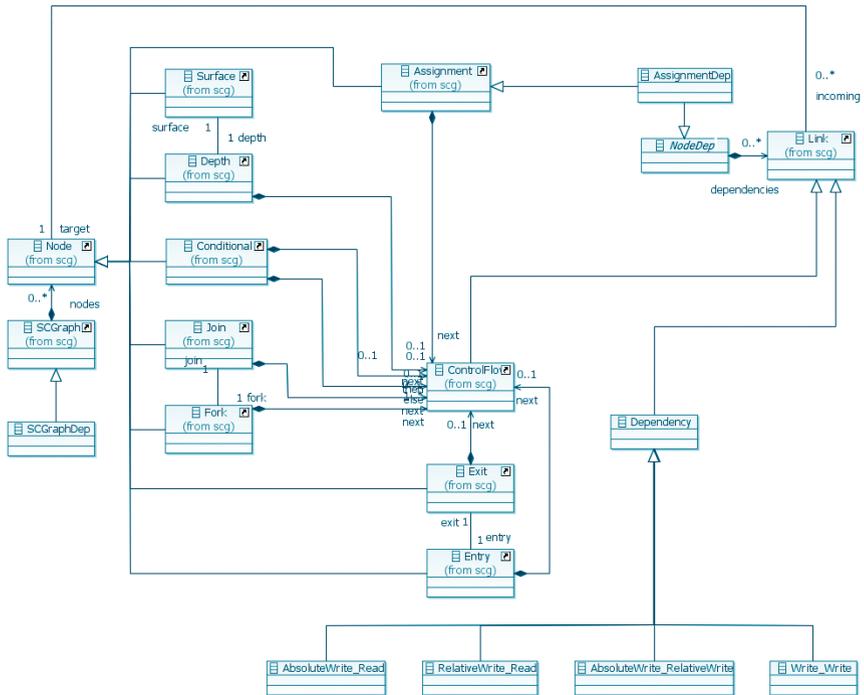


Figure 2.7.3. SCG meta model: Abstract control-flow graph representation including explicit dependencies (adapted from [MSvH14])

2.7.3 SCG Example

Figure 2.7.4 gives an example of a non-concurrent SCG. The control-flow starts at the entry node. An assignment $O = \text{false}$ follows. By entering the surface node, the reaction computation finishes for the current tick. Only in the next tick, the control-flow continues at the corresponding depth node. A conditional node testing for a boolean input A follows. If A is false then the control-flow enters the surface again which finishes the tick computation. If A is true then the control-flow proceeds with the

2.8. Model-Based Compilation and Compiler Infrastructure

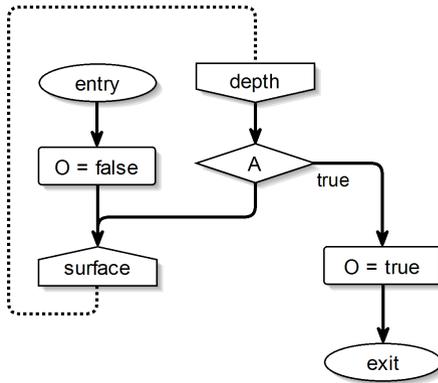


Figure 2.7.4. A simple SCG for the AO example

assignment $O = \text{true}$ and reaches the exit node. This terminates the program. A more comprehensive SCG that includes concurrency is explained later and illustrated in Figure 5.2.4 on page 132.

2.8 Model-Based Compilation and Compiler Infrastructure

Compilation and performing model transformations is closely related. Stefen [Ste97] observed this fact quite early. He proposes *consistency models* to detect inconsistencies between different model descriptions. Furthermore, he relates performing model transformations to giving a semantics to a programming language by translation into an intermediate language. A number of modeling approaches have been developed that base on this observation. To name just a few of them, *Cinco*⁹ can automatically construct code generators from a given meta model [NTI⁺14], Grundy et al. [GHL⁺13] presented *Marama*¹⁰, which provides a set of mostly visual meta tools for language specification and tool building, including synthesis.

⁹<http://cinco.scece.info>

¹⁰<http://wiki.auckland.ac.nz/display/csidst>

2. Background and Related Work

A difference of the aforementioned approaches compared to the interactive incremental compilation approach presented in this theses is that in SLIC the aim is to

- ▶ have rather simple transformation steps,
- ▶ have a sequential pass where each transformation is applied just once,
- ▶ ensure that intermediate models are valid and understandable models,
- ▶ expose intermediate models to the modeler and tool smith,
- ▶ make use of automatically generated graphical views, and
- ▶ use a purely textual description for input, editing, and persistency.

The interactive incremental compilation approach as presented here requires a compiler infrastructure which allows the modeler to control the compiler interactively and which allows the tool smith to generically apply the SLIC approach to a certain modeling language such as SCCharts. It also requires an editor which allows to manipulate an abstract model by concrete syntax changes. Related work in this context dates back to the 80's, e. g., by Gilles Kahn who was involved in both, the Mentor [DGHL80] and the Centaur [BCD⁺87] project. In those days, the Mentor project dealt with providing a structured editor for the Pascal programming language to the programmer. Mentor is a processor to apply manipulations on data structures such as the abstract syntax tree of an editor. These manipulations were, e. g., normalization or documentation transformations on Pascal programs. Hiding the parsing and serialization to the user and also to some extend to the tool smith can these days be realized by using suitable frameworks such as Xtext. These concepts are a bases for *textical modeling* (cf. Section 6.4.4 on page 327). Centaur is a generic and interactive compiler environment, consisting of a data base, a logical engine, and a man-machine interface. The data base provides access to formal objects which exist in standardized representation. The logical engine allows to execute formal specifications. The man-machine interface gives access of the Centaur system functionality to the user. This early fundamental work can be related to the interactive model-based compilation approach presented here, where the formal objects are the abstract models, the logical engine is the model transformation framework and the man-machine interface is the GUI, e. g., for selecting certain model transformations.

The SCCharts Language

SCCharts is a visual Statecharts dialect with a synchronous semantics. More specifically, SCCharts borrow their visual syntax from Charles André's

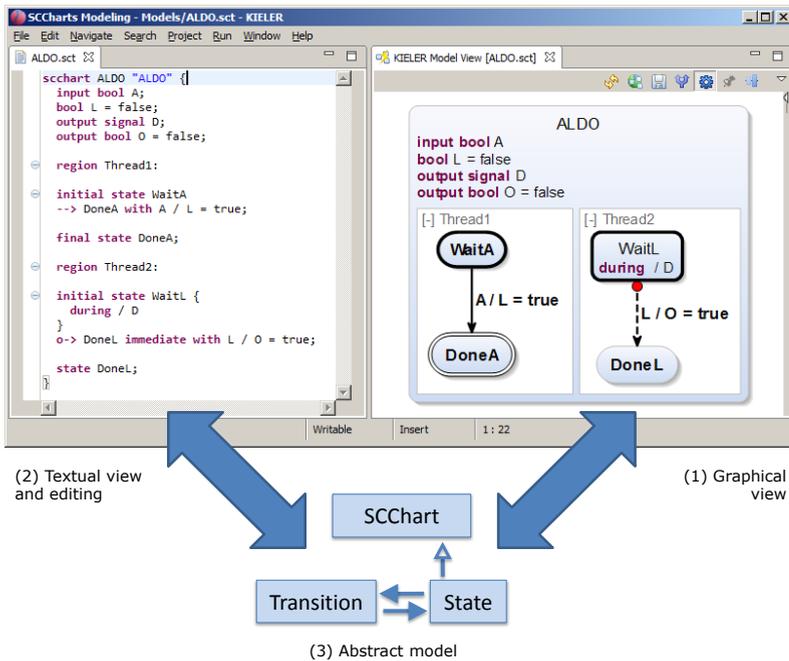


Figure 3.0.1. KIÉLER SCCharts *textical* modeling tool: (1) SCCharts graphical view, (2) SCCharts textual editing, (3) abstract SCCharts model

3. The SCCharts Language

SyncCharts [And96] and have a sequentially constructive synchronous semantics (cf. Section 2.6 on page 36).

This chapter introduces SCCharts using a small example that additionally will be used later as a primary example for compiling SCCharts. Afterwards, it explains the semantic differences to SyncCharts and then discusses all SCCharts visual language features. Next, the textual SCCharts modeling language (cf. (2) in Figure 3.0.1) is introduced. Afterwards, the abstract syntax of SCCharts (cf. (3) in Figure 3.0.1) is given. It is used for representing and transforming SCCharts internally.

3.1 Basic Concepts

Figure 3.1.1 shows ALDO for a brief introduction to SCCharts. The informal specification of ALDO is as follows:

- ▶ The interface has a boolean input variable A , an output signal D , and a boolean output variable O .
- ▶ There is a local boolean variable L for communication purposes between the two concurrent regions Thread1 and Thread2 .
- ▶ Region Thread1 waits for the input A to become true in state WaitA .
- ▶ As soon as A becomes true after the initial tick, the transition to DoneA is taken. This sets the local variable L to true.
- ▶ Concurrently, region Thread2 waits in WaitL for L to become true.
- ▶ While Thread2 is waiting in WaitL , it emits D in each non-initial tick.
- ▶ As soon as L becomes true, the emission of signal D is preempted and the transition to DoneL is taken immediately, i. e., possibly also in the initial tick. This sets the output variable O to true.
- ▶ In DoneL , the program stops reacting but does not terminate.

3.1.1 General Structure

An SCChart has exactly one root state which is a superstate containing internal behavior. ALDO is the root state consisting of a declaration block and

3.1. Basic Concepts

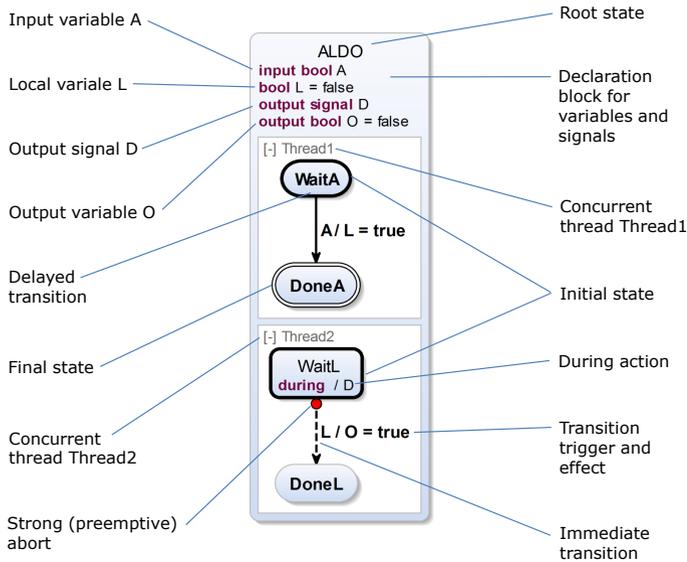


Figure 3.1.1. ALDO SCCharts example

two concurrent regions Thread1 and Thread2. Both concurrent regions have their initial state WaitA and WaitL, respectively, drawn with a bold border. Thread1 has a final state DoneA, drawn with a double border. Thread2 does not have a final state. When entering a superstate with regions, the distinct initial state of each region is entered immediately. As in SyncCharts, there must always exactly be one initial state per region. Hence, when entering ALDO, Thread1 and Thread2 are immediately (in the same tick) entered and so are WaitA and WaitB.

3.1.2 Termination

Entering a final state terminates a region. There may exist an arbitrary number of final states per region. If there is no (reachable) final state in a

3. The SCCharts Language

region, the region and its superstate cannot terminate (if entered). E. g., this is the case for `Thread2`, which has no final state. If all regions of a superstate have terminated then the superstate itself may terminate immediately. If all regions of the root state terminate then the SCChart terminates. Since `Thread2` cannot ever terminate, `ALDO` never terminates.

3.1.3 Synchronous Signals

SCCharts have both, variables of specific type and synchronous, Esterel-style signals. In Esterel or SyncCharts, signals allow to communicate with the environment and signals allow concurrent parts of the Esterel program or SyncChart to communicate. Likewise to Esterel and SyncCharts, (*pure*) signals have a consistent presence status for each tick. This means that the *signal coherence law* holds, i. e., a signal is present in a tick iff it is emitted during the reaction computation for this tick. It is absent otherwise. If a signal is emitted in a tick, it typically cannot be *unemitted* any more for this tick. This is the case, e. g., for Esterel or SyncCharts signals. In contrast, the sequentially constructive MoC allows an explicit *unemit* as proposed for a sequentially constructive extension to Esterel, termed SCEst [RSM⁺15]. A pure signal that is already present may be emitted again in the same tick. This will not change its presence status.

The semantics of (pure) signals for SCCharts is mapped to and implemented by boolean variables. Such a variable is true iff the according signal is present in a tick and false iff the signal is absent. It is therefore possible to test a signal for absence in a transition trigger of an SCChart.

Valued signals carry a typed value in addition to their presence status. The value is kept persistent across ticks and is only changed if the valued signal is emitted. Multiple emissions of the same valued signal in the same tick are only allowed if a combination function exists. This combination function must be associative and commutative, i. e., the order in which the emissions occur does not matter. In SCCharts, a valued signal is mapped to and implemented by a combination of a boolean variable for the presence status and a second variable which is typed to hold the value of the signal.

3.1.4 Declarations

The *declaration* of the interface to the environment of an SCChart is done in its root state. Also, local variables and signals that are not visible to the environment can be declared here. ALDO has two interface variables A and O and one interface signal D. The local variable L is also declared at the top level so that both regions Thread1 and Thread2 are in the scope of this variable, i. e., have access to L. Generally, SCCharts also support *local declarations* within the scope of any region or superstate.

3.1.5 Instantaneous Communication and Preemption

In ALDO, L is used for communicating the fact that Thread1 has taken its transition from WaitA to WaitB. Hence, when L becomes true this is immediately (in the same tick) noticed by Thread2 and leads to a preemptive *strong abort* of WaitL. A strong abort is drawn with a red circle (cf. Figure 3.1.1) and means that any immediate internal behavior of the aborted superstate is preempted. Here, the *during action*, which emits D in every but the initial tick, is preempted such that D is not emitted in the tick when L becomes true. There are no further ticks to consider because the program stops reacting with the tick when L and O become true. Note that because of instantaneous communication, in this example, L and O always become true in the same tick.

3.1.6 Transitions

A transition typically has a trigger and an effect divided by a slash ("/"). An empty trigger is an implicit true-trigger. An effect can be a variable assignment or a signal emission. A transition can only be taken if the trigger evaluates to true and the transition is *active*, i. e., it is checked whether it can be taken or not w. r. t. its trigger. If a transition is taken then the effects are executed in order and immediately in the same tick.

There are two types of transitions visible in ALDO. The transition from WaitA to DoneA is drawn with a solid line indicating a *delayed transition*. It means that in the tick when WaitA is entered, this transition is not active but

3. The SCCharts Language

only in subsequent ticks. Since `WaitA` is an initial state and its parent region is entered in the initial tick, `DoneA` is also entered in the initial tick. Hence, in the initial tick, `WaitA` is entered and the outgoing delayed transition to `DoneA` is not active and hence not checked. It can never be taken in the initial tick even if its trigger `A` would evaluate to true.

In contrast, the transition from `WaitL` to `DoneL` is drawn with a dashed line indicating an *immediate transition*. It means that in the tick when `WaitL` is entered, this transition is already active, i. e., checked and possibly taken. It remains active for subsequent ticks. Since `WaitL` is an initial state, it is entered in the initial tick. Hence, in the initial tick, `WaitL` is entered and the outgoing immediate transition to `DoneL` is active and checked.

Note that because `L` is a local signal and can only become true by taking the delayed transition from `WaitA` to `DoneA`, `L` cannot be true in the initial tick. Since the transition from `WaitL` to `DoneL` waits for `L` to become true, it can never be taken in the initial tick.

3.1.7 Actions

The during action of `WaitL` is non-immediate. This means that it is not executed in the tick when `WaitL` is entered which is only the initial tick. Hence, `D` is not emitted in the initial tick. Note that when `A` is true in the second tick then `D` will never be emitted. This is because it is not emitted in the initial tick and preempted in the second tick.

3.2 Visual Syntax and Semantics

Figure 3.2.1 shows the full set of SCCharts features split into two parts. The region in the upper part aggregates all *Core SCCharts* features while the region in the lower part accumulates all *Extended SCCharts* features. The modeler typically wants to draw from the full set of all core and extended features. This enables them to hide complexity of the model by using adequate SCCharts features. Hiding complexity is a key to readability and maintainability of non-trivial models because a model with less elements is simpler to parse and understand. Readability and maintainability are

3.2. Visual Syntax and Semantics

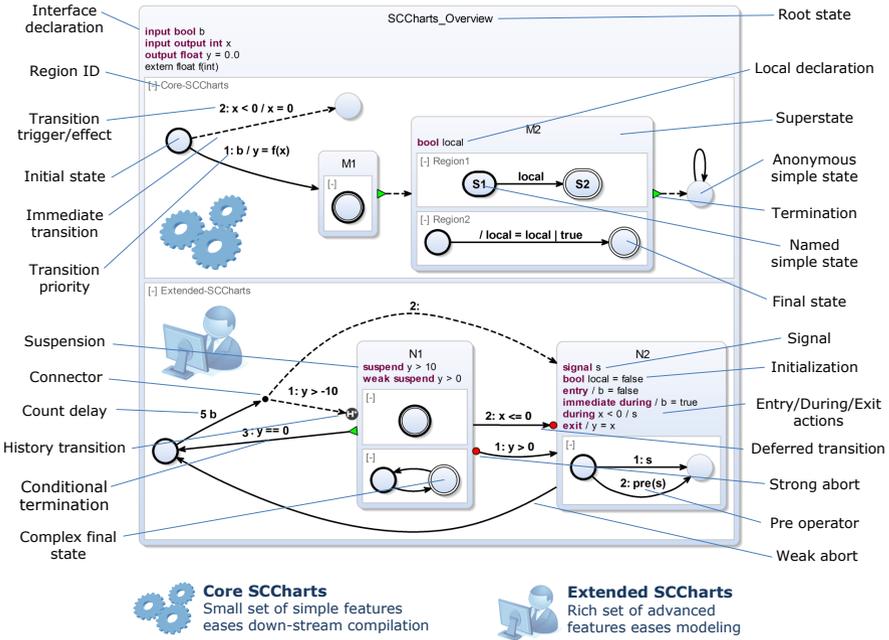


Figure 3.2.1. SCCharts visual syntax overview (based on [MSvHM13])

both essential for safety-critical reactive systems. Each extended feature can be expressed as one or more core features. Hence, all extended features are considered *syntactic sugar*. Using model transformations, the compiler can eliminate all extended features by replacing them with semantically equivalent constructs of core features. This simplifies down-stream compilation significantly since the lower-level parts of the compiler only need to deal with the much smaller set of core features. Additionally, expressing each extended feature based on core or other lower-level extended features gives each extended feature a concrete semantics. Figure 3.2.2 presents how an SCChart using extended features can be expressed by a semantically equivalent SCChart using core features only, picking up the example of ALDO. Note that the choice of core features in principle is not unique but

3. The SCCharts Language

was made in a way that reasonably limits the grow of a model during compilation and for efficiency reasons (e. g., cf. Section 5.6.2 on page 279).

The ALDO Extended SCCharts model of Figure 3.2.2a uses initializations, during actions, and strong aborts, which all are extended features. Figure 3.2.2b shows a semantically equivalent Core SCChart of ALDO where these extended features have all been replaced by core features, while preserving the semantics. Chapter 5 gives details on how arbitrary Extended SCCharts can likewise be compiled to Core SCCharts and further on.

The following sections will discuss all extended and core features as shown in Figure 3.2.1 that have not been used in the ALDO example.

3.2.1 Termination Transitions

Another core feature of SCCharts, shown in the upper region of Figure 3.2.1, are (unconditional) *termination* transitions. These are visually drawn with a green triangle at the start of the transition. The termination transition has the same semantics as the *normal termination* transition of SyncCharts. In short, it joins concurrent control-flow. It has no explicit trigger. The implicit trigger is the termination of the originating superstate. A termination transition is only necessary for superstates, not for normal states. Furthermore, for Core SCCharts, there should only be one termination transition. Finally, termination transitions are always immediate.

Figure 3.2.1 also shows a *conditional termination* in the lower region. This extended feature can be transformed into semantically equivalent core features. A conditional termination may be delayed or immediate. It can only be taken if all regions of the superstate have terminated, the condition holds, and the transition is active.

3.2.2 Transition Priorities

The *transition priority* shown in the upper region of Figure 3.2.1 is a core feature. It is visually drawn as a numerical prefix of the transition label. A transition priority is a tie breaker if more than one transition, originating from the same state, is eligible to be taken.

3.2. Visual Syntax and Semantics

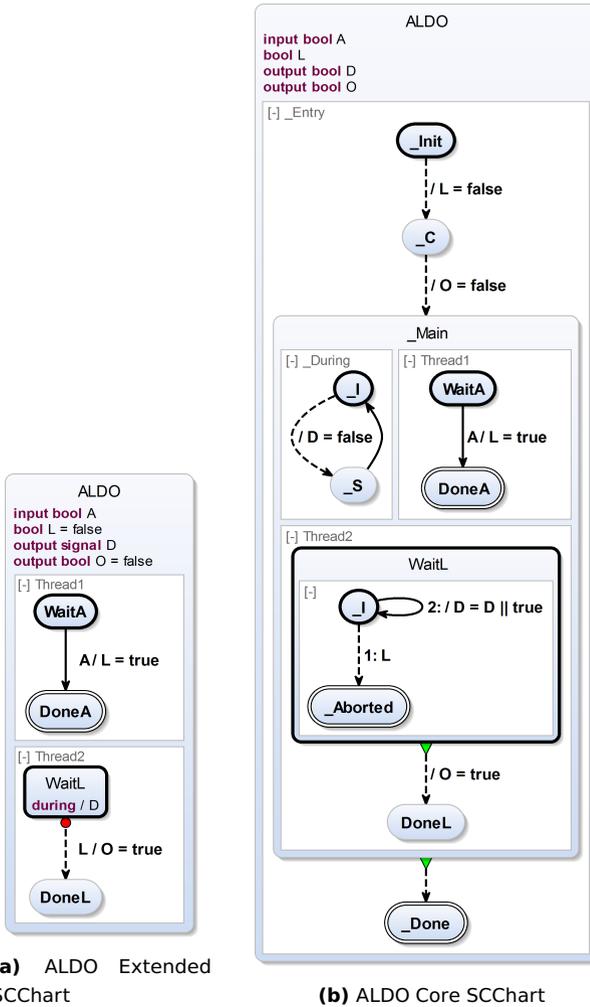


Figure 3.2.2. Expressing the same behavior using extended features or core features only: Extended features help the modeler to hide complexity and enhance readability and maintainability. The usage of core features eases down-stream compilation because fewer features must be handled.

3. The SCCharts Language

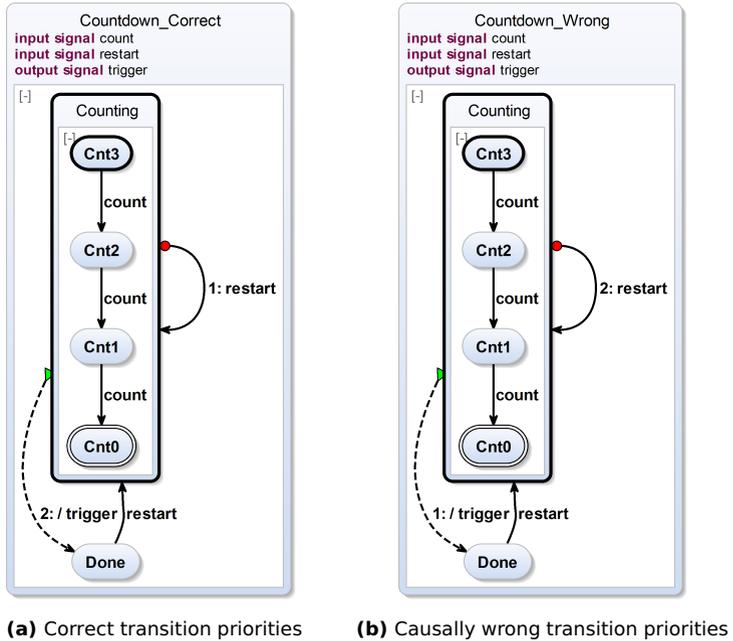


Figure 3.2.3. Transition priorities are used as a tie breaker for multiple outgoing transitions of a state that are active and have a trigger that evaluates to true. This example additionally demonstrates that transition types induce constraints to transition priorities w. r. t. causality. Specifically, strong aborts must have priority over termination transitions.

Hence, a transition priority is only necessary if a state has more than one outgoing transition. It is visually shown only in this case. Furthermore, transition priorities must always be distinct for all transitions originating from the same state. The smaller the transition priority number, the higher the priority. I. e., the transition priority number determines the order in which outgoing (active) transitions and their triggers are tested.

Figure 3.2.3a shows an example of a count down that starts in state Counting and counts down from 3 to 0 at each tick in which the input signal

3.2. Visual Syntax and Semantics

count is present. If the count down reaches the final state Cnt0, it terminates and emits the signal trigger ending up in state Done. The count down system can be restarted at any time by the input signal restart. If the system is in state Cnt1 and count as well as restart are present, then the abort self-loop transition wins over the termination transition because of higher priority.

It is noteworthy that in some cases, such as the one shown in Figure 3.2.3a, transition priorities cannot be swapped. Strong abort transitions always need to have a higher priority than termination transitions. This is exemplified by Figure 3.2.3b where this is not the case and hence a causality problem exists. In this model, the termination has a higher priority than the self-loop strong abort transition. The causality problem here is a little subtle. In short, a termination transition is always triggered by internal behavior, namely the termination of regions of a superstate. Since a strong abort preempts any internal behavior, the internal behavior must never contradict the choice of taking a strong abort.

To discuss the matter in detail, consider the case that the model, shown in Figure 3.2.3b, is in state Cnt1 and count as well as restart are present. Clearly, the termination could be taken because the final state Cnt0 is reached which terminates the region and the termination transition has higher priority than the preemptive strong abort self-loop. However, this is not the only option. Likewise, the abort self-loop transition could be taken because restart is present. This would preempt any inner behavior, it would also prevent the final state Cnt0 from being reached which prevents the termination to be triggered at all. If the termination is not triggered then the next transition considered to be taken is the preemptive strong abort self-loop. In consequence, there would be two possible valid choices but this kind of non-determinism is forbidden for SCCharts. In essence, the choice of transition priorities causes this causality problem as explained before. To prevent this kind of causality problems, models where (conditional) terminations have a higher priority than strong abort transitions should be rejected.

3. The SCCharts Language

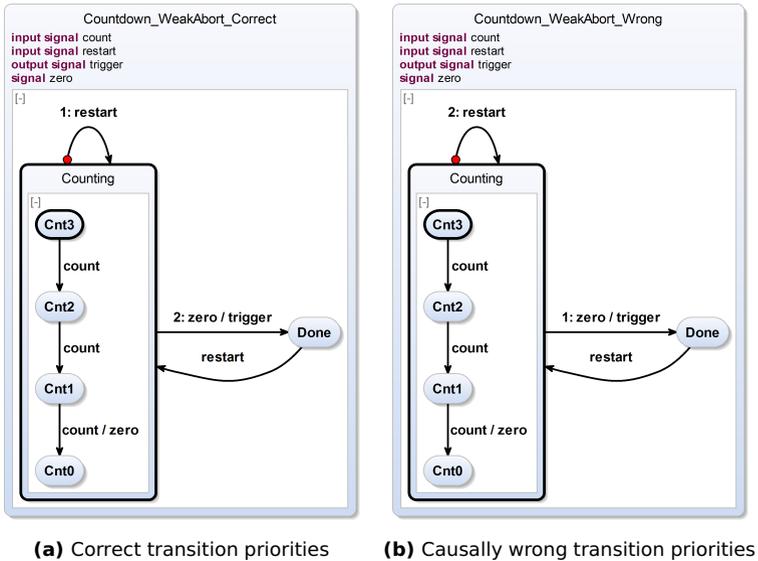


Figure 3.2.4. Weak abort transitions from a superstate may be triggered from the inside of this superstate. However, transition priorities need to be chosen carefully such that no causality problems occur. Specifically, strong aborts must have priority over weak aborts.

3.2.3 Weak Abort Transitions

Another important extended feature are *weak aborts*. In Figure 3.2.1 the state N2 has an outgoing weak abort transition to the initial state in the lower region. In contrast to strong aborts, as presented earlier as part of the ALDO example (cf. Figure 3.1.1 on page 49), weak abort transitions also allow to abort a superstate but do not preempt any internal immediate behavior in the tick when the abort takes place. This is often referred to as allowing a *last will* to the weakly aborted superstate.

Figure 3.2.4 shows very similar models to the ones in Figure 3.2.3. In both models of Figure 3.2.4, not final states and terminations are used to detect reaching the end of the count down, but a local signal zero is emitted

3.2. Visual Syntax and Semantics

when the count down reaches zero. This signal zero triggers the weak abort transition of the superstate Counting to state Done.

Now, consider Figure 3.2.4b. This is a similarly causally incorrect model as Figure 3.2.3b. The reason is likewise the wrong choice of transition priorities. Strong abort transitions need to have a higher priority than weak abort transition priorities. This is the case for the correct model shown in Figure 3.2.4a, but not for the model of Figure 3.2.4b.

Consider again the case that the model (Figure 3.2.4b) is in state Cnt1 and the signals count and restart are present. Clearly, if we take the transition from Cnt1 to Cnt0 then this emits signal zero, which triggers the weak abort transition from state Counting to state Done because it has higher priority than the strong preemptive self-loop triggered by restart. So, choosing the weak abort transition seems to be a valid choice. On the other hand, choosing the strongly preemptive self-loop would preempt again any internal behavior of the state Counting and hence it would preempt the emission of signal zero. If zero is not present then the weak abort cannot be taken and the preemptive strong abort self-loop is the only transition that can be taken and hence a valid choice as well. This is contradictory and a causality problem, where the root of the problem in short is that again the internal behavior must never contradict the choice of taking a strong abort. To prevent this kind of causality problems, models should be rejected, where a weak abort has a higher priority than a strong abort transition.

Weak vs. Strong Abort Transitions

Since weak abort transitions, as used in Figure 3.2.4a from state Counting to state Done, allow a *last will*, it is causally no problem to trigger (here using signal zero) a weak abort from the inside of a superstate.

Note that this transition cannot be modeled as a strong abort like it is done in the causally incorrect model of Figure 3.2.5. The reason for this is the following contradiction: If this strong abort transition is taken because zero was triggered from the inside of the superstate Counting then it would at the same time preempt the instantaneous behavior, i. e., the *last will*, which is unfortunately exactly the emission of zero. However, if the emission of zero is preempted then zero is not present. If zero is not present then this

3. The SCCharts Language

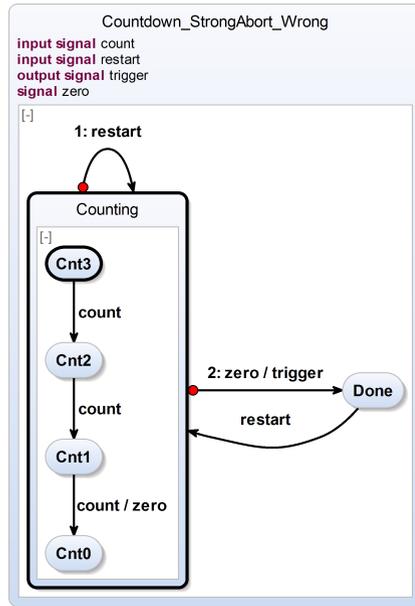


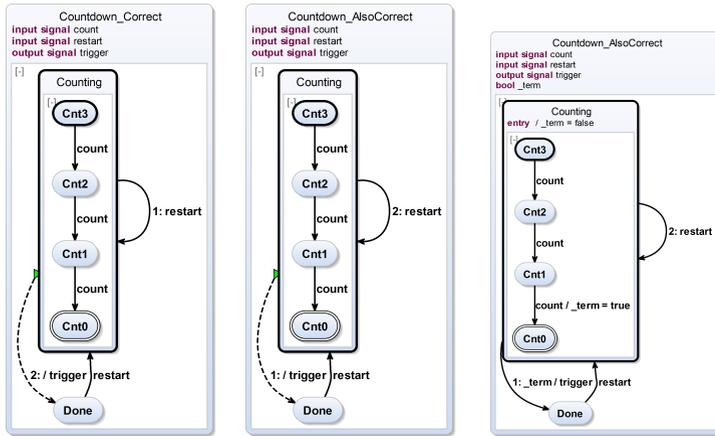
Figure 3.2.5. Causally wrong use of a strong aborts: Count down controller of Figure 3.2.4 where the transition from superstate Counting to state Done is modeled as a strong abort but this must not be triggered from within Counting.

strong abort transition, which is triggered by zero, cannot be taken. Now, if the strong abort is not taken then it will not preempt anything inside the superstate Counting such that zero can be emitted. However, if zero is emitted then again the strong abort should be taken because it is triggered, and so on.

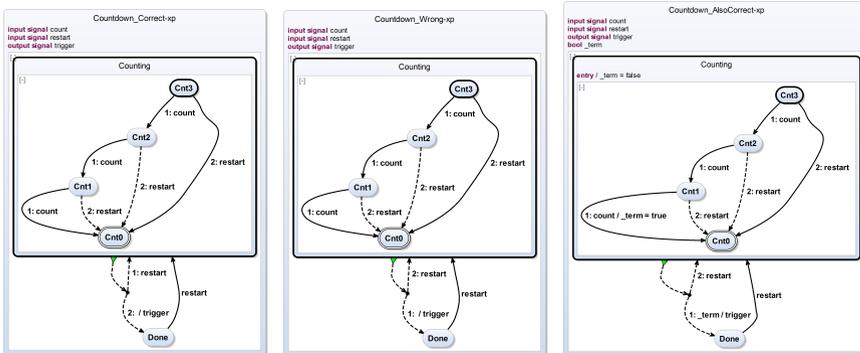
Weak Abort vs. Termination Transitions

André [And03] suggested to give weak abort transitions a higher priority than termination transitions. He also pointed out that a termination transition with a higher priority than a weak abort does create causality problems

3.2. Visual Syntax and Semantics



(a) Correct transition priorities (b) Also correct transition priorities (c) Alternative implementation of (b)



(d) Expanded version of (a) (e) Wrongly expanded version of (b) (f) Correctly expanded version of (b/c)

Figure 3.2.6. Transition types weak abort or termination do not induce any further constraints to transition priorities.

3. The SCCharts Language

similar to the previously described examples.

Figure 3.2.6 illustrates the combination of a weak abort and a termination transition. In Figure 3.2.6a, the weak abort self-loop transition originating in state Counting has a higher priority than the termination transition from state Counting to Done. If in the model state Cnt1 is active and count and restart are true then the weak abort is taken because of its higher priority. However, because of the *last will* any possible internal actions of Counting are still performed.

Figure 3.2.6b illustrates the same model but with the priorities swapped for the weak abort and the termination transition. Interestingly, this model is still causally correct. If in the model again state Cnt1 is active and both count and restart are true then the termination is always taken because of its higher priority.

The expanded versions of both models (cf. Figure 3.2.6a and Figure 3.2.6b) are shown in Figure 3.2.6d and Figure 3.2.6f, respectively. This clarifies the semantics for SCCharts.

Technically, the originally proposed abort transformation [vHDM⁺13c] could not correctly handle terminations with a higher priority than weak aborts as illustrated in Figure 3.2.6e. The core of the problem was “information loss” whether the resulting termination transition was triggered by a original weak abort or by the original termination of the inner region itself. If this information is made explicit beforehand (cf. the intermediate model in Figure 3.2.6c), e. g., by a dedicated `_term` auxiliary variable, then the abort transformation would be able to produce the desired result (cf. Figure 3.2.6f). Such a preprocessing can be done automatically and is implemented in the current abort transformation implementation, see Section 5.2.8 on page 146.

For SCCharts we do not enforce that weak aborts must have a higher priority than termination transitions but only that strong abort transitions have a higher priority than both, termination and weak abort transitions.

3.2.4 Entry Actions

In Figure 3.2.1 on page 53, the state N2 specifies an *entry action* that sets `b` to false when the state is entered or re-entered by any transition. Likewise to during actions, entry actions are part of the state declaration. Entry actions

3.2. Visual Syntax and Semantics

are an extended feature. Entry actions are always executed sequentially before during actions, before exit actions, and before possibly internal behavior of a superstate. Entry actions are sequentially executed after the transition that enters the state. Multiple entry actions themselves are sequentially ordered. Additionally, entry actions are always immediate. They can have an optional trigger (condition).

3.2.5 Exit Actions

Exit actions also are an extended feature. In Figure 3.2.1 on page 53, the state N2 specifies an *exit action* that sets y to x with the implicit true trigger. Exit actions can have an optional additional trigger. They are executed if a state is left by a transition. Multiple exit actions are executed sequentially ordered and before the leaving transition which has triggered the exit action.

3.2.6 During Actions

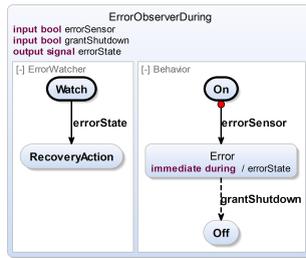
The ALDO example in Figure 3.1.1 on page 49 already contained a *during action* extended feature. During actions are part of the state declaration and are possibly executed whenever the superstate they are declared in is active. During actions may have an optional trigger and may be immediate or delayed. For example in Figure 3.2.1 on page 53, the state N2 specifies a delayed conditional during action, which is triggered by $x < 0$. It emits a signal s in each tick that N2 is active but not entered whenever x is negative in this tick.

During actions are often used in the following two scenarios:

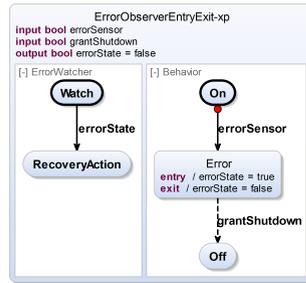
1. A local signal should be kept present while the system is in a certain state. Other parts of the system concurrently react to the presence status of this signal.
2. An externally connected component expects an output or input/output interface variable or signal to stutter in its value actively while the system is in a certain state.

In both scenarios it is essential not only to set a flag when entering a state and reset the flag when leaving the state.

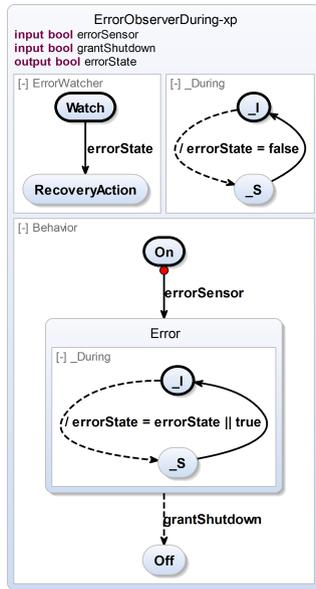
3. The SCCharts Language



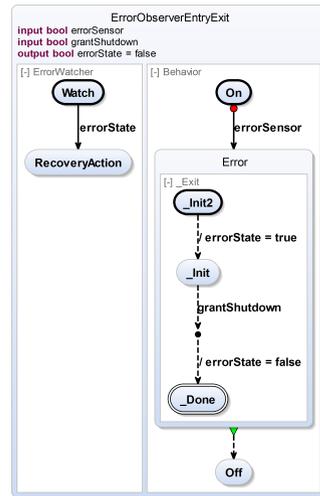
(a) Using during actions



(b) Using entry and exit actions



(c) Expanded version of (a)



(d) Expanded version of (b)

Figure 3.2.7. Visibility pitfall when replacing during actions as in (a) with a combination of entry and exit actions as in (b): When tracking active states, transient states may be neither visible externally nor internally. Note that On represents a collapsed superstate with a strong abort.

3.2. Visual Syntax and Semantics

At first glance this seems to be an obvious and a valid alternative but this is not generally the case. The alternative to use entry and exit actions in place of during actions has two drawbacks:

1. If entering and exiting a state is done instantaneously, i. e., in the same tick, then it may not be possible to observe externally that the state was active in the same tick.

Additionally, when also observing this situation internally (in a concurrent part of the SCChart) then this might foster inconsistencies as explained in the following example.

2. If the variable can also be reset in other parts, this leads to a fragile design. Using during actions may have more computation costs but often leads to a more robust design when a variable is actively given a certain value in each tick.

Both is illustrated in Figure 3.2.7, where an error observer uses an immediate during action within state Error emitting signal errorState to track whether the state Error is active in a tick. If the state Error becomes active in a tick then the (internal) concurrent ErrorWatcher notices this and goes to the RecoveryAction state. After an error, the system typically shuts down but may wait for other subsystems using a grantShutdown variable.

The critical behavior occurs when the grantShutdown variable is true when entering state Error. In this case the state Error is entered and left in the same tick, hence Error is a *transient state*. However, because Error is left by a weak abort, the during action (Figure 3.2.7a) still emits the signal errorState.

The system can possibly also be modeled using entry and exit actions to track whether the state Error is active. This is shown in Figure 3.2.7b. When entering state Error, the boolean variable (not a signal any more) errorState is set to true and when leaving state Error, the variable is reset to false. At first glance the behavior seems to be equivalent to the one of the model in Figure 3.2.7a — but it is not. This is exposed in the expanded versions in Figure 3.2.7c and Figure 3.2.7d where during, entry, and exit actions have been eliminated by model transformations.

If state Error is transient and left by a weak abort, using a during action and signals makes it possible to observe that the state Error was active during

3. The SCCharts Language

the tick even if it is left instantaneously as in the case if `grantShutdown` is true. Figure 3.2.7c clearly sets `errorState` to true (using a relative-write after it has been initialized in the concurrent region absolutely with false, see the discussion of signal expansion in Section 5.2.11 on page 190) before the weak abort immediate transition from `Error` to `Off` triggered by `grantShutdown` is taken immediately. The concurrent region will see `errorState == true` in this tick and can start the recovery action. Additionally, `errorState` is an output variable and can be tracked externally, i. e., outside of the SCChart. Figure 3.2.7d exposes the difference when entry and exit actions were used to track if state `Error` is active. In summary, the difference is that in the same tick when `grantShutdown` is true and `Error` is or became active, the variable `errorState` will be false instead of true. This is because of sequentiality and because both absolute writes are executed before the concurrent region `ErrorWatcher` may observe (read) the value of `errorState`. As a consequence, if `Error` is entered and left in the same tick, the change of `errorState` cannot be observed concurrently and is also not visible externally. Hence, region `ErrorWatcher` will remain in state `Watch` and not go into state `RecoveryAction` even if an error had occurred.

3.2.7 Initializations

Contrary to `SyncCharts`, in (Core) `SCCharts` the primary elements for communications are not signals but variables. Signals in `SCCharts` are an extended feature that is eliminated during compilation and which is also based on variables. The value of input (and input output) variables must be set by the environment for each synchronous tick. However, local or output variables keep their values until the value is changed by an executed variable assignment as part of a local action or a transition. The initial value of local or output variables shall not be *undefined* as this may be source of non-determinism which must be avoided. In order to properly initialize a variable at the beginning, i. e., right after its declaration, an explicit *initialization* feature can be used.

In Figure 3.2.1 on page 53 the state `N2` contains an initialization of variable `local` that is initialized with the value `false`. Due to the local variable, the initialization will only take place if a transition to `N2` is taken. Initializations

3.2. Visual Syntax and Semantics

are transformed to *entry actions*. Note that regarding robustness, the tooling shall check for uninitialized variables. Additionally, an SCCharts compiler shall offer *default initializations* w. r. t. the concrete variable type, e. g., when the user uses a special *pragma* to turn on/off such a feature. Default initializations could be part of the initialization transformation. However, unfortunately this is not yet done by the current version of the KIELER SCCharts compiler. It is planned for future versions.

3.2.8 Connectors

A *connector* state is used to split transitions into several segments to allow re-use of common trigger/effect parts according to the *Write Things Once* (WTO) principle [Ber00b]. Connectors are sometimes also referred to as *conditional nodes*. Connectors are an extended feature and a connector is shown in the lower region of Figure 3.2.1 on page 53 as a target of the transition originating from the initial state. By convention, all outgoing transitions from a connector are implicitly immediate, which in Figure 3.2.1 on page 53 is indicated visually by dashed transition lines. “Implicitly” means that one does not model the outgoing transitions explicitly as being immediate using the `immediate` keyword but they are still immediate just because they originate in a connector. Another convention makes all connector states transient: A *default* outgoing transition from a connector that only has the implicit true-trigger but no additional trigger is always required. This ensures that control can never rest inside a connector state which is the definition of a *transient state*. Observe that the connector and the three attached transition segments in Figure 3.2.1 on page 53 could have been replaced by two transitions and no connector. In this case one transition from the initial state to state N1 would have the trigger `5 b` and `y > -10` and the other transition from the initial state to state N2 would have the trigger `5 b`. As can be seen, the trigger part `5 b` would need to be duplicated which violates the WTO principle. Note that both alternative transitions would have to be delayed because the transition from the initial state to the connector state is delayed.

As this discussion makes clear, the connector concept is rather trivial, as one might as well just use an ordinary state instead of a connector. However,

3. The SCCharts Language

we still find the connector concept useful, as it makes transient states immediately obvious and diagrams a bit more lightweight and readable.

3.2.9 Suspensions

An SCChart state can be *suspended* by a condition. In Figure 3.2.1 on page 53 the state N1 specifies the suspension condition $y > 10$ in the lower region as an extended feature. Internal control is frozen for a suspended state. Therefore, no transitions are taken, no local or output variables or signals are written and so on for any tick where the suspension condition holds. A state cannot suspend itself meaning the condition must not have a dependency to the internal behavior of a superstate.

SCCharts also offers *weak suspension* as known from Quartz [Sch09b] or Esterel v7 [Est05]. In Figure 3.2.1 on page 53, the state N1 also specifies a weak suspension condition $y > 0$. This is also an extended feature that can be transformed into core features. A weak suspension is similar to a suspension with one important difference: It allows the weakly suspended state to express its *last will* whenever the weak suspension trigger holds. Hence, any immediate behavior is allowed and will be executed. The interesting point is that control is still frozen and the weakly suspended state remains in the exact same internal state(s) as if it would have been suspended strongly (not weakly). In subsequent ticks, immediate behavior is again permitted and control keeps being frozen until, eventually, the weak suspension condition may evaluate to false.

3.2.10 Count Delays

Another useful extended feature are *count delays*. In the lower region of Figure 3.2.1 on page 53 a count delay, namely 5 b, is used for the transition from the initial state to the connector state. A count delay is expressed by a number that is preceding a trigger of a transition. This number n declares that the transition can only be taken when the trigger of the transition evaluates to true the n th time since the state is active. Count delays help to reduce complexity of a model that would otherwise be increased by explicit counting.

3.2. Visual Syntax and Semantics

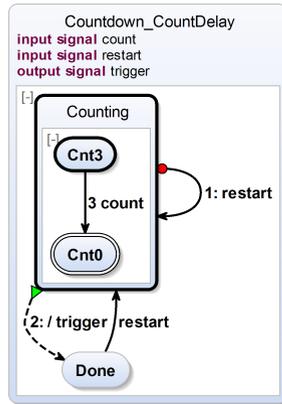


Figure 3.2.8. Example use-case of count delay Extended SCCharts feature: The count down controller of Figure 3.2.3a on page 56 simplified by using a count delay transition.

Figure 3.2.8 demonstrates a use-case where a count delay simplifies the original count down controller model (cf. Figure 3.2.3a on page 56) by hiding the complexity of an explicit counting using explicit states. The original model had four states in order to count to zero. The model in Figure 3.2.8 makes do with only two (visible) states. Internally, the counter for the count delay transition still increases the state space. However, as this is only implicit, the count delay feature, as the other extended features, helps to increase readability and maintainability.

3.2.11 The Pre Operator

The `pre()` operator is defined for signals and variables and `pre()` gives access to the presence status/value of the previous tick. `pre()` can be nested to access far earlier tick instances. `pre()` is an extended feature that can be transformed into core features. Figure 3.2.1 on page 53 demonstrates the use of `pre()` in the inner region of state N2. One transition from the initial state checks for `s` being present, the other checks for `pre(s)` being present, i. e., it checks whether `s` was present in the last tick.

3. The SCCharts Language

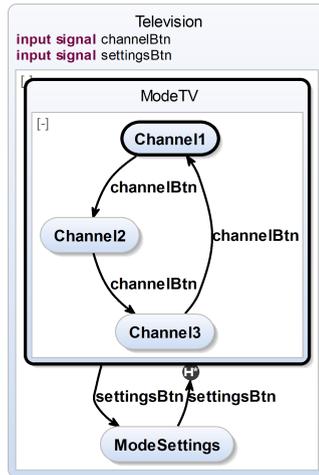


Figure 3.2.9. Example use-case of history transition Extended SCCharts feature: A TV with three channels *remembers* the selected channel when coming back from the settings mode to the TV mode.

3.2.12 History Transitions

Typically, if a superstate is entered, all of its regions start in their initial states. If these initial states are superstates again, the regions of these superstates also start in their initial states and so on.

In contrast, *history transitions* are useful if a superstate that is left should possibly be re-entered later and should restart in its unchanged last internal substates. This is a common behavior used in many control applications. Harel introduced the concept of history transitions in its original Statecharts [Har87] proposal.

Figure 3.2.9 shows an example model using the history feature. It models the control of a 3-channel-TV that has two modes, a **ModeTV** mode for watching TV and a **ModeSettings** mode to configure settings such as language or sound. Initially, the TV is in the **ModeTV** state starting with the first channel **Channel1**. Using the `channelBtn` input, the user may switch through the three channels. When pressing `settingsBtn`, the controller switches to the

3.2. Visual Syntax and Semantics

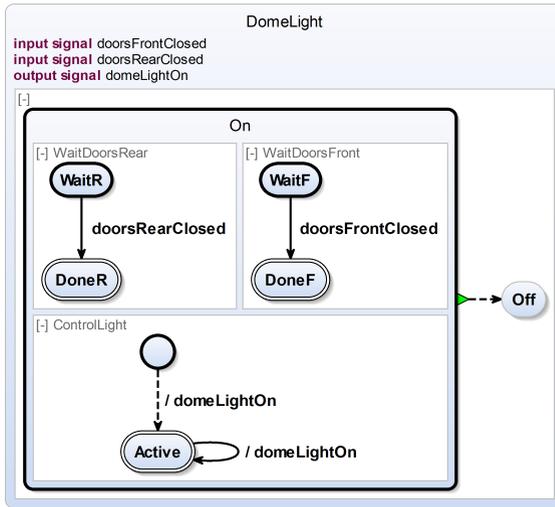


Figure 3.2.10. Example use-case of complex final state Extended SCCharts feature: A dome light controller of a car waits for all front and rear doors to be closed and emits a signal domeLightOn while waiting in order to keep the dome light of the car on. If all doors are closed, it terminates and does not emit the signal any more.

ModeSettings state. Internally, this model will save the internal state of the superstate ModeTV in order to be able to remember the selected channel at a later time. This is needed for the history transition from ModeSettings to ModeTV. If this history transition is taken then the superstate ModeTV will not be in its initial state Channel1, but in the last internal substate it was in when the ModeTV state was left. This can be any of the three internal states Channel1, Channel2, or Channel3.

History transitions should be used only when explicitly needed as they increase the overall state space.

3. The SCCharts Language

3.2.13 Complex Final States

Typically, a final state marks the end of a concurrent thread. This means that control ends there. For Core SCCharts it is therefore not allowed to model outgoing transitions from a final state. Furthermore, it is not allowed for a final state to have any internal behavior. Thus, a superstate cannot be a final state and a final state cannot have during or exit actions in Core SCCharts.

As Figure 3.2.10 demonstrates, this could still be desirable in some situations. The dome light controller modeled here emits a signal `domeLightOn` that will keep the dome light of a car on as long as at least one door is still open. If front and rear doors are both closed, it turns itself off, not emitting the signal `domeLightOn` any more which turns the light off.

The states `DoneF` and `DoneR` are ordinary final states with no outgoing transitions. Hence, the regions `WaitDoorsFront` and `WaitDoorsRear` only use Core SCCharts features. In contrast, the state `Active` in region `ControllLight` is a final state that has a self-loop in order to emit the signal `domeLightOn` in each tick. However, still this region should claim it potentially is done — if the other regions are. This is done by using a *complex final state*.

A complex final state is a final state that is allowed to have outgoing transitions or even internal behavior. The semantics for a complex final state is that a superstate terminates if all of its regions are in either final states or complex final states. Note that this is a *shallow* definition as a complex final state may also contain some other final states in its regions that are not considered for termination of the outer superstate. Complex final states can be left even in the same tick in which they have been entered. The semantics for termination of the superstate is that termination has the lowest priority. Hence, an entered-and-left complex final state may prevent the termination of its superstate.

3.2.14 Static Variables

Variables declared in a superstate only live in the scope of this state. When the superstate is re-entered the variables will be re- initialized. This may not always be desired as shown in the example of Figure 3.2.11. In this

3.2. Visual Syntax and Semantics

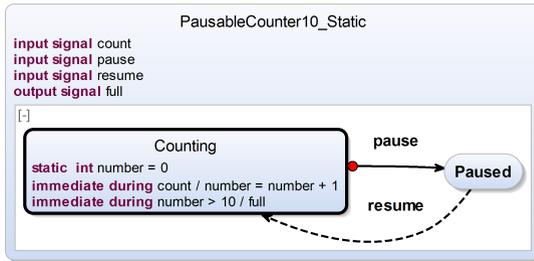


Figure 3.2.11. Example use-case for static variables: A counter that can be paused and which can count to 10 and then emit a signal full.

example, a counter is able to count to 10 and then emits a signal full. It counts up an internal integer variable `number` whenever it is triggered by the input `count`. The model has an initial state `Counting` and another state `Paused`. The counter can be paused any time when the input `pause` is set. It then transitions to the `Paused` state. In `Paused`, no counting occurs but when the input `resume` is set then the counter transitions to state `Counting` again. When re-entering `Counting`, typically, any declared local variables would be re-initialized. However, `number` is declared to be *static*. This prevents any re-initialization and the counter can truly resume.

3.2.15 Deferred Transitions

Deferred transitions are an extended feature of SCCharts and can be seen as the dual to abort transitions. Where abort transitions change state and preempt instantaneous behavior, i. e., the *last will* of the source state, deferred transitions in contrast preempt instantaneous behavior, i. e., the *first will*, of the target state. Deferred transitions can also help to break dependency cycles because they enforce a tick boundary when transitioning to the target state. The target state is prevented to react or be left instantaneously. The concept of deferred transitions is adapted from SCADE, see discussion in Section 2.4.1 on page 30.

Figure 3.2.12 demonstrates a use-case for deferred transitions that could be the preemption of entry actions, which are instantaneous behavior. The

3. The SCCharts Language

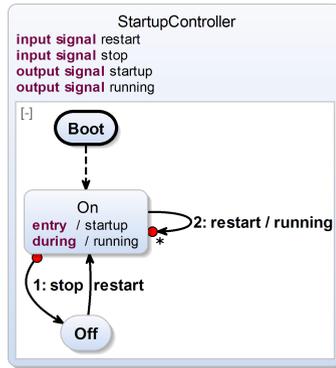


Figure 3.2.12. Example use-case of deferred transition Extended SCCharts feature: A startup controller that emits signal running if the controller is On. It emits signal startup if it is booted initially or restarted after it was switched Off previously.

depicted StartupController emits the signal startup when booting. In the following ticks it will be in state On emitting the signal running. If it is switched Off by the input signal stop, it will not emit running any more. Later, in the first tick when it is restarted, it will emit startup again and in the following ticks it will emit running being in state On. The controller can also be restarted when it is already On and running. In this case, a deferred transition will prevent the controller from emitting the startup signal again, which is modeled as an entry action.

The example shows that it sometimes may be desirable to prevent instantaneous behavior when entering a target state. Using deferred transitions, a state can be entered “hidden” such that immediate transitions behave the same way as non-immediate transitions (only in consecutive ticks) and entry actions are skipped.

This behavior can be achieved manually but it will affect readability and maintainability. Hence, deferred transitions should be used when this behavior is desirable.

There is a shallow (default) and a deep variant of deferred. The deep variant is denoted with an additional asterisk in the SCChart diagram. It

preempts all instantaneous behavior in all hierarchy layers of the entered superstate. The shallow variant without the asterisk only preempts the outgoing immediate transitions of the entered state but allows any internal immediate behavior.

3.3 The Textual Syntax: SCT

The previous section introduced all core and extended features of SCCharts and their visual representation. The current Eclipse¹-based KIELER implementation, as shown in Figure 3.0.1 on page 47, includes a textual model editor and a compiler for SCCharts. The model editor is a text editor with an automatically updated graphical view that always shows the current visual representation of the model opened in the textual editor.

For specifying SCCharts textually in a model editor, the SCCharts Textual Language (SCT) is used. This language is based on previous works by Schneider [Sch11] and Bayramoglu [Bay09].

The ALDO example SCChart was introduced in Section 3.1 on page 48. Its textual description using the SCT language is given in Listing 3.3.1. The SCChart is named and labeled in the first line. Lines 2 to 5 define the declaration of the signals and variables used. Line 7 defines the first region Thread1 that contains two states WaitA and DoneA. The initial state WaitA is defined in lines 8 to 9. The transition from state WaitA to DoneA belongs to the source state WaitA and hence is defined in line 9 together with the trigger A and the effect L = true. The final state DoneA is defined in line 10. The second region Thread2 is defined in line 12. It has also two states WaitL and DoneL. The initial state WaitL is a superstate defined in lines 13 to 16. It contains a during action that emits D in each non-initial tick. Furthermore, it has an outgoing strong abort immediate transition to state DoneL as defined in line 16. This transition has the trigger L and the effect O = true. The state DoneL is defined in line 17.

Listing 3.3.2 shows the overview of all SCCharts visual language elements from Figure 3.2.1 on page 53 in their textual SCT syntax.

¹<http://eclipse.org>

3. The SCCharts Language

```
1 scchart ALDO {
2   input bool A;
3   bool L = false;
4   output signal D;
5   output bool O = false;
6
7   region Thread1:
8     initial state WaitA
9     --> DoneA with A / L = true;
10    final state DoneA;
11
12   region Thread2:
13     initial state WaitL {
14       during / D
15     }
16     o-> DoneL immediate with L / O = true;
17     state DoneL;
18 }
```

Listing 3.3.1. ALDO example as SCCharts Textual Language (SCT)

3.3.1 Grammar

Listing 3.3.3 shows an excerpt of the SCT grammar based on the Xtext Eclipse framework². The excerpt shows how concrete SCT syntax for regions (line 57), states (line 65), and transitions (line 87) are defined and linked to the SCCharts abstract syntax introduced in Section 3.4 on page 81. For example, a transition can be annotated (line 88), it has a specific transition type (line 89), and specifies a target state (line 90). Optional keywords such as immediate, deferred, or history (lines 91 to 93) can be used to set appropriate flags in a model. After a with keyword (line 95), an optional delay can be specified followed by a boolean expression which serves as the trigger of the transition. The effects list follows after the “/” character (line 96). It can be an empty list.

3.3.2 Identifier Name vs. Label

States and regions have a (unique) name as an identifier. Additionally, they may have an optional label (quotation marks) that not needs to be unique.

²<http://eclipse.org/Xtext>

3.3. The Textual Syntax: SCT

```
1 scchart SCCharts_Overview {
2   input bool b;
3   input output int x;
4   output float y = 0.0;
5   //'extern float f(int)';
6
7
8   region "Core—SCCharts":
9
10  initial state A ""
11  --> M1 with b / y = 'f(x)'
12  --> B immediate
13     with x < 0 / x = 0;
14
15  state B "";
16
17  state M1 {
18
19     initial final state C "";
20  }
21  >--> M2;
22
23  state M2 {
24     bool local;
25
26     region "Region2":
27
28     initial state C ""
29     --> D with / local = local | true;
30
31     final state D "";
32
33     region "Region1":
34
35     initial state S1
36     --> S2 with local;
37
38     final state S2;
39  }
40  >--> E;
41
42  state E ""
43  --> E;
44
45  region "Extended—SCCharts":
46
47  initial state F ""
48  --> G with 5 b;
49
50  connector state G ""
51  --> N1 history with y > -10
52  --> N2 immediate;
53
54  state N1 {
55     suspend y > 10;
56     weak suspend y > 0;
57
58     region:
59
60     initial state H ""
61     --> I;
62
63     final state I ""
64     --> H;
65
66     region:
67
68     initial final state J "";
69     }
70     o--> N2 with y > 0
71     --> N2 deferred with x <= 0
72     >--> F with y == 0;
73
74  state N2 {
75     signal s;
76     bool local = false;
77     entry / b = false;
78     immediate during / b = true;
79     during x < 0 / s;
80     exit / y = x;
81
82     initial state K ""
83     --> L with s
84     --> L with pre(s);
85
86     state L "";
```

Listing 3.3.2. SCCharts feature overview visualized in Figure 3.2.1 as its textual SCT equivalent

E. g., state C defined in line 18 of Listing 3.3.2 has the identifier name C but an empty label. If no label is specified, the name is the implicit label.

3. The SCCharts Language

```
57 Region returns sccharts::Region:
58   {sccharts::Region}
59   (annotations += Annotation)*
60   'region' (id=ID)? (label=STRING)? ':'
61   (declarations+=Declaration)*
62   (states+=State)+
63 ;
64
65 State returns sccharts::State:
66   (annotations += Annotation)*
67   (
68     ((initial?='initial') (final?='final'))?
69     |
70     ((final?='final') (initial?='initial'))?
71   )?
72   (type=StateType)? ('state' (id=ID) (label=STRING)?
73   (
74     ('references' referencedScope = [sccharts::State|ID]
75     ('bind' bindings += Binding (' bindings += Binding)*))
76   )
77   |
78   ('{'
79     (declarations+=Declaration | localActions+=LocalAction)*
80     ( (regions+=SingleRegion|
81       regions+=SingleRegion)? (regions+=Region)* )
82   '}')
83   )?
84   (outgoingTransitions+=Transition)* ';'
85 ;
86
87 Transition returns sccharts::Transition:
88   (annotations += Annotation)*
89   (type=TransitionType)
90   targetState=[sccharts::State|ID]
91   (immediate?='immediate')?
92   (deferred?='deferred')?
93   (history=HistoryType)?
94   (label=STRING)?
95   ('with' ((delay=INT)? trigger=BoolExpression)?
96   ('/' effects+=Effect (' effects+=Effect)*)?
97   )?
98 ;
```

Listing 3.3.3. SCT Xtext based grammar excerpt

3.3.3 Priorities

Priorities are not explicitly defined in the textual SCT definition. However, the textual order of outgoing transitions for a state defines the priorities

implicitly. E. g., in lines 68 to 70 of Listing 3.3.2, three outgoing transitions originating from superstate N1 are defined. The strong abort transition “o->” has the highest priority because it is defined first, whereas the termination transition “>->” has the lowest priority since it is defined last.

3.3.4 Host Code

Line 5 of Listing 3.3.2 demonstrates the inclusion of host code in the declaration of the SCChart root state. The host code must be in the language that this SCChart is ultimately transformed to before a machine code compiler is used. Typically, this is C or Java, but in principle it can be any other programming language. Line 11 shows how to use host code in an transition effect or in a transition trigger. The declared host code function $f()$ is used here for assigning the SCCharts variable y to the return value of calling $f()$ with the parameter x , which is another SCChart variable.

Host Code Usage: The use of host code within the model should be minimized. The reason is that host code ties down the target language and the model cannot be translated to some other target language any more. For example, if developing an SCChart for the Java-based NXT Lego Mindstorms including Java host code, a C-based simulator cannot be used for this SCChart and will fail to compile the Java host code parts. Furthermore, dependencies may become obfuscated and obscured so that the SCCharts compiler cannot handle them correctly when scheduling concurrent parts. Any kind of model checking on an SCChart might be affected likewise. Finally, if a host code call is executed within the tick function call, e. g., when taking a transition, the timing of the tick function gets less predictable in terms of jitter. Therefore, it is good practice to use only variables as a primary interface and use host code only for the wrapping code that the generated tick function is integrated in. This way, the tick function can be simulated and translated into any target language and the wrapping code can be adjusted for the desired use-case. Figure 3.3.1 illustrates how a host code call meshed up inside the model (cf. Figure 3.3.1a) can be avoided by simply introducing an additional interface variable `motorOn` (cf. Figure 3.3.1b). Since such a clean separation

3. The SCCharts Language

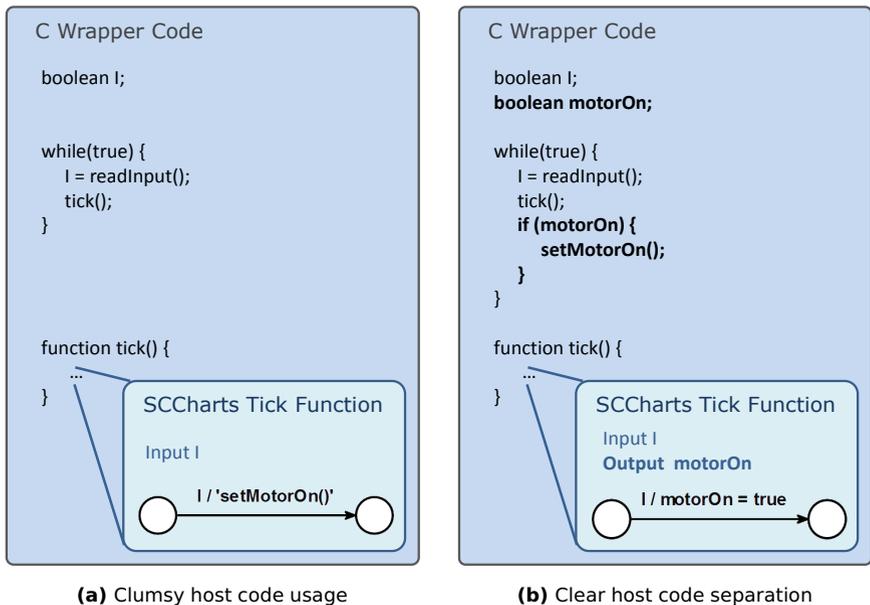


Figure 3.3.1. Host code usage suggestion example: Two very similar models and their wrapper code: The left model uses host code inside the model which leads to non-portable code as well as timing and dependency difficulties. The right model uses an additional interface variable to avoid these kind of problems.

might increase the number of interface variables, this might not be beneficial and necessary for temporal debugging code.

3.3.5 Other Elements

Termination transitions are specified using *symbolic notation* “>->” as seen in line 20 of Listing 3.3.2. Similarly, for strong abort transitions “o->” is used (line 68). Weak abort transitions are the default also for transitions originating from simple states, i. e., from non-superstates, and are specified

using “-->” (line 69). Local variables like local within state M2 in line 23 can be defined at the beginning of the superstate’s declaration part. Similarly, local signals can be declared, e.g., signal s in line 73. Entry and exit actions are defined similarly to during actions as shown in lines 75 and 78. Local variables can be initialized when entering the declaring state, e.g., in line 74. The `pre()` operator can be used as-is inside expressions as shown in line 82. Connector states additionally carry the connector keyword in front of state as shown in line 48. Suspend and weak suspend are defined as in lines 53 and 54, respectively. A history transition is specified using the history keyword for the transition as in line 49. Deferred transitions are marked with the deferred keyword (line 69).

3.4 Abstract Syntax

Besides the concrete visual syntax shown in Figure 3.0.1 on page 47 which is borrowed from SyncCharts and the textual SCT representation for modeling SCCharts, internally a meta model exists which defines the abstract syntax of SCCharts. The abstract syntax describes the *language* of SCCharts models.

The SCCharts meta model evolved over time. It is largely based on the SyncCharts meta model and its KIELER implementation [Sch09a]. Like the visual concrete syntax of SCCharts w. r. t. its color scheme, also the SCCharts meta model was influenced by the first SCCharts editor prototype. This first editor prototype [Har13] was based on the Yakindu statecharts tools³ from itemis⁴, see Section 6.4 on page 322.

Figure 3.4.1 shows the meta model in its current KIELER implementation. Note that this is a simplified version that does not contain all details to facilitate readability. However, it contains all major classes that are needed to get an idea of how the abstract syntax of SCCharts is structured.

³<http://www.yakindu.de>

⁴<http://www.itemis.org>

3. The SCCharts Language

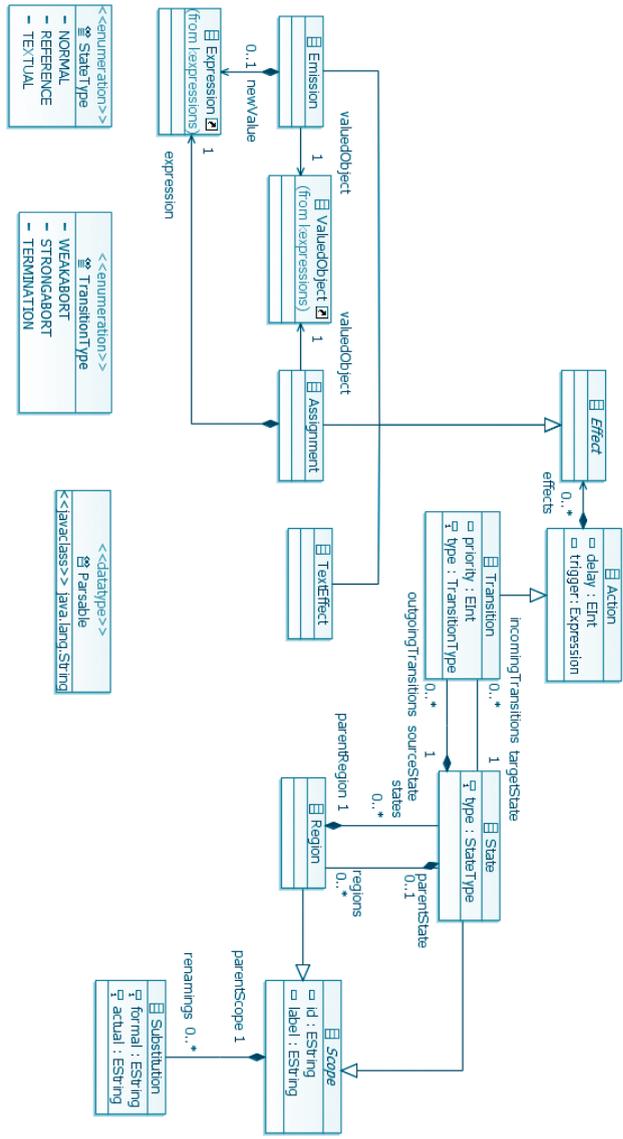


Figure 3.4.1. SCCharts meta model (simplified): Abstract SCCharts model representation (adapted from [MSvH14])

3.4.1 States and Regions

The structure of an SCChart consists of nested states and regions. The top level root state contains one or more regions. These regions contain states and if these states are superstates then they contain regions again. This structure suggests to be directly reflected in the meta model as well. Figure 3.4.1 shows that a Region object contains an arbitrary number of State objects or no State object at all. A State object itself has one parentRegion but contains an arbitrary number of Region objects or no Region object. Each region has one parentState as well. Both states and regions are modeled as Scopes. A Scope basically has a unique id and a label, which is not necessarily unique.

3.4.2 Transitions

For transitions one could imagine the following three major and plausible design decisions:

- (i) A source state contains its outgoing transitions
- (ii) A parent region contains all states and transitions
- (iii) The root state contains all transitions

Both options (ii) and (iii) make it harder to navigate through an SCChart, e. g., when transforming it, because states or their transformed versions not necessary exist yet. Furthermore, it is harder to identify the set of outgoing transitions and these must be filtered (and at least be cached) for usage. Additionally, option (iii) would make a modular SCChart design and lazy loading harder or even impossible. For these reasons we decided for option (i) and let the source state contain all its outgoing transitions as it is reflected in the meta model shown in Figure 3.4.1.

A State object contains zero or an unlimited number of outgoingTransitions. Each Transition object has exactly one sourceState it is contained in and exactly one targetState. The latter is modeled as a (non-containment) reference to another State object.

Transitions carry a priority and a type. The TransitionType is telling whether the transition is a strong abort, a weak abort, or a termination.

Transition objects are derived from Action objects.

3. The SCCharts Language

3.4.3 Actions

An Action object has an Expression trigger that defines whether the action or transition can be executed or taken. The delay is an integer value that defines an optional count delay. Additionally, there are properties such as immediate and label that are omitted in the simplified meta model shown in Figure 3.4.1.

An action has and contains none or an arbitrary number of effects listed as Effect objects. There are different subtypes for such Effect objects, e. g., assignments to variables or emissions of signals.

3.4.4 Expressions

Expressions of type Expression are used in assignments, emissions, and triggers. They are handled by another meta model outside of the SCCharts meta model. Expressions do not belong to the essential part of SCCharts and are meant to be interchangeable and re-usable in other contexts such as SCGs (cf. Section 2.7 on page 41).

3.4.5 Syntax Validation

The meta model makes only a few basic restrictions. E. g., it allows the target state of a transition to belong to another region as the source state. However, such *inter-level transitions* are forbidden in SCCharts although such invalid SCCharts could be represented with this meta model and hence be modeled.

Having a less restrictive meta model is desired to simplify modeling of SCCharts. If the meta model was more restrictive then the modeler would not be able to save a model (e. g., in its XML representation) or at least a textual model could not be parsed which would prevent the model from being drawn visually. However, it might be much easier to identify erroneous models if at least a (partly) graphical representation is available.

Empty regions are another example for a necessary validation. The meta model allows to model a region without a state but this would not lead to a valid SCChart. However, it may be desirable to be able to save a model even when it has not been modeled completely yet.

3.4. Abstract Syntax

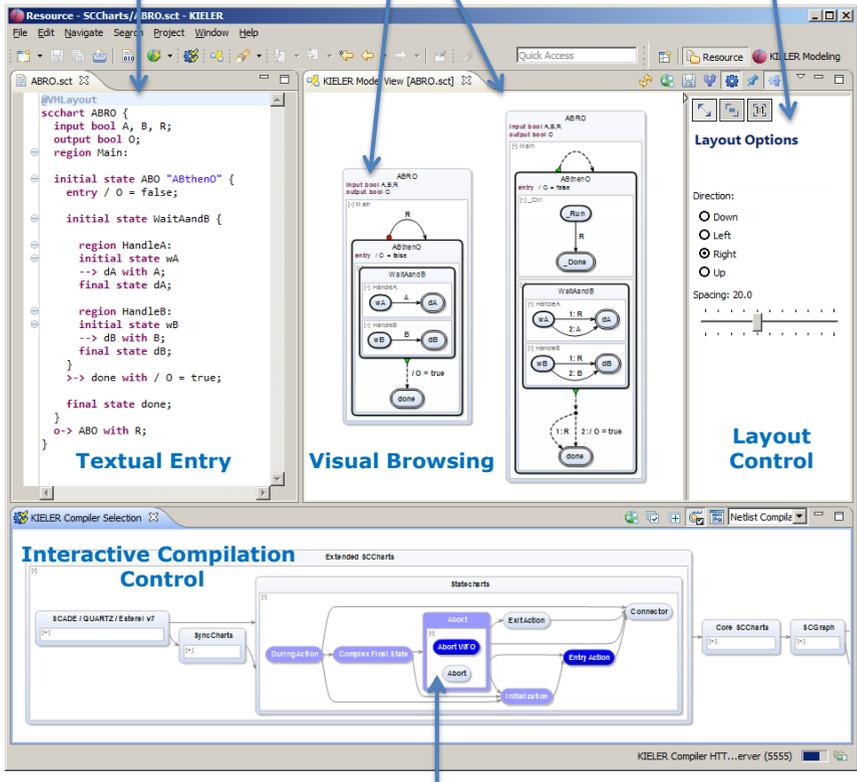
For the above reasons, the design decision is to accept a wider range of syntactically invalid SCCharts w. r. t. the meta model but to alert the user using *model validation* techniques combined with error or warning markers for all the situations that the meta model allows but that would lead to incorrect SCCharts.

Interactive Incremental Compilation

Interactive incremental compilation differs from traditional compilation in that the user who uses the compiler has more control over the compilation in regards to the compilation strategy and intermediate results. This way, the user gains more information about the internal compilation process and the meaning of language features. A first impression is given in Figure 4.0.1. It shows the KIELER SCCharts tool suite, which serves as reference implementation of the ideas around interactive incremental compilation. The language for modeling is SCCharts, which is edited and persisted in a textual format (SCT). On the left side of Figure 4.0.1 a textual editor for SCT is shown. The user (1.) creates or edits the model in this textual editor. A synthesized diagram, the graphical SCChart, is automatically created from the textually specified model. The diagram is shown on the left side of the middle window in Figure 4.0.1 and allows to visually browse the model. The user then can use the Interactive Compilation Control window to influence the compilation, i. e., influence the Model-to-Model (M2M) transformations that are applied to the source model. These transformations can be a series of incremental compilation steps from the original model down to, e. g., C, Java, or VHDL. In this example the modeler (2.) selected the Abort feature to be transformed. Furthermore, he or she selected a specific Abort transformation to be applied. They may then (3.) inspect the intermediate compilation result after this Abort transformation has been applied in the Visual Browsing window on the right side. Both visual models, the one before applying this transformation and the intermediate result (model) after applying this transformation are depicted next to each other

4. Interactive Incremental Compilation

1. Edit SCT code
3. Inspect original + transformed SCChart
4. Adjust layout



2. Select transformations

Figure 4.0.1. KIELER SCCharts tool annotated with high-level user story for interactive model-based compilation (adapted from [MSVH14]).

and can be visually inspected, compared, and studied by the modeler. (4.) The Layout Control window allows to fine-tune the graphical views of the SCChart. The Visual Browsing window is updated whenever any input in any of the other three windows changes, e. g., when the model is changed, when the compiler selection is changed, or if the layout options are modified.

4.1. Single-Pass Language-Driven Incremental Compilation (SLIC)

Advantages: For a modeler the possibility to view not only the original model and the final result, but also the effects that different transformation/compilation phases or options have on the model, can help to understand the exact semantics of different features or combined language constructs. It can also help to fine-tune the original model to optimize intermediate models and ultimately the resulting code. Furthermore, the tool smith can validate the compiler incrementally, i. e., one language feature at a time. This compiler validation support is desirable for any language and compiler and is essential for a compiler of a language that targets safety-critical systems.

Model-Based Compilation: Model-based compilation has been studied earlier as summarized in Section 2.8 on page 45. Interactive incremental compilation is based on a white box compiler that exposes its internal structure and allows user interaction. It further requires all intermediate results to be valid models in the sense of the modeling language to maintain understandability for the compiler user, the modeler. Internal compilation steps are the building blocks of an interactive incremental compiler and consist of a series of M2M transformations from one intermediate result to the next. Each M2M transformation can be considered to be one *increment* in the compilation process. It is *interactive* because of integral user control of the selected and processed specific M2M transformations. This is a *model-based compilation* strategy that is based on the SLIC approach as described in the following.

4.1 Single-Pass Language-Driven Incremental Compilation (SLIC)

Single-Pass Language-Driven Incremental Compilation (SLIC) [MSvH14] is an incremental model-based compilation strategy based on a series of M2M transformations applied to a source model in order to compile it down to some target code such as C, Java, or VHDL. SLIC has four essential properties.

4. Interactive Incremental Compilation

1. *Single-pass*: Each transformation is only applied once to the model.
2. *Increment*: Each transformation can be considered an increment of the compilation, e. g., by expanding exactly one feature.
3. *Language-driven*: The order in which the transformations are applied is derived from the language features and their transformations.
4. *Intermediate models*: All intermediate results are self-contained valid models and can be fully inspected also by the modeler.

The SLIC approach has several advantages:

- ▶ Deriving complex language constructs as *syntactic sugar* from a small set of elementary constructs allows a compact, lightweight definition of the core semantics.
- ▶ Intermediate transformation results are open to inspection, which can also help certification for safety-critical systems, e. g., tool qualification [Rie13].
- ▶ Existing languages and infrastructures for M2M transformations allow high-level formulations of transformations that can also serve as unambiguous definitions of advanced language constructs.
- ▶ Complex transformations are broken into individual components. This approach allows a divide-and-conquer validation and maintenance strategy for the compiler.
- ▶ The modularization of the compilation facilitates language/compiler subsetting.

Figure 4.1.1 sketches the SLIC approach from an Original Model as created or edited by the modeler to a Fully transformed Model, which ultimately can be runnable C or Java code or VHDL. The Intermediate Models can be completely inspected by the modeler and they are self-contained. This means there is no information hidden and the models are valid by themselves preserving the semantics of the original model.

However, it is not a trivial question how a SLIC transformation sequence as shown in Figure 4.1.1 can be designed. Such a sequence is referred to as a *SLIC schedule*. The design of the concrete transformations plays a key role here and the SLIC schedule imposes design decisions on the transformations and vice versa. The following sections will discuss these design decisions

4.1. Single-Pass Language-Driven Incremental Compilation (SLIC)

and their consequences on the SLIC schedule after grounding essential SLIC foundations.

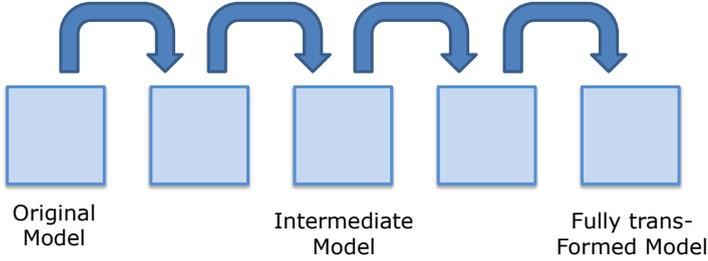


Figure 4.1.1. Single-Pass Language-Driven Incremental Compilation as an incremental model-based compilation strategy: A concrete *SLIC schedule*, i. e., a single-pass sequence of model transformations.

4.1.1 SLIC Foundations

Given a set of *language features* Φ consisting of features $f \in \Phi$ and a set of *transformation rules* T consisting of transformations $\tau_f \in T$.

Expanding Features: Each $\tau_f \in T$ expands a model (program) m that uses f into another, semantically equivalent¹ model m' that does not use f . We say “ f is expanded by τ_f ”.

4.1.1 Definition ($Expand_{\tau_f}$). $Expand_{\tau_f} := \{f \mid f \in \Phi \wedge f \text{ is expanded by } \tau_f\}$. $Expand_{\tau_f}$ is defined to have cardinality of 1, i. e., each transformation τ_f handles exactly one feature and this is f .

4.1.2 Definition (Base Feature). The set $B_T := \{f \mid \nexists \tau_f \in T \wedge f \in Expand_{\tau_f}\} \subseteq \Phi$ is the set of all *base features* w. r. t. T .

4.1.3 Definition (Extended Feature). The set $F_T := \{f \mid \exists \tau_f \in T \wedge f \in Expand_{\tau_f}\} = \Phi \setminus B_T$ is the set of all *extended features* w. r. t. T .

¹In case of SCCharts, *equivalency* is comprised w. r. t. to well known features of, e. g., SyncCharts, Esterel, or Statecharts. However, in general, a more formal definition is preferable.

4. Interactive Incremental Compilation

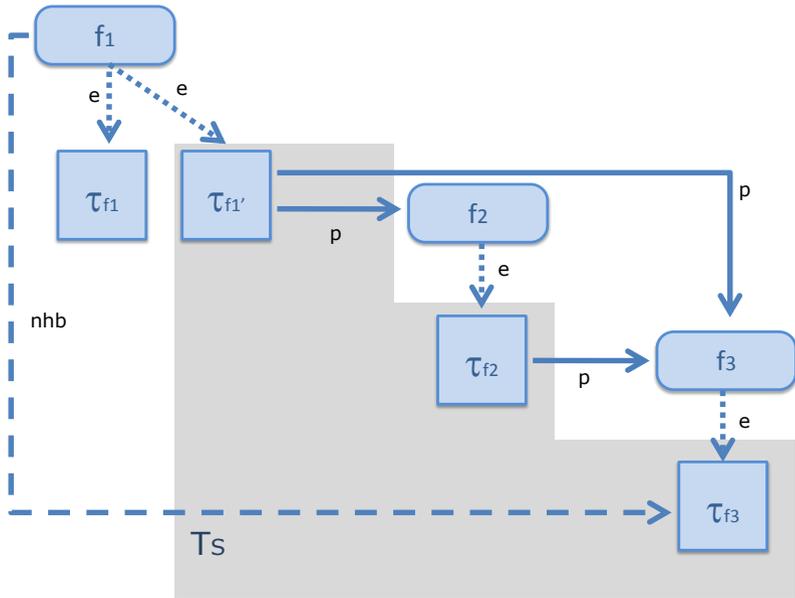


Figure 4.1.2. Example SLIC features f_1, f_2, f_3 , SLIC transformations $\tau_{f_1}, \tau_{f_1'}, \tau_{f_2}, \tau_{f_3}$ and their expand (e and dotted line), produce (p and solid line), and not-handled-by (nhb and dashed line) order relations are shown. A transformation choice T_S for a selection S of transformations is visualized.

For the sake of simplicity, in the following, we will only consider a fixed set T of transformations and hence only write F (instead of F_T) for the extended features regarding this set T . Furthermore, we will call the extended features $f \in F$ just “features”.

Alternative Transformations: For all features $f \in F$ there must be at least one transformation τ_f available to expand this feature. But there is no restriction on the number of such transformations. If there are more distinct transformations, say τ_{f_1} and τ_{f_2} both expanding feature f , then its the user’s choice to select one of these *alternative* transformations (or take a predefined preference).

4.1. Single-Pass Language-Driven Incremental Compilation (SLIC)

4.1.4 Definition (Alternative Transformation). $\tau_f \in T$ is an *alternative transformation*, iff there exists another transformation $\tau'_f \in T$ expanding the same feature f but $\tau'_f \neq \tau_f$.

To expand a feature f , only one of the alternative transformations can be executed in the end. A *transformation selection* S gives a precedence for one transformation which is chosen automatically if no other, concrete (alternative) transformation is chosen.

4.1.5 Definition (Transformation Selection). A *transformation selection* S is defined as follows: $F \rightarrow T, f \mapsto S(f) = \tau_f$. S is bijective.

4.1.6 Definition (Transformation Choice). A *transformation choice* T_S is defined as $T_S := \{\tau_f \mid f \in F \wedge \tau_f = S(f)\}$.

We define T_S to be the set of transformations τ_f for all features $f \in F$ where we applied the transformation selection S to each feature f in order to get the selected (possibly alternative) transformation. Clearly, $T_S \subseteq T$.

4.1.7 Theorem. *All transformations $t \in T_S$ are non-alternative transformations.*

Proof. Sketch: T is the set of all transformations for features $f \in F$. T can contain alternative transformations. T_S is a projection on T derived from T when applying $S(f)$ for all f from all $\tau_f \in T$. Let $\tau_{f_1} \in T$ and $\tau_{f_2} \in T$ with $\tau_{f_1} \neq \tau_{f_2}$ be (the only two) alternative transformations for a feature $f \in F$ and let $S(f) := \tau_{f_1}$ be the transformation selection for a transformation choice T_S . Clearly, T_S will contain only τ_{f_1} but not τ_{f_2} when applying the projection on all transformations of T . By definition τ_{f_1} is only an alternative transformation if also τ_{f_2} exists in the same set T_S , τ_{f_1} cannot be an alternative transformation in this context. In other words τ_{f_1} is the (only) choice in T_S for expanding a feature f . \square

Example: Fig 4.1.2 presents an example of three features $f_1, f_2,$ and f_3 and four transformations τ for these features. τ_{f_3} is a transformation expanding feature f_3 , τ_{f_2} is a transformation expanding feature f_2 , and both transformations τ_{f_1} and τ'_{f_1} are expanding feature f_1 . Clearly, τ_{f_1} and τ'_{f_1} are alternative transformations. A selection S is implicitly given by a transformation choice T_S which is the set of transformations $\tau'_{f_1}, \tau_{f_2},$ and

4. Interactive Incremental Compilation

τ_{f_3} . Note that τ_{f_1} is not part of the transformation choice (because it is not selected according to S).

Note that in the following, we implicitly always require a concrete transformation selection S and all transformations τ are picked to be part of a transformation choice T_S regarding this transformation selection S . Not picking τ from T but from some T_S frees us from considering transformation alternatives.

Producing Features: A transformation $\tau_f \in T_S$ may produce other features $f_p \in F$ when expanding a feature f . These other features need to be expanded afterwards by other transformations in order to retrieve a model that contains base features only in the end.

4.1.8 Definition ($Produce_{\tau_f}$). $Produce_{\tau_f} \subseteq F$ is the set of features $f_p \in F$ that a transformation $\tau_f \in T_S$ may produce when expanding a feature f .

Note that a transformation is never required to produce any features $f \in Produce_{\tau_f}$ but *may* produce any of them depending on the concrete model. Further note that τ_f must not produce f itself, i. e., $f \notin Produce_{\tau_f}$ because trivially this would result in a cycle and not be *single-pass* any more.

Since all features $f_p \in Produce_{\tau_f}$ are (possibly) produced by τ_f and expanded by all transformations τ_{f_p} , these transformations τ_{f_p} should be performed *after* τ_f to prevent multiple applications of transformations.

Handling Features: A transformation $\tau_f \in T_S$ that expands f is able to *handle*, i. e., conserve, features $f_{hb} \in F$.

4.1.9 Definition ($Handle_{\tau_f}$). $Handle_{\tau_f} \subseteq F$ is the set of features $f_{hb} \in F$ that a transformation $\tau_f \in T_S$ is able to handle when expanding a feature f .

Trivially, $Handle_{\tau_f}$ must include f such that τ_f is able to expand f .

Not Handling Features: A transformation $\tau_f \in T_S$ that expands f may *not* be able to handle, i. e., conserve, some other features $f_{nhb} \in F$.

4.1.10 Definition ($NotHandles_{\tau_f}$). $NotHandles_{\tau_f} \subseteq F$ is the set of features $f_{nhb} \in F$ that a transformation $\tau_f \in T_S$ cannot handle when expanding some feature f .

4.1. Single-Pass Language-Driven Incremental Compilation (SLIC)

Hence, it follows that $NotHandles_{\tau_f} = F \setminus Handle_{\tau_f}$.

Since all features $f_{nhb} \in NotHandles_{\tau_f}$ cannot be handled by τ_f , these features must be expanded by other transformations $\tau_{f_{nhb}}$ before τ_f is applied to the model.

Feature Transformation Dependencies: Based on $Expand_{\tau_f}$, $Produce_{\tau_f}$, and $NotHandles_{\tau_f}$ we define the following relations on F and T_S :

4.1.11 Definition (Feature Expansion Order). The *expansion order* \rightarrow_e is defined as $f \rightarrow_e \tau_f$

We say that “ τ_f expands feature f ”. We refer to this order as *expansion order*.

4.1.12 Definition (Feature Production Order). $\tau_g \rightarrow_p f$ iff $f \in Produce_{\tau_g}$.

We say that “ τ_g produces f ” and refer to this order as *production order*.

4.1.13 Definition (Feature Handling Order). $f \rightarrow_{nhb} \tau_g$ iff $f \notin Handle_{\tau_g}$.

We say “ f is not handled by transformation τ_g ” and refer to this order as *handling order*. An alternative definition is to use $NotHandles_{\tau_g}$: $f \rightarrow_{nhb} \tau_g$ iff $f \in NotHandles_{\tau_g}$.

Example: Fig 4.1.2 gives an example for produce and not-handled-by orders: Transformation τ'_{f_1} , which expands feature f_1 , produces feature f_2 as well as feature f_3 . Transformation τ_{f_2} , which expands feature f_2 , produces feature f_3 . Transformation τ_{f_3} , which expands feature f_3 , cannot handle feature f_1 .

4.1.2 Deriving a SLIC Schedule

Figure 4.1.1 suggests a sequence of transformations to be executed one after the other where every transformation expands one feature and every transformation is only executed once per compilation run for a model. This section explains how to use the SLIC foundations to derive such a sequence. We call this single-pass sequence a static “SLIC schedule”. We will now use the definitions from the previous section to define a SLIC schedule.

4. Interactive Incremental Compilation

Transformation Dependencies: The expansion order is folded into the production order and into the handling order. The goal is to retrieve a pure *transformation* dependency order that can be used to derive a SLIC schedule.

4.1.14 Definition (Transformation Production Order). $\tau_f \rightarrow_p \tau_g$ iff $\tau_f \rightarrow_p g \wedge g \rightarrow_e \tau_g$.

This means transformation τ_f must be executed before transformation τ_g because τ_f produces a feature g that τ_g expands.

4.1.15 Definition (Transformation Handling Order). $\tau_f \rightarrow_{nhb} \tau_g$ iff $\tau_f \rightarrow_e f \wedge f \rightarrow_{nhb} \tau_g$.

This means that transformation τ_f must be executed before transformation τ_g because τ_f expands a feature f that τ_g cannot handle.

After having defined the orders on transformations for handling or producing features, we can now derive a SLIC transformation order.

4.1.16 Definition (SLIC Transformation Order). $\tau_f \rightarrow \tau_g$ iff $\tau_f \rightarrow_p \tau_g \vee \tau_f \rightarrow_{nhb} \tau_g$.

This means that in any concrete SLIC schedule, which again is the SLIC transformation sequence, τ_f must be executed before τ_g whenever τ_f produces a feature g that τ_g expands or a feature f expanded by τ_f is not handled by τ_g or both. We say “ τ_f must precede τ_g ”.

Example: Fig 4.1.2 presents an example for three features f_1 , f_2 , and f_3 together with their expanding transformations τ'_{f_1} , τ_{f_2} , and τ_{f_3} that were selected. It gives an example for expand, produce, and not-handled-by dependencies between features and transformations. When lifting the *feature* production/handling order to a pure *transformation* production/handling order, then for finding a SLIC schedule, feature expansion can be omitted as shown in Fig 4.1.3. Transformation τ_{f_2} inherits the incoming produce dependencies for the feature f_2 it expands, namely from transformation τ'_{f_1} . Similarly, transformation τ_{f_3} inherits the incoming produce dependencies for the feature f_3 it expands, namely from transformations τ'_{f_1} and τ_{f_2} . Additionally, τ'_{f_1} inherits the outgoing not-handled-by dependencies from the feature f_1 it expands, namely to transformation τ_{f_3} .

4.1. Single-Pass Language-Driven Incremental Compilation (SLIC)

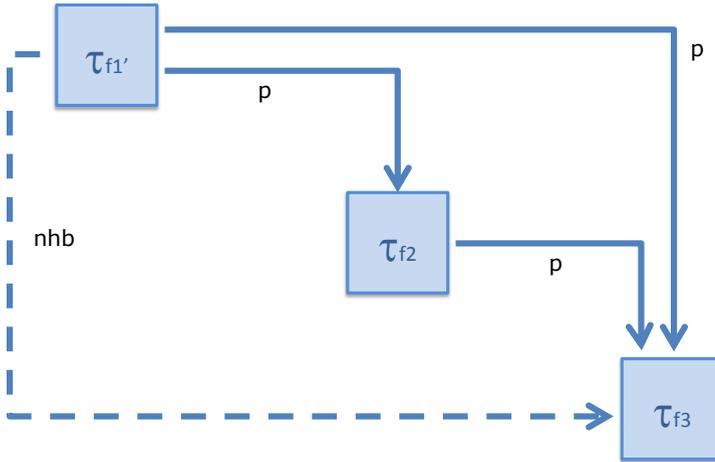


Figure 4.1.3. Example SLIC transformations τ'_{f_1} , τ_{f_2} , τ_{f_3} regarding a transformation the choice T_S (cf. Figure 4.1.2) are shown with their produce (p) and not-handled-by (nhb) order relations.

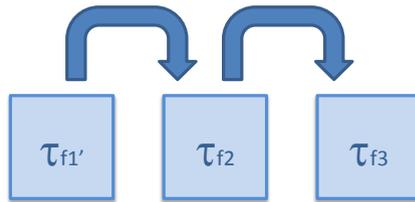


Figure 4.1.4. Example SLIC schedule derived for transformations τ'_{f_1} , τ_{f_2} , τ_{f_3} (cf. Figure 4.1.3)

Deriving a concrete transformation sequence is to find a SLIC schedule, which can be accomplished by using, e.g., a topological sort algorithm [OW02] on the SLIC transformation order. Note that there may be different concrete but valid SLIC schedules, all meeting the same SLIC order. Further note that in case of semantic variations, artificial production

4. Interactive Incremental Compilation

dependencies could help finding a deterministic and desired SLIC schedule (see discussion about *SLIC Variations* below).

4.1.17 Definition (SLIC Schedule). If the SLIC order is acyclic (see *SLIC feasibility*), a static *SLIC schedule* is an assignment of indices $i(\tau_f)$ to each transformation τ_f for all features $f \in F$ such that $\tau_f \rightarrow \tau_g \Rightarrow i(\tau_f) < i(\tau_g)$.

Example: Fig 4.1.4 shows a concrete SLIC schedule, i. e., the transformations τ'_{f_1} , τ_{f_2} , and τ_{f_3} (cf. Figure 4.1.2 on page 92) ordered by their dependencies. Note that in this particular example, there is just one feasible SLIC schedule (see below).

4.1.3 SLIC Key Questions

When developing a SLIC transformation sequence, three non-trivial aspects play a key role: 1. Does a feasible SLIC schedule exist? 2. What is a concrete SLIC schedule (i. e., the order in which the transformations should be executed)? 3. Are any additional semantic variations not resolved by given dependencies?

SLIC Feasibility: Given a set of features F , transformations $\tau \in T$ for all features $f \in F$, and a transformation selection S on T , the question is to determine if a feasible SLIC order exists. If the SLIC order \rightarrow for all $\tau_f \in T_S$ is acyclic then a feasible SLIC order exists.

SLIC Schedule: If a feasible SLIC order exists, the SLIC order \rightarrow on the transformations T_S gives a concrete SLIC schedule that is an assignment of indices reflecting the order in which these transformations can be executed according to the SLIC paradigm.

SLIC Variations: In case of semantic variations depending on the order in which two features are eliminated and no \rightarrow_{nhb} or \rightarrow_p dependency exists, an *artificial production dependency* should be used to clarify the exact order in which transformations should be processed (see discussion of entry/exit actions and strong aborts in Section 5.2.5 on page 140).

4.1. Single-Pass Language-Driven Incremental Compilation (SLIC)

	produces \rightarrow_p																
not-handled-by \rightarrow_{nhb}	Weak Suspend	Deferred	History	Static	Valued Signal	Pure Signal	Suspend	Pre	Count Delay	During	Complex Final	Abort	Const	Exit	Initialization	Entry	Connector
Weak Suspend		\rightarrow_p		\rightarrow_p						\rightarrow_p	\rightarrow_p						
Deferred										\rightarrow_p					\rightarrow_p		
History				\rightarrow_p											\rightarrow_p	\rightarrow_p	
Static															\rightarrow_p		
Valued Signal						\rightarrow_p									\rightarrow_p		
Pure Signal										\rightarrow_p							
Suspend			\rightarrow_{nhb}							\rightarrow_p							
Pre					\rightarrow_{nhb}	\rightarrow_{nhb}					\rightarrow_p				\rightarrow_p		
Count Delay							\rightarrow_{nhb}			\rightarrow_p						\rightarrow_p	
During			\rightarrow_{nhb}								\rightarrow_p				\rightarrow_p		
Complex Final			\rightarrow_{nhb}									\rightarrow_p			\rightarrow_p		
Abort			\rightarrow_{nhb}						\rightarrow_{nhb}		\rightarrow_{nhb}				\rightarrow_p	\rightarrow_p	\rightarrow_p
Const																	\rightarrow_p
Exit												\rightarrow_{nhb}					
Initialization														\rightarrow_{nhb}		\rightarrow_p	
Entry												\rightarrow_{nhb}					\rightarrow_p
Connector																	\rightarrow_p

Figure 4.1.5. SCCharts transformation dependency details

4.1.4 SLIC Order for SCCharts Compilation

As an example, we now discuss the SLIC order for compiling SCCharts. We focus on the “expand features” part (see compiling SCCharts overview, (1) Expand in Figure 5.0.1 on page 114), but the same principles apply to the all other incremental compilation steps for compiling SCCharts down to any target such as C code or VHDL. Figure 4.1.5 presents the transformation dependencies for Extended SCCharts features given by a produced (\rightarrow_p) or not-handled-by (\rightarrow_{nhb}) relation for a pair of two transformations. The produced orders can be read above the diagonal line, the not-handled-by orders can be read below the diagonal line. The data in Figure 4.1.5 follows from the transformation implementation for every listed feature. Alternative transformations exist but can be omitted here as these transformations have the same produced or not-handled feature dependencies.

4. Interactive Incremental Compilation

The concrete feature transformations and their semantics are discussed later in Chapter 5. Here, we will focus on their dependencies and derive a SLIC schedule from these dependencies.

Figure 4.1.6 shows the transformation dependencies for all Extended SCCharts features. The Extended SCCharts features are grouped into three categories:

- C1: *Basic Statecharts features*. Common features of various statechart dialects as known from Harel Statecharts [Har87], e. g., entry actions, exit actions or strong and weak preemption.
- C2: *SyncCharts features*. Extended SCCharts are quite rich and include, for example, language features proposed for SyncCharts [And03], e. g., synchronous signals or suspension.
- C3: *Further features*. Extended SCCharts include additional features adopted from other synchronous languages such as weak suspension from Quartz [Sch10] or deferred transitions from SCADE. We also categorize History transitions here for sublanguage purposes (cf. Section 4.1.5), even though they were part of the original Harel Statecharts.

The transformation rules for SCCharts are not only used to *implement* M2M transformations, but also serve to unambiguously *define* the semantics of the extensions. Each such transformation is of limited complexity, and the results can be inspected by the modeler or certification agencies [Rie13]. This is something we see as a main asset of SCCharts for the use in the context of safety-critical systems.

The SLIC order for SCCharts is acyclic which can be validated by visual inspection of Figure 4.1.6, where all transformations $\tau_f \in F$ are ordered top-to-down according to \rightarrow_p (solid arrows) and \rightarrow_{nhb} (dotted arrows).

For deriving the SLIC schedule, Figure 4.1.7 shows a version without groups that can be easily sorted topologically in different ways. One of possibly many concrete SLIC schedules is also visible by the concrete numbering, as each $\tau_f \in F$ is prefixed with a “ $i(f)$.” label that shows its SLIC schedule index.

4.1. Single-Pass Language-Driven Incremental Compilation (SLIC)

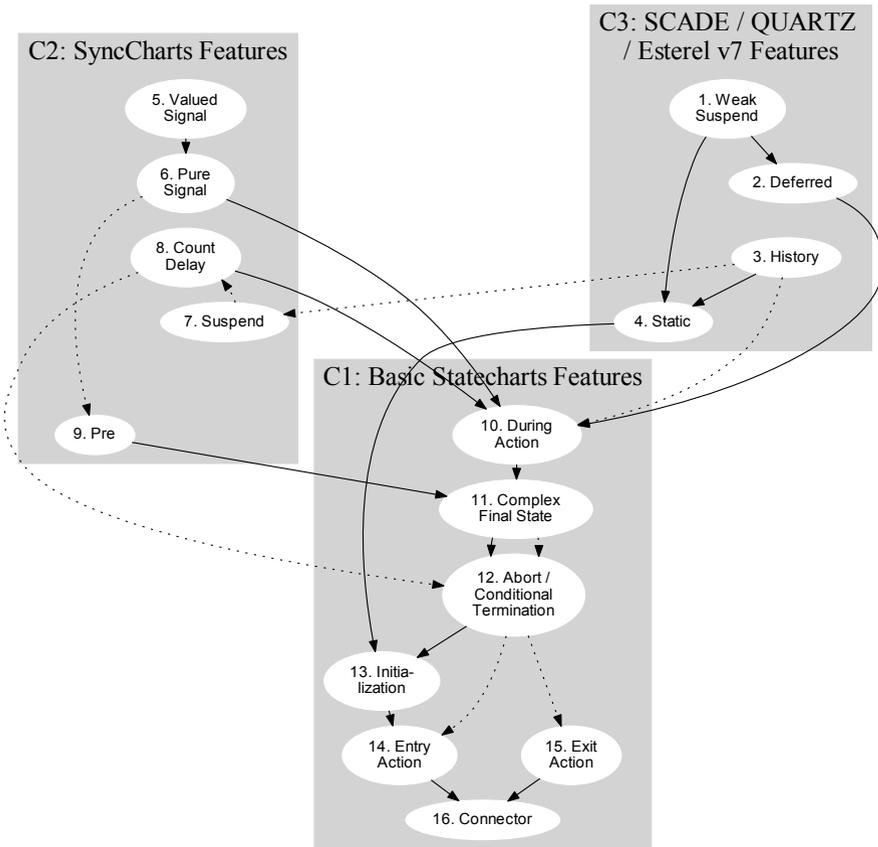


Figure 4.1.6. SCCharts transformation dependencies

4.1.5 SLIC Language and Sublanguage

A language consists of a set of extended and base features (syntax) together with transformations that implicitly define the semantics of the extended features. Note again that at least one expanding transformation is required for every extended feature that is not a base feature.

4. Interactive Incremental Compilation

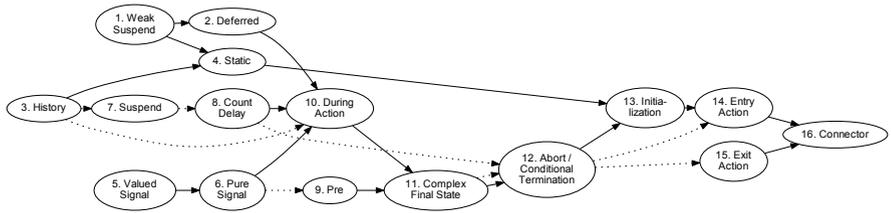


Figure 4.1.7. SCCharts transformation dependencies without groups

4.1.18 Definition (Language). Given an expansion order \rightarrow_e and a production order \rightarrow_p , a *language* L is a triple (B_T, F_T, T) of base features $b \in B_T$, extended features $f \in F_T$ and transformations $\tau \in T$ where $\forall f \in F, \tau \in T : (f \rightarrow_e \tau \wedge \tau \rightarrow_p g) \Rightarrow g \in \Phi$.

This means that regarding all transformations $\tau \in T$ for features $f \in F_T$, if these τ produce any features g then these produced features g will be in F_T or B_T again. Recall that Φ is the set of *language features* which is constituted by B_T and F_T (cf. Section 4.1.1 on page 91).

A *sublanguage* is a language with possibly fewer features and possibly fewer transformations.

4.1.19 Definition (Sublanguage). A *sublanguage* L' is a triple $(B'_{T'}, F'_{T'}, T')$ regarding a language $L = (B, F, T)$, where L' is a language and $B'_{T'} \subseteq B_T \wedge F'_{T'} \subseteq F_T \wedge T' \subseteq T$. We also write $L' \subseteq L$.

For example, the SCCharts language proposal is very rich which nicely illustrates how a wide range of different features proposed in SyncCharts, SCADE etc. can be grounded in a small set of Core SCCharts features. However, this variety of features may be overwhelming for the user. Also, some features might be rarely used in practice or not be appropriate for certain domains (such as, in our experience, suspension), or might be considered non-desirable for some reasons (such as history transitions, which increase the state space drastically).

Example: A conservative approach to ensure a subset of features and transitions is a sublanguage would include in F' all features whose SLIC

4.1. Single-Pass Language-Driven Incremental Compilation (SLIC)

schedule index is above a certain value. E. g., for SCCharts, if we define F' such that it includes all features with schedule index 10 and higher, we would obtain all features in category C1 which would be a feasible language subset. However, the definition of a sublanguage permits other subsets for features and transformations as well. E. g., the subset of SCCharts features with indices 12, 13, 14, 15, 16 which includes Aborts (index 12) and all subsequently produced features together with the respective transformations, together would also be a feasible sublanguage.

4.1.6 SLIC Design Challenges

Modularization: Concerning the feature/transformation categories C1, C2, and C3, we observe that inter-category precedence constraints are only of type $C3 \rightarrow C2$, $C3 \rightarrow C1$, and $C2 \rightarrow C1$. Thus, we can modularize our schedule according to categories: First transform away all features from C3, then all features from C2, and finally all features from C1. This clearly is an optional design choice for concrete transformation implementations and for restricting possible SLIC schedules.

Acyclicity: Whether the SLIC order is acyclic or not is not an inherent property of the language features themselves, but depends on how exactly the transformations for the features are defined. For example, we might have defined our transformation rules τ_f such that each extended feature f would be transformed directly into Core SCCharts by τ_f alone ($Prod_{\tau_f} = \emptyset$), while preserving all other features ($Handle_{\tau_f} = F$). This would have resulted in an empty SLIC order that would be trivially acyclic. However, this would have hindered the purpose of modularizing the compilation, since at least some of the transformation rules would have to be unnecessary complex.

Complexity: We wish the transformation for each f to be rather lean and not complex. For that purpose τ_f may make use of other features, as reflected by a non-empty $Prod_{\tau_f}$. Furthermore, in defining τ_f , we may restrict the models to be transformed to not contain all features in F , meaning that $Handle_{\tau_f}$ may be small. However, care must be taken not to introduce cycles this way. Hence, the more “primitive” a feature f is, the more features τ_f

4. Interactive Incremental Compilation

should be able to handle. This enables other transformations to produce this feature which may help to reduce their complexity.

Trade-offs: A transformation rule should be lean and not complex for the purpose of readability and maintainability. It should make use of other features if possible but also include optimization.

On the one hand, the downside of using too many other features and too few optimizations is that for a compilation run the model has to be processed too often and the intermediate model may get quite large which both is inefficient. Another problem with really lean transformations is that this is more prone to lead to dependency cycles, which are forbidden.

On the other hand, the downside of using too few other features and too many optimizations is that a transformation quickly becomes complex and hard to maintain. A further problem with more complex transformations is that a lot of processing used in a transformation may also be done in another transformation. Such *code-clones* [Kri07] amplify maintainability problems. Another downside is that it gets hard or impossible to define feasible sublanguages because this more monolithic approach lacks modularization.

4.2 Interactive Compilation

Interactive compilation leverages the SLIC approach in order to support the modeler to build reliable models. It also helps the compiler developer to build a reliable compiler. This section first compares the interactive compilation to traditional compilation. It then points out the interactive compilation user story and further advantages like element tracing. Figure 4.2.1 illustrates the user stories for modeling using traditional compilation and for modeling using interactive model-based compilation.

4.2.1 Traditional User Story

Traditionally, a compiler is a black box as depicted in Figure 4.2.1a. The user creates or edits his or her model textually or graphically, then compiles it, and later inspects the final resulting code. Typically, the user does not

4.2. Interactive Compilation

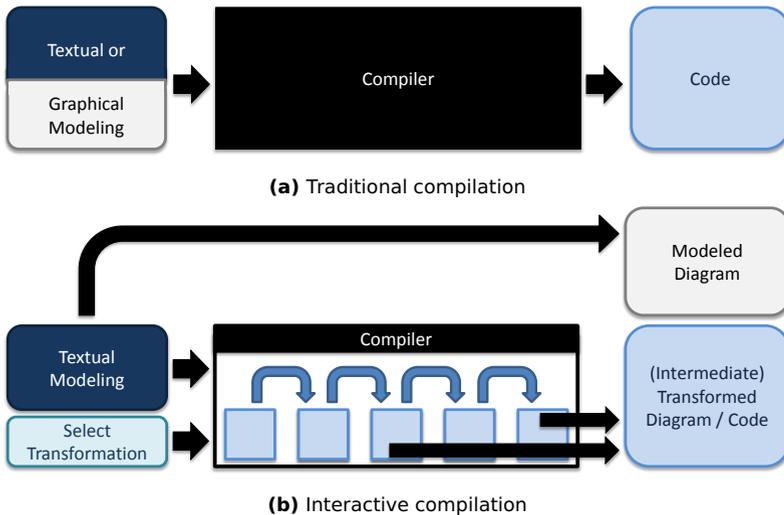


Figure 4.2.1. User stories for modeling using traditional compilation and for modeling using interactive model-based compilation [Mot14b]

know anything about the internals of a compiler and does not see or at least not understand any intermediate results. Sometimes this might be sufficient because the internals and intermediate results do not matter.

Further, it generally does not help to have access to intermediate results because the user typically has not enough knowledge to fully understand or interpret these.

However, there are drawbacks of this traditional compilation approach.

Understanding the language: Especially for beginners of a language there is little guidance what specific language features mean, i. e., how they compile. This often results in a frustrating trial and error scenario.

Fine-tuning: Making optimizations or fine-tuning of the compilation result is difficult because intermediate implications are hidden to the user.

Compiler developer: If an error inside the compiler is exposed, it may be hard for the compiler developer themselves to track it down depending on

4. Interactive Incremental Compilation

the organization and structure of the internals. However, somehow the error has to be fixed in isolation, i. e., without breaking other functional parts of the compiler. This can get difficult.

Extendability: It may only be feasible for the compiler developer to extend a black box compiler. The user is limited to some pre or post processing.

4.2.2 Interactive User Story

Figure 4.2.1b sketches an *interactive compiler* in contrast to its black box counterpart in Figure 4.2.1a. The interactive compiler is a white box compiler. The modeler textually or graphically edits their model. However, we propose *textical modeling* using a textual model editor combined with an automatically synthesized diagram [RSS⁺13] (cf. Section 6.4.4 on page 327). One essential part of an interactive compiler is that it takes not only 1. the source model but also 2. an interactive user compiler selection as an input. This way, the user is able to influence the compilation strategy and the applied transformations. Since the SLIC compilation consists of stepwise model transformations, the user is able to select features that they want to compile. This way, they can influence the transformations that are applied to the source model.

Furthermore, besides the final resulting code, intermediate results are also part of compiler output depending on the interactive compiler selection. These results can be fully inspected by the user because any intermediate result in SLIC is defined to be a valid model.

The advantages of this interactive model-based SLIC compilation approach are discussed in the following and summarized in Table 4.2.1. Note that in this and all following tables of this kind, “+” means “better” or “more” and “-” means “worse” or “less”, “+ or -” does not mean a binary/absolute “yes” or “no”. Furthermore, “++” means “even better”, “--” means “even worse”, and “+/-” means “pros and cons tend to balance”.

Understanding the language: Being able to select to compile certain features only, the modeler is able to see the intermediate compilation result, i. e., how these features are expressed by means of other (more basic) features. This facilitates learning the semantics of language features of

4.2. Interactive Compilation

a modeling language and overall helps in understanding a language. Additionally, not only the static semantics of language features can be studied by the user but also the dynamic semantics as intermediate results may be simulated because these are always valid models.

Understanding the models: A crucial point for developing safety-critical systems is that the final implementation on the system relies on 1. a reliable model and 2. a reliable compiler. In order to retrieve reliable models, the modeler has to understand the language features they use and additionally they must be able to validate that they use the language feature in a correct and intended way in their model. This is also facilitated by being able to simulate and test intermediate results and of course it requires understanding the language in the first place.

Fine-tuning: Being able to actually see and understand the effects of increments of a SLIC compilation, the modeler can compare the intermediate results for different language feature options and value which language feature is better suited for accomplishing a given task. Moreover, there sometimes might be more than one transformation available for expanding one and the same language feature. Being able to compare intermediate results, enables the modeler to choose the better suited transformation choice.

Compiler developer: As a compiler developer, one wants to make sure that the compiler is maintainable. Each transformation should be lean and easy to validate according to the SLIC approach. The interactive compilation enables validating selective transformations, e. g., by regression testing. Also, if an error is exposed in the compiler it is much easier to track it down if transformations can be applied selectively. When fixing an error, it is essential to do this in isolation to avoid breaking other parts of the compiler. Since of the modular SLIC approach, where each transformation just expands one feature, this is much easier to achieve than in a monolithic system. Also, selective regression testing can help to validate compiler fixes.

Extendability: Since the white box compiler basically consists of a selection of M2M transformations applied to models of the input modeling language, the user (modeler) itself is able to extend the compiler and easily

4. Interactive Incremental Compilation

introduce new features with transformations that can be based on other existing features.

Although there are several of advantages using the interactive compilation, the benefits come at the price that the implementation could be a little slower (mostly depending on the SLIC design trade-offs, cf. Section 4.1.6) compared to a pure monolithic compiler. However, for targeting safety-critical systems the advantages clearly are outbalancing because they facilitate a reliable compiler and reliable models. Still, in our experience (cf. Section 7 on page 365) even SLIC had proven to leave many possibilities to optimize while still maintaining reliability.

Table 4.2.1. Comparison of traditional and interactive model-based SLIC compilation and according user stories

	Traditional	Interactive SLIC
Understand language feature	-	+
Understand language	-	+
Compare language features	-	+
Compare compilation options	-	+
Fine tuning	-	+
Choose best suited features	-	+
Choose best suited transformations	-	+
Study static feature semantics	-	+
Study dynamic feature semantics	-	+
Understanding the models	-	+
Maintainability	-	+
Selective validation	-	+
Isolated error fixes	-	+
Extending the language/compiler	-	+
Performance	+	+/-

4.2.3 Element Tracing

When performing a single model transformation during SLIC, the model may be modified within the same meta model or a new model in a different meta model may be created. New elements may be created and others may be deleted. It may be desirable to capture these relations as mappings of a so called *tracing tree* while applying one model transformation after another on a source model. Afterwards, this information can be used to *convergecast* [Lyn96, p. 505], i. e., “send” information from low-level elements (or leaves) of intermediate models or the fully transformed model back to high-level elements of the source model.

Lopes [LHBJ06] studies the general specification of mappings between models in MDE. This inspired the tracing for SLIC. However, Lopes’s approach considers the mapping specification between two distinct meta models, where the tracing in SLIC considers mappings between models of arbitrary meta models, even of the same meta model.

Schulz-Rosengarten [SR14] further elaborated and implemented the element tracing for SLIC in the context of the KIELER project. The tracing is further described as an extension to SLIC elsewhere [RSM⁺16].

Tracing Tree Construction

Figure 4.2.2a shows the idea of the element tracing of a model transformation which transforms model element

- ▶ A into both F and G (1),
- ▶ B into I (2),
- ▶ C and E into H (4), and
- ▶ which does not transform the model element D into anything (3).

Note that for this abstract representation it does not matter whether the two models conform to the same or to two different meta models. Furthermore, it does not matter whether B and I are the same model elements or different model elements with a bijective mapping.

While each transformation is applied, a mapping of source elements and target elements collects the information necessary to construct the *tracing tree* as shown in Figure 4.2.2a.

4. Interactive Incremental Compilation

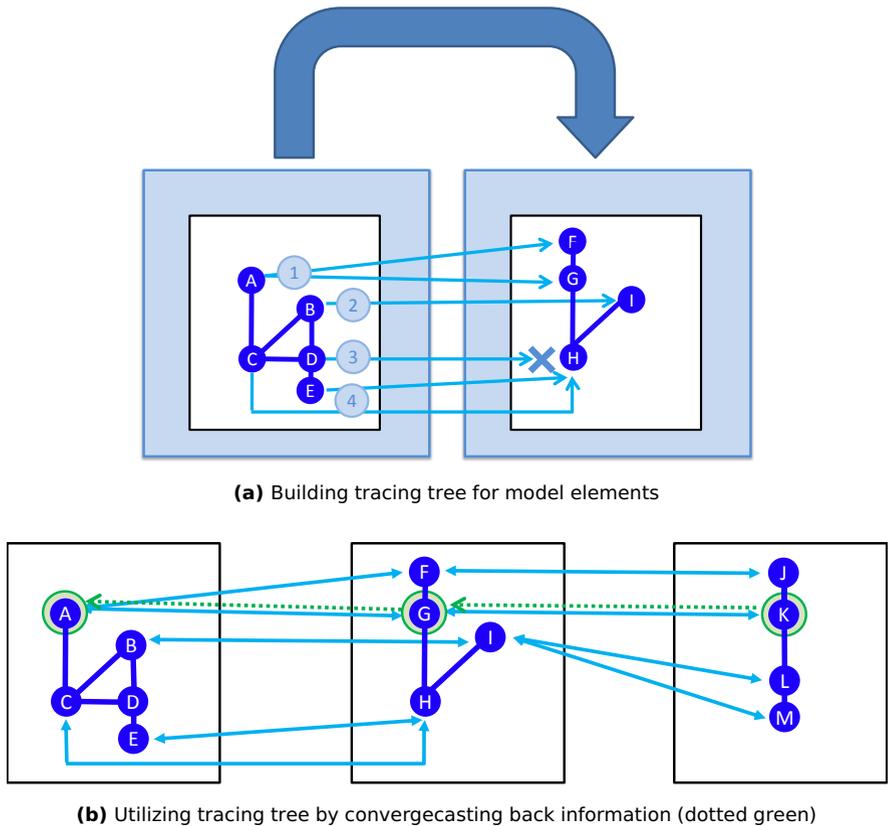


Figure 4.2.2. Element tracing with SLIC and usage

Tracing Tree Usage

Once all transformations have been applied, the tracing tree can be used to convergencast any information from lower-level representations to higher-level representations. This can be useful for example to display simulation data such as an active state or a taken transition in the visual representation of an SCChart or for displaying low-level timing information to the mod-

4.3. Interactive Incremental SCCharts Compilation

eler [FBSvH14]. Figure 4.2.2b illustrates convergencasting from a low-level model element K , which may represent a currently executed (active) code block, to a high-level model element A , which may represent the actual active state in the source SCChart model.

4.3 Interactive Incremental SCCharts Compilation

Consider again Figure 4.0.1 on page 88 that shows the KIELER SCCharts tool and sketches the interactive model-based compilation user story. After introducing the SLIC fundamentals in Section 4.1 that internally play a key role inside the KIELER Compiler, we are now able to match the interactive compilation user story concept onto the KIELER SCCharts tool.

This is shown in Figure 4.2.3: The textual modeling takes place in the Textual Entry window, which here is the SCT editor where models are created or modified. The modeled visual diagram for the textual model is shown in the centered Visual Browsing window on the left side. This is a transient view [SSvH13] for the model, which is basically updated on every model change. The central part is the KIELER Compiler (KiCo) itself. It is an implementation of an interactive white box SLIC grounded model-based compiler. It takes 1. the model from the textual SCT editor as an input and 2. it allows the user to select features/transformations by means of a GUI front end which is shown at the bottom as the Interactive Compilation Control window. The compiler takes these two inputs and processes the input model according to the interactive compiler selection producing either an output intermediate result or the final (possibly runnable) target code. Both can be visually inspected in the Visual Browsing window on the right side next to the diagram of the original model.

4. Interactive Incremental Compilation

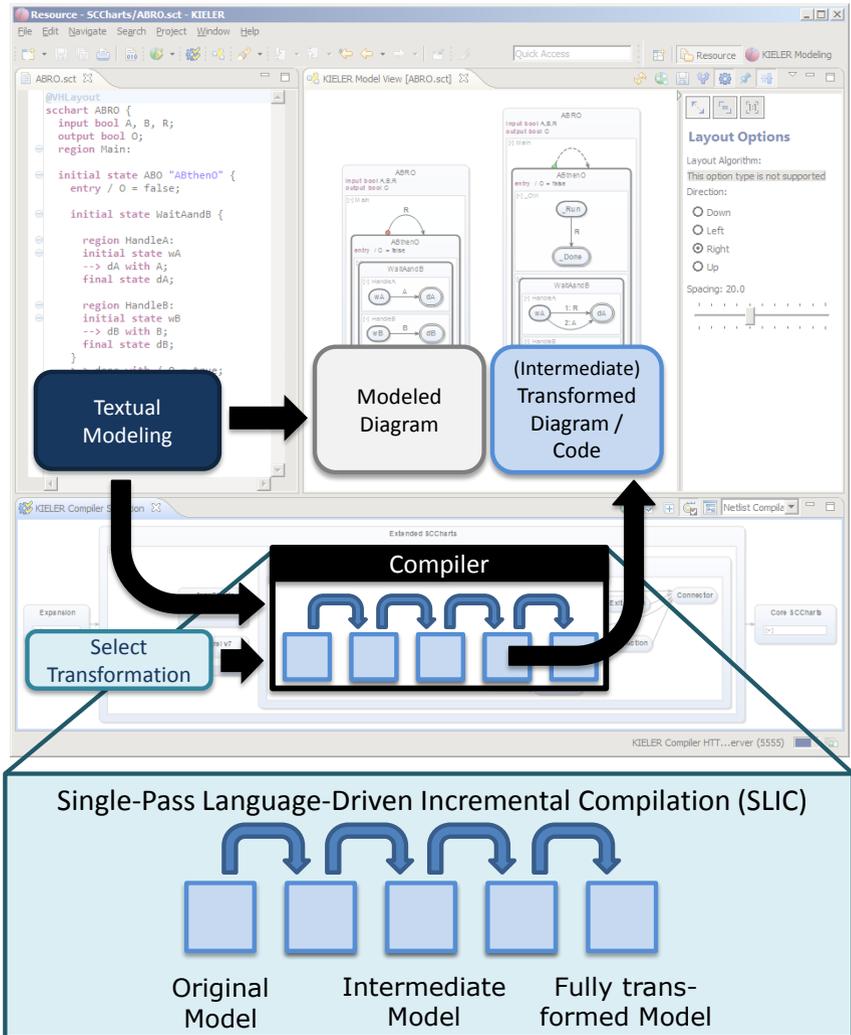


Figure 4.2.3. Interactive incremental model-based SLIC compilation user story mapped onto the KIELER SCCharts tool

Compiling SCCharts

SCCharts is a high-level synchronous modeling language. Hence, in principle there are many different possibilities to compile an SCChart into, e. g., runnable code. As Chapter 4 describes, this thesis presents an incremental compilation strategy of consecutive small-step M2M transformations. In the context of safety-critical systems this SLIC approach enables the tool smith to build a reliable and maintainable compiler. Hence, we propose a compilation strategy for SCCharts as illustrated in Figure 5.0.1. The figure is using statechart notation. The compilation splits up into a high-level synthesis part and a low-level synthesis part. The next two sections briefly describe the high-level and low-level part. Section 5.1 reflects on a SyncCharts compiler that some ideas of the current SCCharts compiler were based on and that gave inspiration for the overall SLIC approach. Section 5.2 on page 123 gives examples and details for the high-level compilation of Extended SCCharts. Section 5.3 on page 232 presents pseudocode for the high-level transformations discussed before. The following two sections, Section 5.4 on page 263 and Section 5.5 on page 273, give details for the low-level compilation followed by Section 5.6 on page 276, which discusses design choices for compilation and by Section 5.7 on page 283, which exhibits specific software and hardware targets for the SCCharts compilation. This chapter is based on previously published work [vHDM⁺13c, vHDM⁺14, MSvH14].

5.0.1 High-Level Compilation Overview

The high-level compilation, as visualized in Figure 5.0.1, starts with an Extended SCChart, i. e., an SCChart that contains one or more extended features of the SCCharts language (cf. Figure 3.2.1 on page 53).

5. Compiling SCCharts

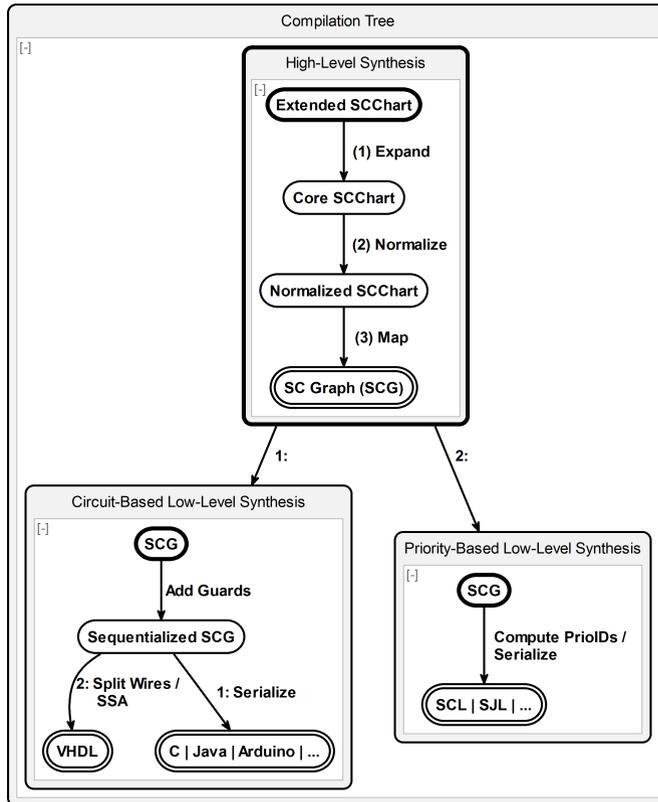


Figure 5.0.1. SCCharts compilation tree (from [MSvH14])

- (1) It first expands the extended features of an Extended SCChart utilizing the SLIC approach by applying a sequence of M2M transformations that eliminate all extended features step-by-step until the SCChart only contains core features and hence is termed Core SCChart.
- (2) Then the Core SCChart is normalized by a M2M transformation which reduces the number of patterns (cf. Figure 5.0.2). In its normalized form, the SCChart is termed Normalized SCChart.

	Region (Thread)	Superstate (Parallel)	Trigger (Conditional)	Action (Assignment)	State (Delay)
Normalized SCCharts					
SCG					

Figure 5.0.2. Five patterns for Normalized (Core) SCCharts and their direct SCG mapping (from [MSvH14])

(3) Finally, the Normalized SCChart is transformed into an SCG (cf. Section 2.7 on page 41) by mapping each pattern to an SCG element according to the mapping table shown in Figure 5.0.2. Hence, the SCG is a semantically equivalent representation that is used for convenience in down-stream compilation. In principle, one could also use the Normalized SCCharts representation down-stream.

5.0.2 Low-Level Compilation Overview

As Figure 5.0.1 suggests, we generally proposed two possible low-level synthesis paths, which both use a common high-level compilation:

1. A circuit-based option which is statically scheduled and hence can be mapped directly to hardware but can also be used for software implementations.
2. A priority-based option which is dynamically scheduled with statically calculated priorities that cannot be mapped directly to hardware but which has advantages for software implementations.

There are different possible targets for compiling SCCharts. Section 5.7 on page 283 discusses some possible targets in detail. The following para-

5. Compiling SCCharts

graphs give a first overview of such targets. Both approaches are based on a Control-Flow Graph (CFG) representation that is referred to as SCG (cf. Section 2.7 on page 41).

Circuit-Based Targets: The circuit-based approach is based on a sequentialized version of the SCG. Hence, there already exists a static schedule for the cyclic reaction computation function (cf. Figure 1.0.2 on page 3). This makes it relatively easy to produce, e. g., VHDL code. If the Sequentialized SCG is just serialized then there are several options for code generation targets. This could be plain C code that can run on any embedded device where a C cross compiler exists, it can be Java code that can run on Java platforms, or it can be code for a micro controller such as the Arduino.

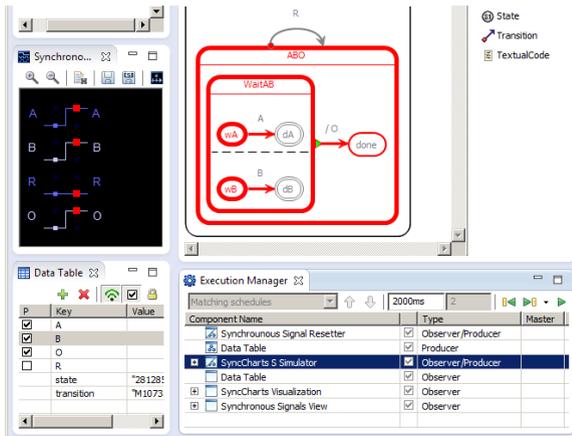
Priority-Based Targets: The priority-based approach is based on the SCG directly. Priorities can be computed that reflect the interdependencies given by sequential control-flow and communicating concurrent parts. These priorities can be used for a compile-time or a runtime scheduling. There are synchronous extensions for the C and the Java language, Lightweight Synchronous C (SCL) and Lightweight Synchronous Java (SJL), respectively. These offer a runtime scheduling based on statically computed priorities.

In the following sections, the common high-level compilation is discussed in more detail. Especially the M2M transformations and their design decisions are presented and discussed. Afterwards, the two low-level compilation paths and possible targets for SCCharts are presented. Before that, Section 5.1 recapitulates and compares two KIELER SyncCharts compiler implementations that helped the current SCCharts compiler and the general SLIC approach to evolve.

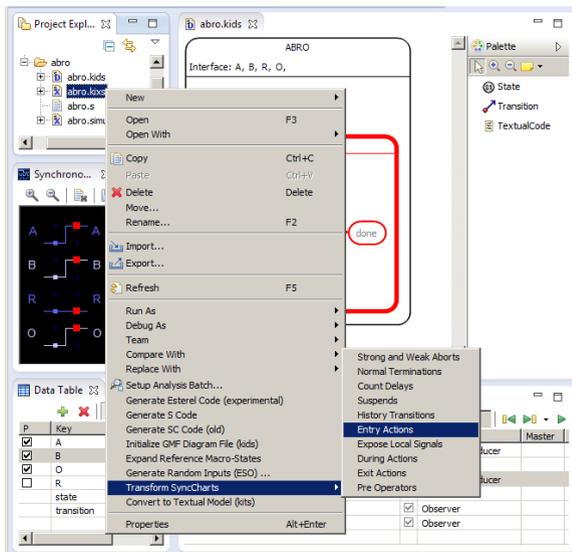
5.1 SyncCharts Compilation

Historically, the ideas of high-level transformations used for compiling Extended SCCharts largely originate from a SyncCharts compiler

5.1. SyncCharts Compilation



(a) SyncCharts-S compiler simulating ABRO



(b) "Extended SyncCharts" compiler transformations

Figure 5.1.1. The SyncCharts-S compiler of KIELER 0.8.0 release compiled SyncCharts to S intermediate code (cf. Section 5.5.1 on page 273) from which executable SC (C) or SJ (Java) could be generated.

5. Compiling SCCharts

released together with KIELER version 0.8.0¹ in 2012 (cf. Figure 5.1.1). It was a redesign and re-implementation of earlier published work from Traulsen [TAvH10] and Amende [Ame10].

The core of the newer SyncCharts compiler was not able to handle for example normal terminations, entry actions, exit actions, or history transitions directly. In order to still be able to use these “extended SyncCharts features” for modeling, SLIC-like model transformations were used, which eliminated these features beforehand. The transformations transformed the SyncCharts that were using these features into semantically equivalent SyncCharts that did not make use of these features. In essence this is the basic idea of eliminating extended features that is used in today’s interactive SLIC-based SCCharts compiler.

Figure 5.1.1a shows a running simulation for the ABRO example modeled as a SyncChart in KIELER. The SyncCharts compiler is used by a simulation component selected in the lower Execution Manager Eclipse view. This component used to call the extended SyncCharts feature transformations already in a specific order induced by dependencies denoted as `@requires` in Listing 5.1.1. The transformations could also be called individually using the context menu as depicted in Figure 5.1.1b. These transformations were already implemented using the (new) Xtend language which compiles to plain Java code.

However, the order was statically determined at development time. All transformations were called, even transformations that did not need to run because a specific feature was not present in a SyncChart. These *unnecessarily* called transformations were expected to not change the model and behave like the identity function.

Listing 5.1.1 shows a code snippet of the programmatically defined fixed order of transformations. The SyncCharts simulation component called these transformations to eliminate extended SyncCharts features first. Only after that, it called `Synccharts2S` to generate S code, an intermediate representation to facilitate the compilation process (cf. Section 5.5.1 on page 273). The transformations running before reduced the number of features that the essential lower-level core part `Synccharts2S` needs to take care of. In

¹<http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/KIELER+Pre-Release+0.8.0>

5.1. SyncCharts Compilation

```
369 // Create a new transformation object
370 SyncCharts2Simulation syncCharts2Simulation = new SyncCharts2Simulation();
371
372 // Normal Termination transitions (@requires: during actions, @before: exit actions)
373 transformedModel = (new SyncCharts2Simulation())
374     .transformNormalTermination(transformedModel);
375
376 // Count Delays now for the SC (host code) simulation
377 transformedModel = syncCharts2Simulation.transformCountDelay(transformedModel);
378
379 // Exit actions (@requires: entry actions, during actions, history)
380 transformedModel = syncCharts2Simulation.transformExitAction(transformedModel);
381
382 // History transitions (@requires: suspend)
383 transformedModel = syncCharts2Simulation.transformHistory(transformedModel);
384
385 // Suspends (non-immediate and non-delayed) (@requires: during)
386 transformedModel = syncCharts2Simulation.transformSuspend(transformedModel);
387
388 // Entry actions (@requires: during actions)
389 transformedModel = syncCharts2Simulation.transformEntryAction(transformedModel);
390
391 // During actions (@requires: none)
392 transformedModel = syncCharts2Simulation.transformDuringAction(transformedModel);
393
394 // Transform SyncChart into S code
395 Program program = (new Synccharts2S()).transform(transformedModel);
```

Listing 5.1.1. `SyncChartsSSimulationDataComponent.java`:
Snippet of simulation component applying “SyncCharts extended” feature transformations before calling the SyncCharts to S core compiler `Synccharts2S`.

comparison to the previous implementation [Ame10], the advantages were a much more modular and maintainable SyncCharts compiler that turned out to have much fewer problems with valid SyncCharts. Many valid SyncCharts were refused or incorrectly compiled by the previous SyncCharts compiler implementation.

Table 5.1.1 summarizes properties of the three different SyncCharts/SCCharts compiler variants that evolved over time. In the following, the first SyncCharts compiler implementation by Amende [Ame10] is referred to as SyncCharts2C, the more modular re-implementation (cf. Figure 5.1.1) is referred to as SyncCharts2S, and the interactive SCCharts compiler, which evolved from both, is referred to as SCCharts.

5. Compiling SCCharts

Table 5.1.1. Advantages and drawbacks of SyncCharts/SCCharts compilers, where SyncCharts2C is the first SyncCharts compiler implementation by Amende [Ame10] and Traulsen et al. [TAVH10]. SyncCharts2S is the more modular re-implementation (cf. Figure 5.1.1) and the SCCharts compiler is the interactive compiler that evolved from both of its predecessors.

	SyncCharts2C	SyncCharts2S	SCCharts
Compactness	+	-	--
Modularity	--	+	++
Fully-featured	-	+	++
Extendability	-	+	++
Maintainability	-	+	+
Visualizations	-	+	+
Interactive	-	-	+
Compilation			

Compactness: The SyncCharts2C compiler is monolithic using Xtend (1.0) and Xpand (1.0) for a more or less direct code generation from SyncCharts models. It is fairly compact. Both other compilers use Xtend (2.0) and Xpand (2.0). The SyncCharts2S compiler consists of several transformations that must run before the actual compiler is called. Additionally, the simulation visualization is decoupled and implemented in a separate transformation. Furthermore, this compiler uses the S language as an intermediate (model) representation before generating C or Java code. Hence, this compiler is less compact.

The SCCharts compiler is based on the interactive KiCo framework consisting of additional UI components to allow user interaction. Additionally, this compiler consists of a high-level SCCharts and a low-level SCG representation and also uses the S intermediate representation. It further implements a wider range of features, because SCCharts subsume all SyncCharts features but additionally have features such as (concurrent) variables, a weak suspend, or deferred transitions, not present in SyncCharts. Hence, the SCCharts compiler is not as compact compared to both of its predecessors.

5.1. SyncCharts Compilation

Modularity: As already stated, the SyncCharts2C compiler is monolithic and hence not modular. The SyncCharts2S compiler already modularizes the compilation process into separate transformations although the final SyncCharts2S transformation is still fairly monolithic compared to the even more modular SCCharts compiler.

The SCCharts compiler uses very modularly designed model transformations for compiling SCCharts features that build upon each other. The interactive KIELER Compiler (KiCo) framework emphasizes the modularity by giving the user explicit access to intermediate results using the compiler's UI for selecting certain transformations. Clearly, the SCCharts compiler is the most modular compiler out of all three.

Fully-featured: The SyncCharts2C compiler already compiled a wide range of SyncCharts features. However, it does not (fully) support all SyncCharts features as for example exit actions, valued signals, reference states, or history transitions. The re-implemented SyncCharts2S compiler works with valued signals and compiles exit actions and history transitions. Since it is based on separate transformations, it could have been extended also to consider, e. g., reference states as well.

The SCCharts compiler supports all Extended SCCharts features shown in Figure 3.2.1 on page 53 and additionally supports, e. g., reference states. Hence, it supports the highest number of features compared to both other compilers.

Extendability: The SyncCharts2C compiler more or less consists of a single transformation that is called for a SyncCharts model. This makes it hard to extend this compiler without further infrastructure. Since this compiler outputs C code directly and no intermediate representation was chosen, it is not easily possible to add another back end for a different language. In fact, the monolithic compiler must be modified directly. This is also necessary in order to change or extend simulation visualizations, which are built-in into the compiler. In contrast, the SyncCharts2S compiler uses S as an intermediate language. C and Java were already added as back ends. Other back ends could be easily added similarly. The SyncCharts2S compiler uses a fixed sequence of

5. Compiling SCCharts

independently implemented model transformations. New ones for other features can be integrated with minimal changes to the compiler. The SyncCharts2S compiler is more extendable than its predecessor.

The SCCharts compiler builds upon the KiCo infrastructure that is inherently extensible. It consists of independent transformations where no static order is defined at development time but a dynamic order based on the usage of features, transformation interdependencies, and user interaction. The SCCharts compiler is meant to be extended. It already was extended during projects that used the compiler (cf. Chapter 7).

Maintainability: Due to its monolithic structure, the SyncCharts2C compiler turned out to be hardly maintainable. While developing its successor, various faulty behaviors were observed. The SyncCharts2S compiler turned out to be much more maintainable than the SyncCharts2C compiler. The SCCharts compiler also suffered from a number of missing and faulty functionality as a project (cf. Chapter 7) exposed. Fortunately, the modular design, similarly to its predecessor, helped significantly when extending or fixing these issues. Summarizing, the SCCharts compiler has also been proven to be maintainable.

Visualizations: Visualizations of active states and taken transitions are built into the SyncCharts2C compiler. This makes it harder to add new, other visualizations or modify the existing surrounding infrastructure that interacts with the running code equipped with these extra outputs. The outputs are weaved into the monolithic transformation. The SyncCharts2S compiler per default has no such built-in visualizations but a separate visualization transformation is used to modify the SyncChart in a way such that it outputs information about active states and taken transitions. This is done as a SyncCharts M2M transformation that can be re-used for arbitrary SyncCharts compilers. This idea was borrowed from the KIELER Esterel integration that also comes with a visualization of active statements based on a M2M transformation (cf. Section 8.2 on page 393). The SCCharts compiler can easily re-use the same simulation visualization transformation because of its extensible nature. Other visualizations could be possibly added in the future and modifications to the simulation visualization only affect this certain transformation.

5.2. High-Level Compilation

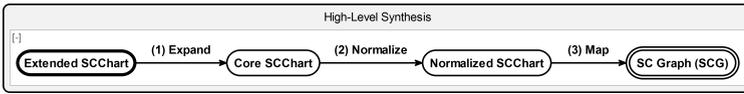


Figure 5.1.2. SCCharts compilation tree (cf. Figure 5.0.1 on page 114): High-Level Synthesis part

Interactive Compilation: Both SyncCharts compilers were not meant to respect any user interaction. The only compiler that is able to react differently and adapted to a certain user selection of transformations is the interactive KiCo-based SCCharts compiler.

It is noteworthy that both, the SyncCharts2C and the SyncCharts2S compiler, produce code by following a Priority-Based Synthesis path (cf. Figure 5.0.1 on page 114). However, because these compilers were the ancestors of today's SCCharts compiler, the implementation were not yet based on the SCG and also not yet using the exact same High-Level Synthesis. However, future and already planned SCCharts/SCG compilers following the Priority-Based Synthesis will (re-)use the common High-Level Synthesis as depicted in Figure 5.0.1 on page 114. The following section will discuss this High-Level Synthesis path in detail. It is already implemented and used by today's Circuit-Based SCCharts2C compiler.

5.2 High-Level Compilation

In the upper part, Figure 5.0.1 on page 114 shows the steps of the high-level compilation for SCCharts, which are once again depicted in Figure 5.1.2. These steps are commonly used for both low-level synthesis paths, i. e., the circuit-based and the priority-based synthesis. The common high-level compilation consists of (1) expanding Extended SCCharts into Core SCCharts, (2) normalizing Core SCCharts to Normalized SCCharts, and finally (3) constructing an SCG by mapping normalized core feature patterns to SCG elements. The high-level transformation of expanding Extended SCCharts involves numerous transformations. This section will use ALDO as introduced in Section 3.1 on page 48 (cf. Figure 3.1.1 on page 49) as a

5. Compiling SCCharts

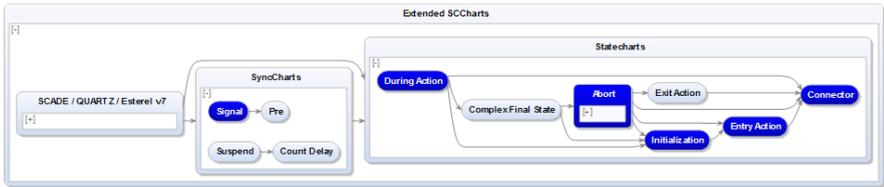


Figure 5.2.1. SCCharts high-level compilation interactive KiCo compiler selection in KIELER SCCharts tooling: The selected (darker blue) transformations are the ones that are necessary to compile the ALDO Extended SCChart to an ALDO Core SCChart as shown by Figure 3.2.2 on page 55.

primary example to illustrate the high-level synthesis path. Additionally, it will give examples and details for latest high-level M2M transformations ideas and implementations, which mostly have not or hardly been changed since earlier publications [vHDM⁺13c, MSvH14]. There exist additional features like reference states, array support, or for constructs which were added after a while proving extendability (cf. Section 7.2.2 on page 370).

5.2.1 Compiling ALDO to Core SCCharts

The ALDO example was introduced in Section 3.1. As Figure 3.2.2 on page 55 showed, the same behavior can be expressed semantically equivalently in Extended SCCharts and Core SCCharts. This section now gives details on how to compile the Extended SCCharts variant of ALDO to the Core SCCharts variant using several *incrementally* applied M2M transformations.

The ALDO SCChart directly contains the following extended features:

- ▶ Signals: Signal D, declared at the root state and used in the during action of state WaitL, is an extended feature.
- ▶ During actions: The during action that emits signal D is itself an extended feature.
- ▶ Aborts: The strong abort transition from state WaitL to state DoneL is an extended feature.
- ▶ Initializations: The initialization of the boolean output variable O that sets O to false is an extended feature.

5.2. High-Level Compilation

When compiling Extended SCCharts to Core SCCharts, all extended features need to be eliminated. A certain SLIC order that is derived from transformation interdependencies ensures that each transformation must only be applied at most once to the model. Section 4.1.4 on page 99 described in detail how such a SLIC order for SCCharts compilation is developed. This chapter and the following sections will focus on the details of single model transformations and their incremental application.

As described earlier, during application of the SLIC transformation chain there may be extended features produced by transformations. This could be extended features that were not used in the model a priori but of course also need to be expanded by other (following) transformations. Such extended features are neither visible in the Extended SCChart nor in the Core SCChart but only in intermediate models during compilation.

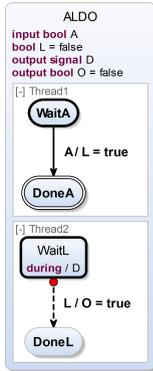
For ALDO such features are:

- ▶ Entry actions: ALDO itself does not contain any entry actions. However, as stated in Figure 4.1.5 on page 99, the transformation for expanding initializations will produce entry actions. Hence, entry actions appear during compilation of ALDO and additionally must be eliminated.
- ▶ Connectors: ALDO itself does also not contain connectors. However, as stated in Figure 4.1.5 on page 99, the transformation for expanding entry actions will possibly produce connectors. Hence, connectors might appear during compilation of ALDO and additionally must be eliminated.

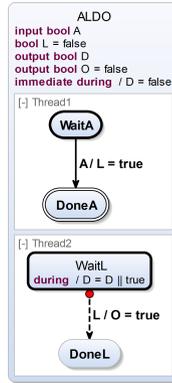
Figure 5.2.1 shows all SCCharts transformations as visible in the KIELER SCCharts tooling. The depicted interactive compiler selection is only a simplified view to the transformation dependencies grouped by hierarchy as visible for the end-user. Note that there are different views proposed and available also for developers. Details on these views are discussed in Section 6.5.6 on page 344.

Figure 5.2.1 also highlights the specific transformations that are necessary to transform ALDO from Extended SCCharts to Core SCCharts. The SLIC order is visible already through the layout of the compiler selection graph. Note that this graph re-uses the SCCharts's statechart notation but itself is no SCChart. The "states" visible in this graph represent the feature transformations and the "transitions" represent their interdependencies.

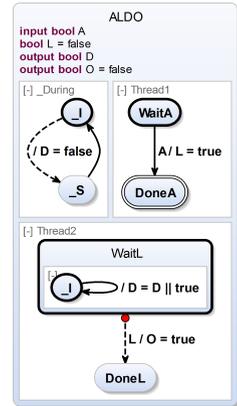
5. Compiling SCCharts



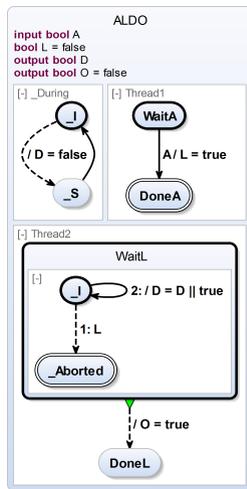
(a) Extended SCChart



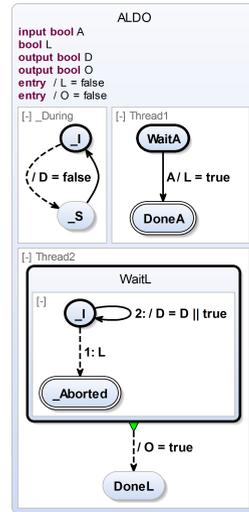
(b) After expanding signals



(c) After expanding during actions



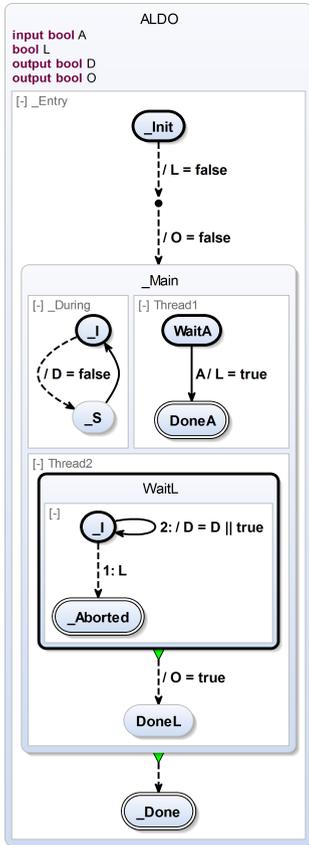
(d) After expanding aborts



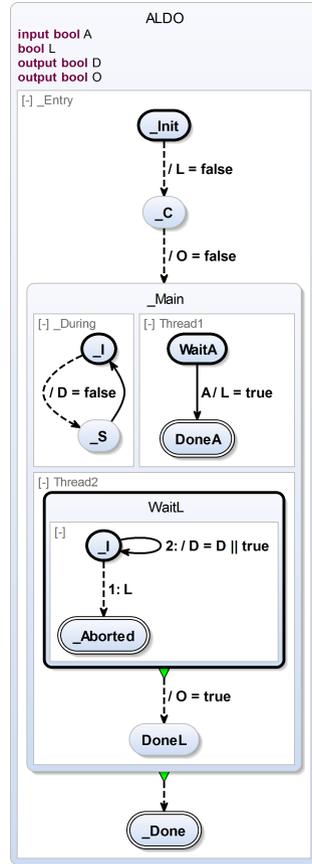
(e) After expanding initializations

Figure 5.2.2. High-level compilation from Extended ALDO to Core ALDO part I of II

5.2. High-Level Compilation



(a) After expanding entry actions



(b) Core SCCharts, after expanding connectors

Figure 5.2.3. High-level compilation from Extended ALDO to Core ALDO part II of II

5. Compiling SCCharts

The following paragraphs present details on how ALDO in the Extended SCCharts version is compiled incrementally to an ALDO in Core SCCharts. Figure 5.2.2 and Figure 5.2.3 together expose all intermediate models that result during compilation after applying each of the selected (dark blue) M2M transformations shown in Figure 5.2.1.

Expanding Signals: (a)→(b)

Starting with the ALDO Extended SCChart of Figure 5.2.2a, the first transformation w. r. t. the SLIC schedule is the expansion of signals. The only signal to transform is declared in the root state. It is the output signal D . The signal is used within state WaitL in a during action that emits the signal in each synchronous tick, while WaitL is active and not aborted. When transforming pure signals that only have a presence status and no value, these are represented as boolean variables where, for each tick, a true value corresponds to the presence of the original signal and a false value corresponds to the absence of the original signal.

Figure 5.2.2b shows the intermediate model after performing the signal expansion. The output signal D is now represented by a boolean output variable D . An immediate during action is added to the declaring state. It sets D to false at the beginning of each tick using an absolute write. In the scope of D all emits have been replaced by relative writes with the value true. Hence, in this example, the emit of the during action was replaced.

Expanding During Actions: (b)→(c)

The intermediate compilation result shown in Figure 5.2.2b after the signal expansion is now the input for the next transformation in SLIC order, namely the during action transformation. Figure 5.2.2c shows the result of the during action expansion which eliminates both during actions, the one that resets D to false at the beginning of each tick, and the one that sets D to true whenever WaitL is the active state. A during action is roughly represented by a concurrent cyclic construct that repeats the action part of the during action whenever its trigger holds. In ALDO, a region is added, one for each during action. The immediate during action translates into two

states in order to respect the WTO principle (cf. Section 3.2.8 on page 67) to not duplicate the trigger and action of the during action for the immediate and the delayed part. Since the other during action is not immediate, it can be translated into a single state with a cyclic delayed self-loop transition.

Expanding Aborts: (c)→(d)

Transforming aborts in general is a more difficult part. For ALDO, it is relatively simple to do that. However, it already nicely illustrates the general procedure. Figure 5.2.2d shows the intermediate model after expanding abort transitions from superstates. Note that there is only one such transition, i. e., the strong abort from WaitL to DoneL triggered by L. The strong abort means that in the tick when L is true and the transition is taken, no internal behavior from WaitL is allowed any more.

In the translation of aborts, priorities ensure that this preemption takes place. A new final state `_Aborted` is created and the abort transition becomes a termination transition. The action for the transition is not changed. Note that the new transition from `_I` to `_Abort` is immediate and has the highest priority 1. This means whenever L is true in a tick, *first* this transition is tested and possibly taken. Only if this transition cannot be taken because L is known to be absent, *then* the other self-loop transition may be taken, which sets D to true.

Expanding Initializations: (d)→(e)

After expanding aborts (cf. Figure 5.2.2d), the expansion of initializations is in order according to SLIC. There are two initializations that need to be eliminated in ALDO. These are `L = false` and `O = false`. Initializations are part of the declaration and take place when the declaring state is entered. Hence, they can be transformed into entry actions.

Figure 5.2.2e depicts the result after applying the initialization transformation. Both initializations are simply replaced by entry actions that set both variables to false whenever the declaring state is entered. The declaring state here is the SCChart ALDO itself and ALDO is entered when the execution of the SCChart begins in the initial tick.

5. Compiling SCCharts

Expanding Entry Actions: (e)→(f)

ALDO did not have any entry actions before but due to the last transformation it now has two entry actions that are extended features and need to be expanded. This is done in the next transformation and the result after applying the entry action transformation is shown in Figure 5.2.3a. Expanding entry actions typically results in a new hierarchy layer (and state) that encapsulates all behavior of the state that declared the entry action (cf. Section 5.2.5 on page 137). For ALDO this new state is `_Main`. The entry actions are translated into immediate transitions targeting this new state while respecting the order of the entry actions. If there is more than one entry action, entry actions get concatenated by connector states. An auxiliary initial state may be necessary if the target was the initial state or root state before. In ALDO, both entry actions for L and O are translated to immediate transitions in the same order in which the entry actions were declared in, concatenated by a new connector state.

Expanding Connectors: (f)→(g)

A connector state is an Extended SCCharts feature. Connectors were not part of ALDO a priori. However, due to the last transformation, which produced a connector, the connector transformation needs to be applied next. Further note that the entry action transformation not necessarily will produce connector states even if entry actions are present in the intermediate model. Connector states will only be created if there is more than one entry action for a state within the model. Nevertheless, according to SLIC order, connector states are expanded last because various other transformations may also produce them.

Figure 5.2.3b shows the result of the connector state expansion transformation. This is the simplest transformation. It replaces the connector state by a normal state, here `_C`. Since outgoing transitions of a connector node are immediate implicitly, this transformation has to take care that all outgoing transformations of the new ordinary state are made immediate explicitly (see discussion in Section 3.2.8 on page 67). After applying the connector transformation, ALDO as in Figure 5.2.3b finally not contains any extended features any more and hence can be termed a Core SCChart now.

Optimization vs. Complexity Trade-Off

Note that when applying the entry action transformation (cf. Figure 5.2.3a), an additional new final state `_Done` is created with a termination transition from state `_Main`. This is superfluous for the case of ALDO because ALDO can never terminate. The reason is that the state `DoneL` is not a final state so there is at least one region (`Thread2`) without a final state.

This illustrates the general need for optimizations after and during transformations. A number of optimizations are already encoded in the transformations itself using case differentiation. However, there always is a trade-off not to increase complexity above a certain level when extending transformations like this (see discussion in Section 4.1.6 on page 103).

5.2.2 Compiling ALDO to SCG

Consider the High-Level Synthesis shown in Figure 5.0.1 on page 114 again. It consists of (1) expanding Extended SCCharts to Core SCCharts, (2) normalizing Core SCCharts to Normalized (Core) SCCharts, and (3) constructing the sequential control-flow graph as an intermediate representation, the SCG. Expanding all extended features (1) of ALDO to retrieve the semantically equivalent ALDO Core SCChart was described in the previous section.

This section will briefly show how to further compile the ALDO Core SCChart to (2) a normalized form termed *Normalized (Core) SCCharts*. Afterwards, this section will show how to construct the SCG for ALDO from the Normalized SCChart.

Figure 5.2.4 shows both, the Normalized SCChart in Figure 5.2.4a and the SCG in Figure 5.2.4b.

Normalization

Figure 5.2.4a shows how the Core SCChart of ALDO shown in Figure 5.2.3b can be represented in Core SCCharts but only using the five basic patterns introduced in Figure 5.0.2 on page 115. Details of the normalization are addressed in Section 5.2.19 on page 221.

Normalization makes the surface and depth of a state explicit as well as the priorities of transitions. The surface is the part of immediately

5. Compiling SCCharts

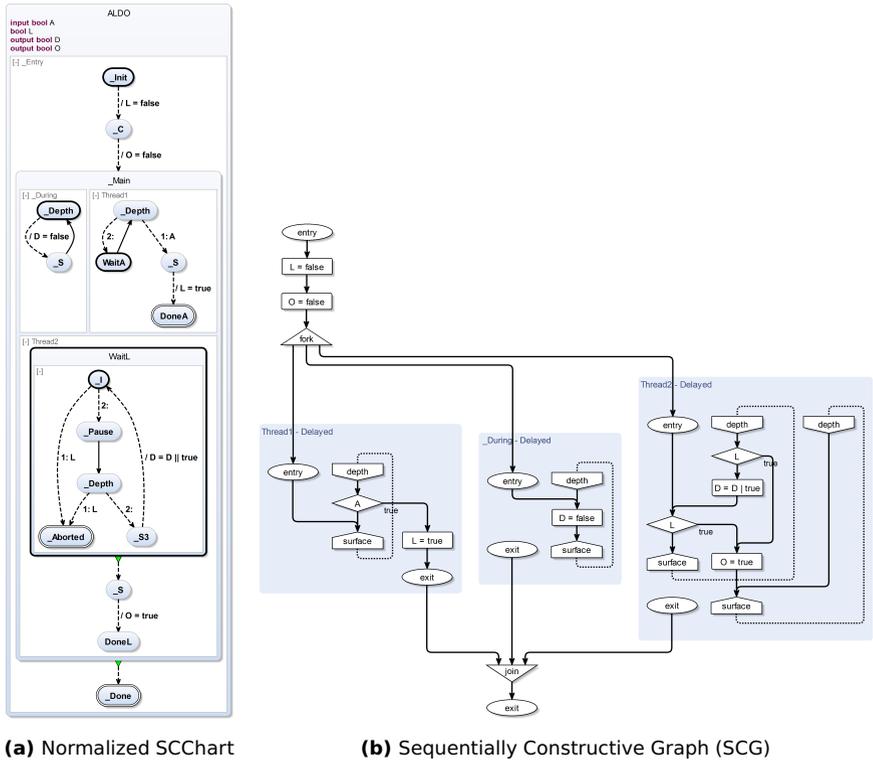


Figure 5.2.4. High-level compilation from Normalized ALDO to ALDO SCG

checked transitions when a state is entered and the depth starts after the pausing state (tick boundary). The pausing state is a state without outgoing immediate transitions but with one delayed transition without any trigger or effect such that it is taken in the next tick after the pausing state was entered. Normalization also makes the default behavior of a state visible if no outgoing transition can be taken. In this case, we end up in the pausing state and start over to check outgoing transition triggers in the next tick (in the order of their priorities).

5.2. High-Level Compilation

Figure 5.2.3b shows state `_Main` that contains three regions in which control can rest after `_Main` is entered. The first region `_During` does not need to be modified for normalization because it already is normalized. It contains two of the five patterns, namely an Action/Assignment including state `_Depth` with the assignment action `D = false` and a State/Delay pattern including state `_S` with the outgoing delayed transition.

In contrast, region `_Thread1` needs normalization. The normalized SCChart is shown in Figure 5.2.4a. The waiting in state `WaitA` is made explicit in the normalized version using the State/Delay pattern (the pausing state) to an auxiliary state `_Depth` and a Trigger/Conditional pattern that in its else branch transitions back to `WaitA`. The if branch of the Trigger/Conditional pattern has the original trigger `A` and leads to an Action/Assignment pattern with the original assignment action `L = true`. `_S` is an auxiliary state.

The region of inner state `WaitL` is normalized accordingly.

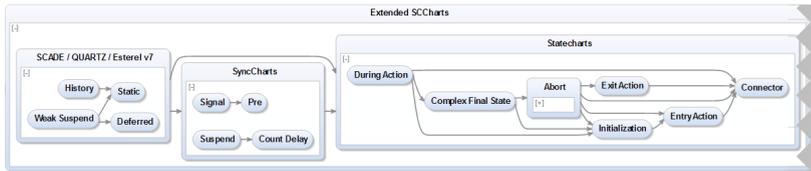
SCG Construction

Figure 5.2.4b shows how the SCG for ALDO can be constructed by mapping each of the five basic patterns to SCG constructs according to Figure 5.0.2 on page 115. Details on how to construct an SCG are addressed in Section 5.2.20 on page 231.

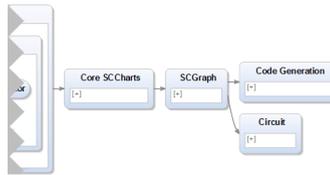
The mapping from normalized Core SCCharts patterns to SCG elements is straight forward as illustrated in Figure 5.0.2 on page 115. E. g., the sequentially ordered Action patterns `L = false` and `O = false` are transformed into SCG Assignment elements and connected respecting the original ordering. An SCG entry and exit node represents the control-flow within a region. A superstate with internal concurrency is transformed into a fork to the corresponding entry nodes and into a termination transition to a join from the exit nodes. Note that not all exit nodes need to be reachable in case of regions without final states that cannot terminate (cf. region `_During` and region `_Thread2`). Note that the SCG is optimized in the sense that it omits forks of single-region superstates like state `WaitL`.

The previous two sections demonstrated the high-level compilation path for incrementally applying M2M transformations to the ALDO example in order to compile ALDO from Extended SCCharts to its SCG representation. In

5. Compiling SCCharts



(a) Left part



(b) Right part

Figure 5.2.5. SCCharts high-level compilation interactive KiCo compiler selection in KIELER SCCharts tooling showing all high-level transformations

the following, details on these high-level compilation M2M transformations are given including an intuition on how to apply these transformations to generic SCCharts.

5.2.3 Transformation Dependencies

Figure 5.2.5 shows all high-level feature transformations that can be selected interactively by the user of the KIELER SCCharts tooling in the KiCo compiler selection. The feature expansion transformations are grouped into Extended SCCharts features and Core SCCharts features. The Extended SCCharts features are further sub-grouped into Statecharts, SyncCharts, and SCADE / QUARTZ / Esterel v7 features.

The arrows depict the interdependencies between transformations. In order to prevent inter-level transitions, transitions in the KiCo compiler selection are lifted to the highest common hierarchy layer necessary w. r. t. the connected source and target transformation. This simplifies the view for the end user but abstracts from concrete interdependencies that actually

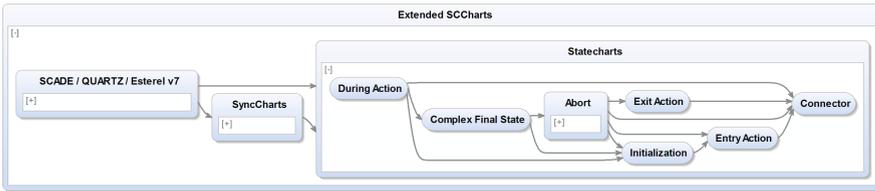


Figure 5.2.6. Statecharts feature transformations in KiCo compiler selection: Connector, Entry Action, Exit Action, Initialization, Abort, Complex Final State, and During Action (see also Figure 5.2.5)

exist as shown in Figure 4.1.5 on page 99 and Figure 4.1.6 on page 101.

Although the detailed transformation dependencies may be hidden, the SLIC order for applying transformations during compilation is still visible in the simplified view of the tool (cf. Figure 6.5.7 on page 346). In the following sections, this order will guide through a more detailed view on transformations from the bottom up, starting with the Connector transformation as part of the Statecharts feature group (see Figure 5.2.6).

5.2.4 Connector

The *connector* transformation (cf. pseudocode in Section 5.3.1 on page 233) is perhaps the simplest transformation. A connector example is given in Figure 5.2.7a. There are four simple states S_1 , S_2 , S_3 , and S_4 connected by various immediate and non-immediate transitions. Some transitions have a trigger and an action.

In this example and in the following examples of this chapter, triggers T_i are placeholders for arbitrary complex triggers that evaluate to true or false and boolean output actions A_i are placeholders for arbitrary actions such as signal emissions or variable assignments.

In the Connector example of Figure 5.2.7a, a connector is used to split transitions from S_1 to S_3 , from S_1 to S_4 , from S_2 to S_3 , and from S_2 to S_4 by re-using common triggers and actions. Recall that control can never rest within a connector. Hence, the *default transition* from the connector to state S_4 must not have a trigger and can be taken when all other triggers of

5. Compiling SCCharts

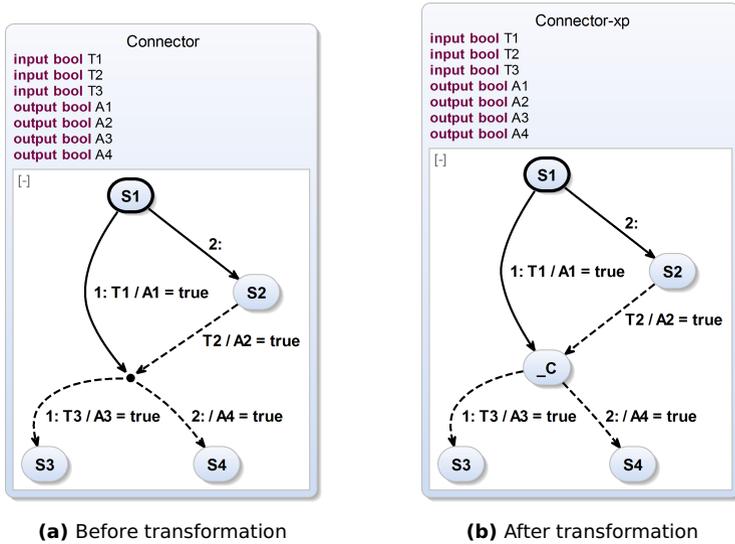


Figure 5.2.7. Connector feature expansion transformation

outgoing transitions of the connector may evaluate to false. Additionally, all of these outgoing transitions must be immediate and hence are immediate implicitly. Here “implicitly” means that these transitions always are interpreted as immediate transitions even if the specific immediate-flag for transition w.r.t. the meta model is not set explicitly to true (see discussion in Section 3.2.8 on page 67).

When transforming a connector state, it is replaced by a simple state. Additionally, all implicit outgoing immediate transitions are changed to be explicit immediate. The expanded version of the connector example, i. e., the connector example after applying the connector transformation is shown in Figure 5.2.7b. The extended feature *connector* has been successfully eliminated by this transformation. Note that the resulting simple state *_C* is a *transient state*, i. e., a state that is entered and left in the same tick. Control will never rest in a transient state.

5.2. High-Level Compilation

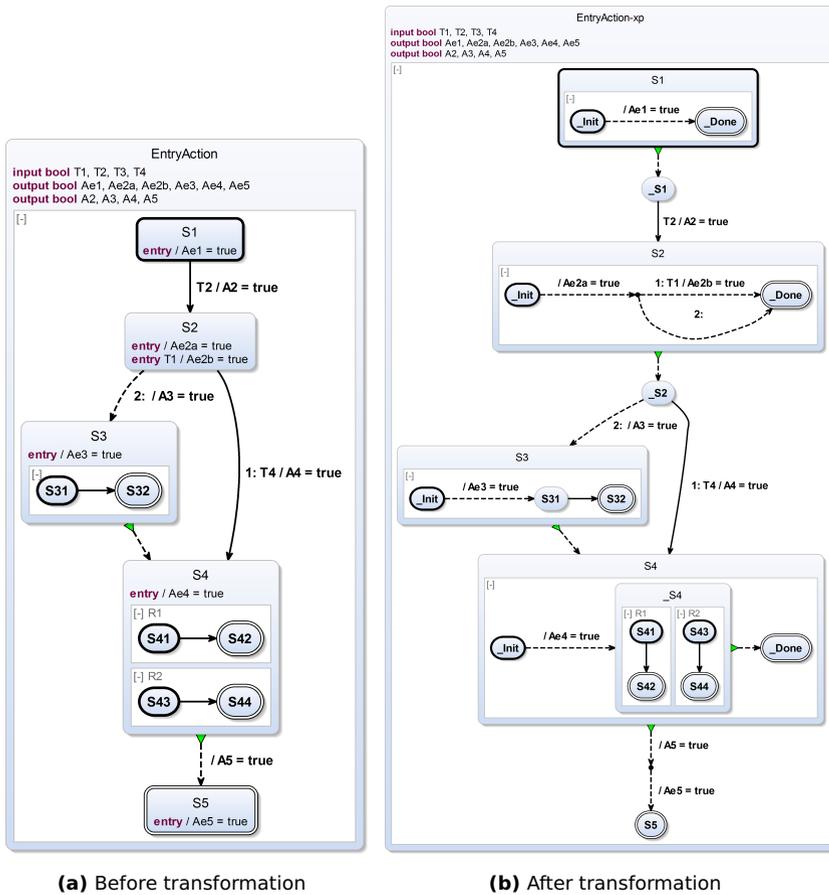


Figure 5.2.8. Entry feature expansion transformation

5.2.5 Entry Action

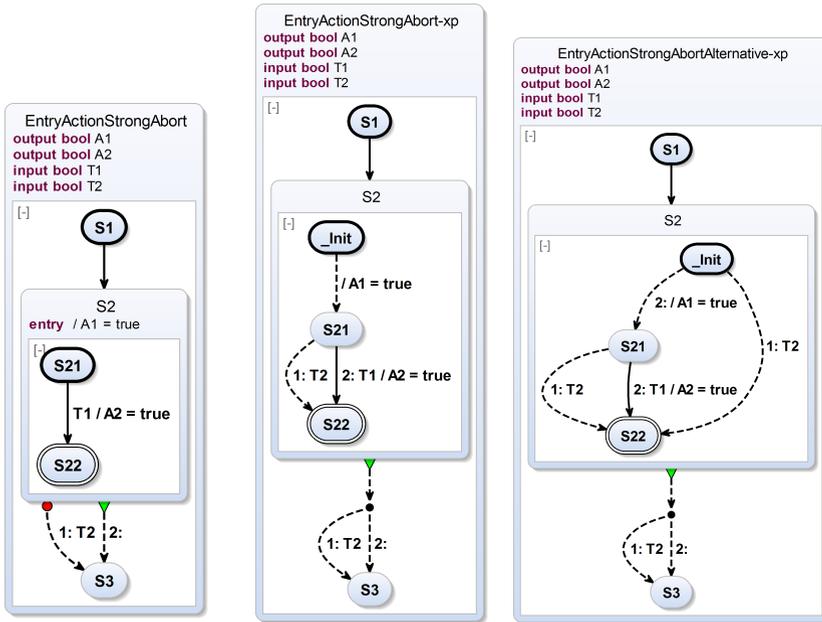
When a state S has an associated *entry action* A (cf. pseudocode in Section 5.3.2 on page 235), then A should be performed whenever S is entered,

5. Compiling SCCharts

before any internals of S are executed. Additionally, A may refer to locally declared variables in the scope of S . If multiple entry actions are present, they are performed in the declared sequential order. This differs from during actions, which are performed concurrently to each other because entry actions (like exit actions) can be clearly ordered. The rationale of this design decision is the effort to avoid unnecessary concurrency on the one hand and to respect the modeler's specified sequential order on the other hand. A is performed even in case S is immediately left again, including leaving through a strong abort. Thus, the entry action transformation is performed after the abort transformation w. r. t. the SLIC order.

The EntryAction example shown in Figure 5.2.8 illustrates the different use-cases of entry actions. $S1$ is an initial state associated with an unconditional entry action $Ae1 = true$ and an outgoing transition triggered by $T2$. The entry action gets transformed into a refinement of $S1$, i. e., $S1$ becomes a superstate with one (explicit) internal region. That region immediately executes $Ae1 = true$ and terminates afterwards. The termination transition out of $S1$ then transfers to a new state $_S1$, which then waits for the trigger $T2$, before proceeding further to state $S2$. Hence, $_S1$ has taken the role of $S1$ which is waiting for the trigger $T2$ in this example. State $S2$ is associated with two entry actions: An unconditional entry action $Ae2a = true$ and another conditional entry action $Ae2b = true$ that is only performed if $T1$ holds. This also gets transformed into a refinement of $S2$ that sequentially performs both entry actions and then, analogously to $S1$, transfers to a new state $_S2$. Note that for the conditional entry action, the else branch default transition is mandatory. $S3$ is a superstate with one internal region. In this case, a fresh initial state $_Init$ is introduced that immediately transitions to the original initial state $S31$ and performs the entry action $Ae3 = true$. Superstate $S4$ has multiple internal regions. These get encapsulated into a new superstate $_S4$, and analogously to state $S3$, the action $Ae4 = true$ gets executed as an action that is immediately performed on a transition originating in the new initial state $_Init$ leading to $S4$. To allow the termination of $S4$, an auxiliary final state $_Done$ is added that gets reached whenever $S4$ terminates. $S5$ is a final state, which may also have an associated entry action. In this example it is $Ae5 = true$. As final states cannot have any internal behavior, as discussed earlier, the transformation of their entry

5.2. High-Level Compilation



(a) Before transformation (b) After transformation (c) After (alternative) transformation with SyncCharts semantics

Figure 5.2.9. Entry action combined with a strong abort and two semantic variations for expansion: (b) shows the current semantics for SCCharts, where the entry action is performed even when a strong abort takes place and (c) shows an alternative, SyncCharts-like semantics.

actions simply introduces a connector node, which all incoming transitions are connected to. This connector then connects to the final state with a transition segment that performs the entry action.

Referring to Local Variables: A non-trivial issue when defining the transformation is that we would like to allow entry actions to still refer to locally declared variables. This facilitates, e. g., the realization of variable

5. Compiling SCCharts

initializations (see Section 5.2.7). Hence, we cannot simply attach entry actions to incoming transitions, as these would then be outside of the scope of local variables. The transformation handles this issue by handling all entry actions within the state they are attached to. This naturally also handles the case of initial states, which do not have to be entered through an incoming transition.

Entry Actions and Strong Aborts: Another non-trivial issue is how to handle the combination of strong aborts with entry actions. As strong aborts are preemptive, one possibility to argue is that they should also preempt entry actions. A contrary view is that an entry action should always be executed when a state is entered even if it is aborted right away. SyncCharts [And96] (cf. Figure 2.3.2 on page 27) implement the first semantic choice where entry actions are not executed when a preemptive abort is true. For SCCharts we chose the second semantic variation and execute entry actions even if a strong abort is executed in the same tick for consistency with exit actions. However, to change this, entry actions would simply have to be transformed before strong aborts in the SLIC transformation order. Figure 5.2.9 shows an example of an entry action $A1 = \text{true}$ combined with a strong abort triggered by $T2$ and originating from state $S2$. In the transformed Core SCChart shown in Figure 5.2.9b the semantics is clearly visible as when entering state $S2$, the entry action $A1 = \text{true}$ is always performed. Only after this, the strong abort triggered by $T2$ may lead to the immediate leaving of state $S2$. The transformed Core SCChart, shown in Figure 5.2.9c with the SyncCharts semantics, can be retrieved when performing the entry action transformation before the abort transformation. Clearly, the entry action itself is also preempted whenever $T2$ is true upon entry of state $S2$.

5.2.6 Exit Action

When a state S has an associated *exit action* A (cf. pseudocode in Section 5.3.3 on page 236), then A should be performed whenever S is left, after any possible internals of S are executed. Additionally, A may refer to locally declared variables in the scope of S . If multiple exit actions are present, they

5.2. High-Level Compilation

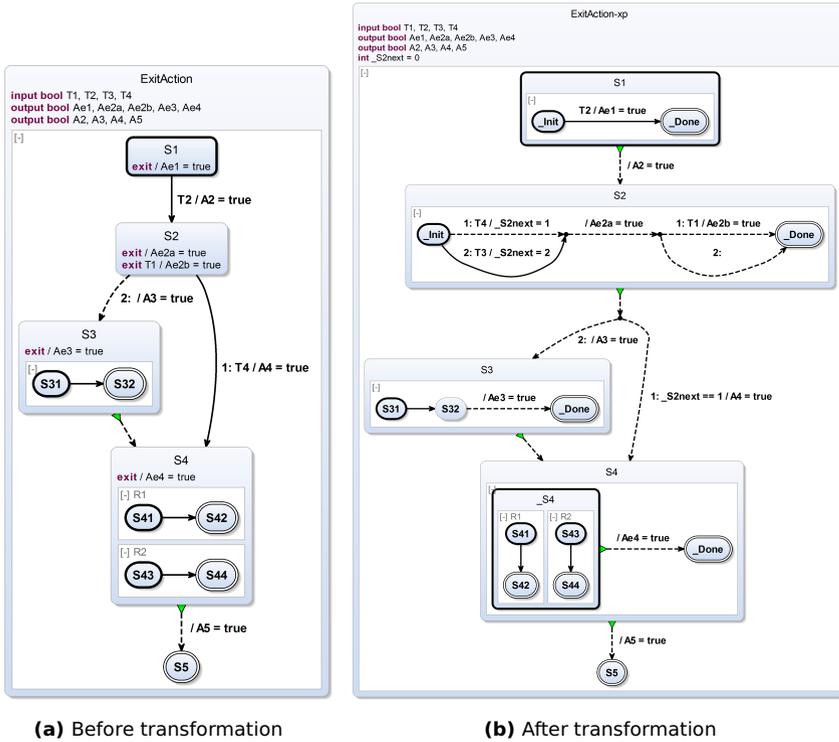


Figure 5.2.10. Exit feature expansion transformation

are performed in the declared sequential order. Likewise to entry actions, this differs from during actions, which are performed concurrently to each other, because exit actions (like entry actions) can be clearly ordered. A is performed even in case S is left with a preemptive abort. Thus, the exit action transformation is performed after the abort transformation w. r. t. the SLIC order.

The ExitAction example shown in Figure 5.2.10 illustrates the different uses-cases of exit actions. S1 is an initial state associated with an unconditional exit action Ae1 = true and an outgoing transition triggered by T2. The

5. Compiling SCCharts

exit action gets transformed into a refinement of $S1$, i.e., $S1$ becomes a superstate with one (explicit) internal region. That region executes the exit action $Ae1 = true$ only if the outgoing transition with trigger $T2$ becomes true. Note that the termination is handled internally and a termination transition transfers to the next state $S2$ now. State $S2$ is associated with two exit actions: One unconditional exit action $Ae2a = true$, followed by a conditional exit action $Ae2b = true$. This also gets transformed into a refinement of $S2$ that sequentially performs both exit actions and then, analogously to $S1$, transfers to a new connector state. This connector state is reached whenever $S2$ is left before transitioning to state $S3$ or $S4$. $S3$ is a superstate with one internal region. In this case, a new final state $_Done$ is introduced together with an immediate unconditional transition from the original final state $S32$ that performs the exit action $Ae3 = true$.

Superstate $S4$ has multiple internal regions. These get encapsulated into a new superstate $_S4$, and analogously to state $S3$, the action $Ae4 = true$ gets executed on a transition originating in the new encapsulated state $_S4$ and leading to the new final state $_Done$. $S5$ is a final state, which is not allowed to have an exit action associated with it.

Referring to Local Variables: Analogous to entry actions, a non-trivial issue when defining the transformation is that we would like to allow exit actions to refer to locally declared variables. Hence, we cannot simply attach exit actions to outgoing transitions, as these would then be outside of the scope of local variables. The transformation handles this issue by handling all exit actions within the state they are attached to.

Exit Actions and Strong Aborts: Another non-trivial issue is how to handle strong aborts in combination with exit actions, analogously to entry actions. As strong aborts are preemptive, one possibility to argue is that they should also preempt exit actions. A contrary view is that an exit action should always be executed when a state was entered before, even if it is aborted right away. SyncCharts [And96] implement the second semantic choice where exit actions are executed even when a preemptive abort takes place. Thus, SyncCharts handle entry and exit actions differently: Entry

5.2. High-Level Compilation

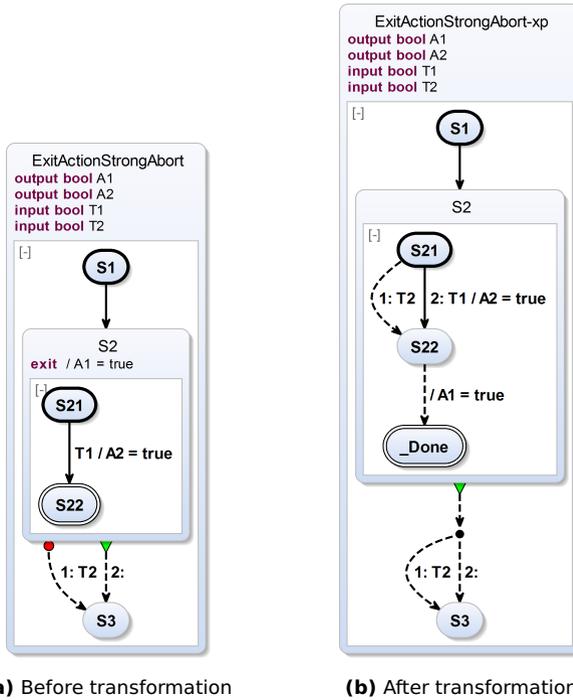


Figure 5.2.11. This example of an exit action combined with a strong abort shows the current semantics for SCCharts that is analogous to the entry action choice as shown in Figure 5.2.9 on page 139.

actions are not executed if a strong aborts holds but exit actions are executed (see Section 2.3.2 on page 27). SCCharts also implement the second semantic variation and execute exit actions even if a strong abort is executed in the same tick. To change this, exit actions would have to be transformed before strong aborts in the SLIC transformation order. The semantical choice for SCCharts was made to be consistent with entry actions and strong aborts (see above). Figure 5.2.11 shows an example of an exit action $A1 = \text{true}$ combined with a strong abort triggered by $T2$ originating from state $S2$. In

5. Compiling SCCharts



Figure 5.2.12. Initialization feature expansion transformation

the transformed Core SCChart, shown in Figure 5.2.11b, the semantics is clearly visible as when leaving state S2 the exit action A1 = true is always performed. Only after this, the state is finally left by the new final state `_Done` and the termination transition leading to state S3.

5.2.7 Initialization

Elementally, (Core) SCCharts interact using variables. These should be initialized before they are first read. This can be done using variable *initializations* (cf. pseudocode in Section 5.3.4 on page 236). In order to transform an SCChart with variable initializations (cf. Figure 5.2.12a) into an equivalent SCChart without variable initializations (cf. Figure 5.2.12b), variable initializations are transformed into *entry actions* as illustrated in Figure 5.2.12. This transformation exploits the fact that entry actions do not get moved outside of the state they are declared in. As the figure demonstrates, the order in which new, additional initialization-entry actions are added matters for the semantics (see next paragraph). Another possible solution would be to introduce another hierarchy and put the declarations and initializations outside to ensure they occur before possibly existing entry actions. But for the purpose of keeping the synthesis simpler we decided to avoid extra hierarchy if possible.

Initialization and Entry Actions: Entry actions are defined to take place after initializations. Initializations are transformed into new entry actions.

5.2. High-Level Compilation

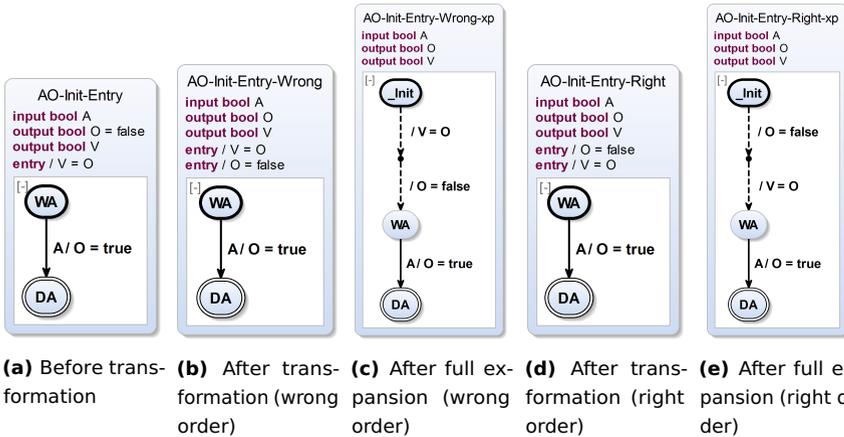


Figure 5.2.13. Initialization combined with existing entry action: The order in which new, additional initialization-entry actions are added matters for the semantics.

We add them in the same order as they are declared and put them in front of the list of possibly existing entry actions.

The reason for that is illustrated in Figure 5.2.13. This figure exposes two different possibilities of an initialization transformation applied for the original model as shown in Figure 5.2.13a.

1. Figure 5.2.13b and Figure 5.2.13c depict an initialization transformation that creates new entry actions from initializations and just adds them to the list of existing entry actions. As can be seen in the figure, this gets problematic if other entry actions refer to initialized variables. In this case, the entry action $V = O$ would require O to be initialized before it takes place. However, if the initialization of O is transformed into a new entry action that is just added to the list of existing entry actions then the initialization $O = \text{false}$ would happen after all previously existing entry actions including $V = O$. Hence, just adding new entry actions at the end of the list of existing entry actions is generally wrong for the initialization transformation.

5. Compiling SCCharts

2. Figure 5.2.13d and Figure 5.2.13e visualize the other possible initialization transformation that is realized for SCCharts. It adds all new entry actions that are transformed initializations to the list of existing entry actions but *before* them, i. e., at the beginning of the list of previously existing entry actions. Of course the order of initializations should still be preserved which is not illustrated by this example because here only one initialization is present and transformed. The rationale is that initializations should be allowed to refer to other, preceding initialized variables.

5.2.8 Abort

Abort (cf. pseudocode in Section 5.3.5 on page 239) is one of the central extended features. The abort transformation handles preemptive strong aborts and weak aborts and transforms all weak and strong abort transitions into core transitions. It further expands conditional termination transitions.

The reason is that in Core SCCharts, the only way to leave a superstate S is to take a termination transition. Hence, the task is to transform aborts such that when a superstate S is aborted due to some trigger T , all regions in S reach a final state which then allows the single termination transition to be taken. The general idea is thus to check for trigger T in all states in all regions of S and to transition to a final state when T holds.

Figure 5.2.14a shows a simple example for the abort transformation. In the state On the output D is set to true in each tick but not in the initial one. This can be aborted immediately by setting the input I to true. When I is true, the strong abort transition instantaneously 1. preempts the inner behavior of On to set D , 2. sets O to true, and 3. transitions to state Off .

The result after performing the abort transformation is shown in Figure 5.2.14b. A (core) termination transition is used to leave state On . It sets O to true. The abort is handled explicitly inside state On : If I is true then an immediate *aborting transition* from Do to a new auxiliary final state $_Aborted$ is taken. It has the highest priority because of the original strong abort. Only while I is false, the normal operation of On , which is the self-loop of Do that sets D to true, is performed.

In general, the approach adds explicit *aborting transitions* for each in-

5.2. High-Level Compilation

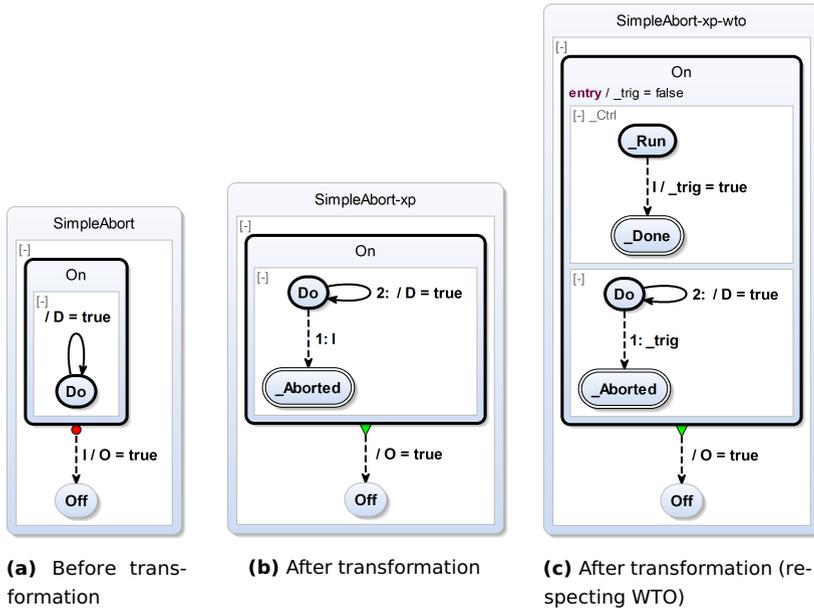


Figure 5.2.14. Simple abort example

ternal state of `On` to the auxiliary `_Abort` state. All these possibly many transitions are triggered by `l`. According to the WTO principle, it is often desirable to evaluate a trigger just once, especially if it is a complex one. For the given example this means that the trigger `l` should not be replicated to all created aborting transitions. The solution is to introduce a *control region* `_Ctrl` and an auxiliary trigger variable `_trig` as depicted in Figure 5.2.14c. `_trig` is set to true if the abort variable `l` is true. This also terminates the control region. Now, if `l` is a complex trigger then this is evaluated only once per tick. This happens in the control region.

The abort transformation looks quite simple in this example. But the abort transformation should also handle weak aborts and conditional or delayed terminations and transform them into Core SCCharts. The following

5. Compiling SCCharts

sections will discuss several corner cases and specialties, ending up with a general form of a WTO and a non-WTO variant of the abort transformation.

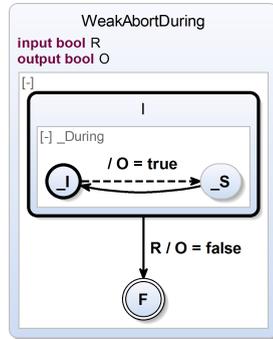


Figure 5.2.15. Weak abort of cyclic action — before transformation

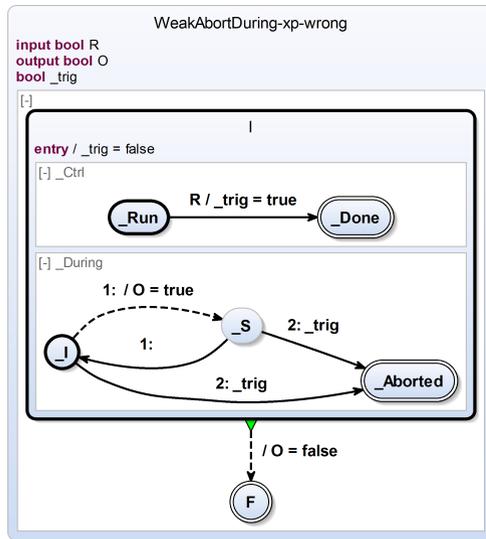


Figure 5.2.16. Weak abort of cyclic action — after transformation (wrong)

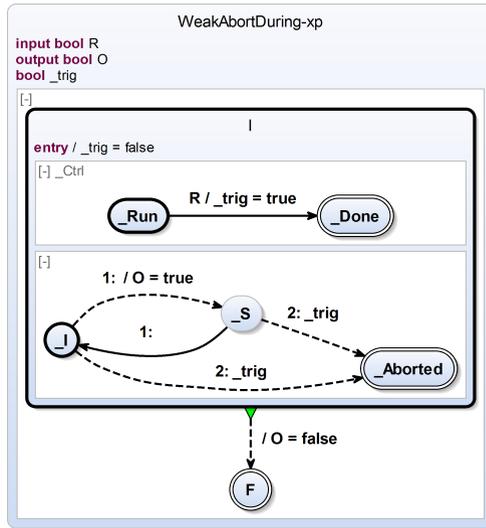


Figure 5.2.17. Weak abort of cyclic action — after correct transformation

Aborting Cyclic Behavior

For strong aborts it is not a problem to abort cyclic behavior because strong aborts lead to new aborting transitions that have a higher priority than any other existing transition, in order to preempt these other transitions and their behavior.

Weak aborts in contrast lead to aborting transitions that have a lower priority than any other existing transition. The reason is that the immediate behavior should be permitted as a so called *last will* (as explained in Section 3.2.3 on page 58). A problem can arise from cyclic behavior like the example shown in Figure 5.2.15. Transforming this model results in the expanded model of Figure 5.2.16. A problem clearly exists here because none of the `_trig` labeled new aborting transitions will ever be taken because of their lower priority. Why is the solution to make all new aborting transitions being immediate? The reason is that immediate cycles are forbidden and once the control rests (which is the consequence of forbidding immediate

5. Compiling SCCharts

cycles) in at least one of the states, the new immediate (weak) aborting transition will be taken, although it has a lower priority than any other existing transition.

Figure 5.2.17 shows the correctly transformed weak abort. If the abort occurs, clearly the internal action is permitted (setting `O` to true) but also the abort is triggered which results in taking the termination transition (sequentially setting `O` to false again).

Ease Down-Stream Complexity

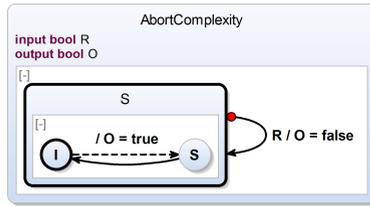
When transforming aborts, there is an auxiliary control region with reactions to triggers that originally were part of outgoing transitions. This can be seen in the example of Figure 5.2.17 where `R` is the trigger in the control region which has been the original trigger of the transformed outgoing weak abort transition. Therefore, the control region transitions are of the same type (immediate or delayed) as the originally outgoing aborts.

This can be seen in the example shown in Figure 5.2.18a where a self-loop is modeled as a *delayed* abort triggered by `R`. The transformed model respecting the WTO principle is shown in Figure 5.2.18b. The delay is captured in the control region `_Ctrl`. Hence, the explicit aborting transitions to state `_Aborted` can all be immediate.

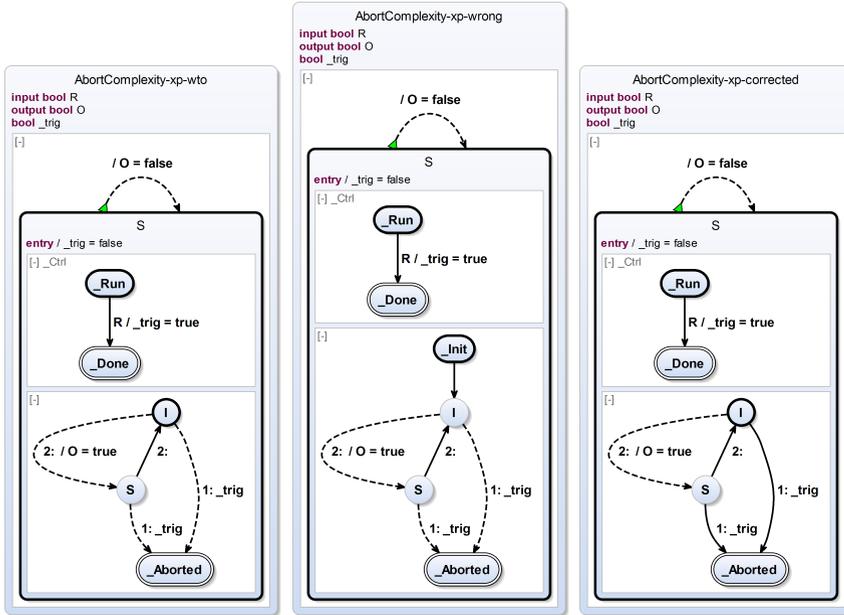
However, only if the original abort would have been immediate, the immediate aborting could ever happen. In the above example there originally was a delayed abort. Hence, it is *impossible* that the `_trig` auxiliary variable becomes true in the same tick when entering the overall state. It is possible to test immediately for `_trig`. However, we know that the control region won't set `_trig` to true in the first tick in which this state is entered. This unnecessarily complicates down-stream synthesis because the scheduling gets much more difficult with this new introduced additional but superfluous immediate test of `_trig`. This causes the region to become potentially instantaneous and would require additional analysis to resolve this in down-stream compiler stages.

Wrong Solutions: Looking at the example shown in Figure 5.2.18a, a quick solution seems to introduce a new initial node (e. g., `_InIt`) in all regions

5.2. High-Level Compilation



(a) Before transformation



(b) After original transformation (correct)

(c) After wrong transformation (incorrect initial tick)

(d) After enhanced transformation (corrected)

Figure 5.2.18. Ease down-stream compilation: Abort transformation for delayed strong aborts: The transition from I to `_Aborted` in (d) is delayed.

5. Compiling SCCharts

(except the control region) and connect it to the original initial node with a delayed transition. This approach is depicted in Figure 5.2.18c. However, this expansion is wrong whenever there is immediate behavior in a state like in this example. In this case, the immediate behavior would erroneously not take place in the tick when `AbortComplexity` is entered and `R` is true. So, this is not a proper solution.

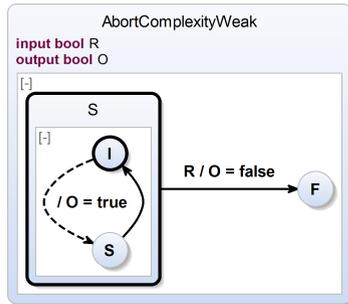
Another quick solution seems to make the termination transition a delayed termination transition in this case. This would be correct but counter-productive because the delayed termination is not a core construct and would have to be eliminated. Again, this would lead to a translation where the delay is handled in the control region which is where we have started.

Correct Solution For Strong Aborts: The correct solution for this issue is to distinguish not only strong and weak aborts for the main region (where the additional aborts lead to the `_Aborted` state) but also their immediate and delayed variant. This makes a total of four additional possible aborting transitions for aborting each state. If a transition was delayed initially, going out from state `S`, then inside the main region of `S` the resulting aborting transitions should also be delayed. This prevents the region to be become potentially instantaneous if it does not necessarily has to be and eases further down-stream compilation steps in the SCG.

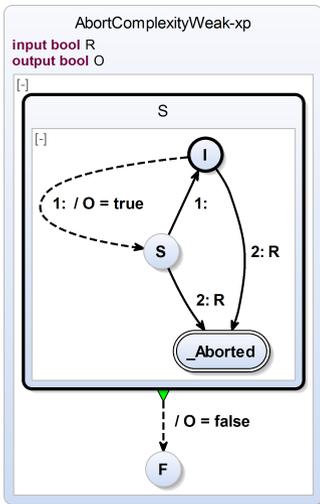
Figure 5.2.18d shows a variant where the aborting transitions now are also delayed in addition to the control transition that already was delayed before. Again, this can be done because the initial abort transition going out from `S` was delayed and not immediate. This eases the down-stream compilation because no additional superfluous immediate transitions are introduced and now both regions are guaranteed to be delayed.

Problem with Delayed Weak Aborts: The following example in Figure 5.2.19a reveals that weak aborts, in contrast to strong aborts (as just explained before), cannot be delayed aborted in general. Trying to do so in this example leads to an expanded model such as the one shown in Figure 5.2.19b. This *never* terminates because the aborting transitions have

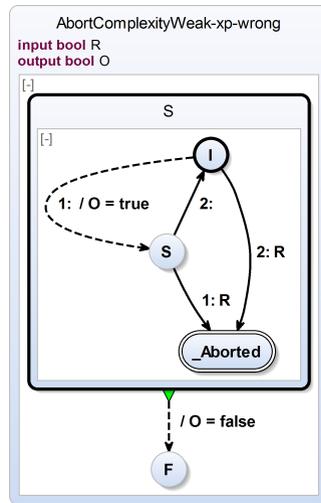
5.2. High-Level Compilation



(a) Before transformation



(b) After transformation (can never abort)

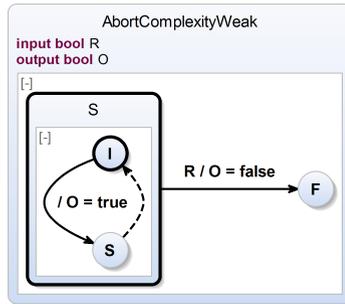


(c) After transformation (generally wrong solution which is correct in this example)

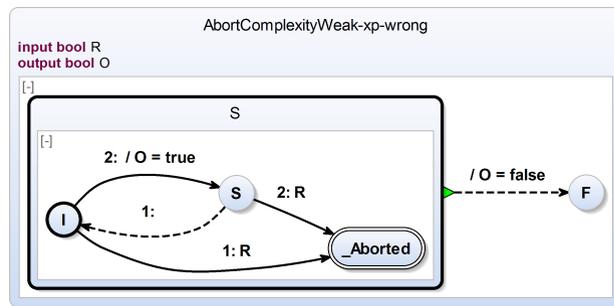
Figure 5.2.19. Abort transformation for delayed weak aborts

always the lowest priorities and therefore will never be taken. Note that the control region is omitted in this example for brevity and the problem

5. Compiling SCCharts



(a) Before transformation



(b) After transformation (revealing wrong solution)

Figure 5.2.20. Abort transformation for delayed weak aborts: Similar example, but the transition from I to S is now delayed and the other transition is immediate.

would also exist with a control region. A solution to this problem seems to be the following: Have the priority of delayed weak abort transition to be

1. lower than any immediate transition in order to allow the “last will”, i. e., further immediate internal behavior but
2. higher than any delayed transition in order to allow aborts.

5.2. High-Level Compilation

This is done for the resulting SCChart shown in Figure 5.2.19c which is correct in *this* case.

However, this is (still) not a correct solution in general. If this approach is applied to the following similar example, shown in Figure 5.2.20a, then it can be seen in the transformed model (cf. Figure 5.2.20b) that the “last will”, i. e., setting output *O* to true, is not permitted to take place when starting in state *I* and in the next tick *R* is true.

Consequence: In case of delayed weak aborts there is no solution other than having still immediate aborting transition inside the main region and have the delay taking place in the control region.

All weak abort transitions in the main region must be immediate. Otherwise, they might be overruled by other transitions and would never be executed. This way, it is made sure that a weak abort is taken if control would rest otherwise (at the end of the normal immediate “last will” behavior of the main region).

Termination Elimination: One drawback mentioned earlier (cf. Section 3.2.3 on page 62) of the first proposed [vHDM⁺13c] abort transformation (cf. Figure 5.2.25) is that the “termination detection” does not distinguish between a termination due to reaching of a final state and a termination due to an explicit aborting transition, which also reaches a final state. The problem becomes noticeable if a termination is modeled with a higher priority than a weak abort. However, as discussed in Section 3.2.3 on page 60, this is allowed in SCCharts.

In the spirit of the earlier example shown in Figure 3.2.6c on page 61, the current abort transformation first eliminates terminations and replaces them by weak aborts and auxiliary termination flags. An example is given in Figure 5.2.21. The termination flag `_term` allows to distinguish whether the abort was triggered by reaching the final state via a previously exiting transition or via an auxiliary aborting transition (here the aborting transition triggered by *I*).

Further Optimizations: Reconsider the SCChart shown in Figure 3.2.6 on page 61. The current abort transformation will create a control region because of the delayed weak abort self-loop as explained earlier.

5. Compiling SCCharts

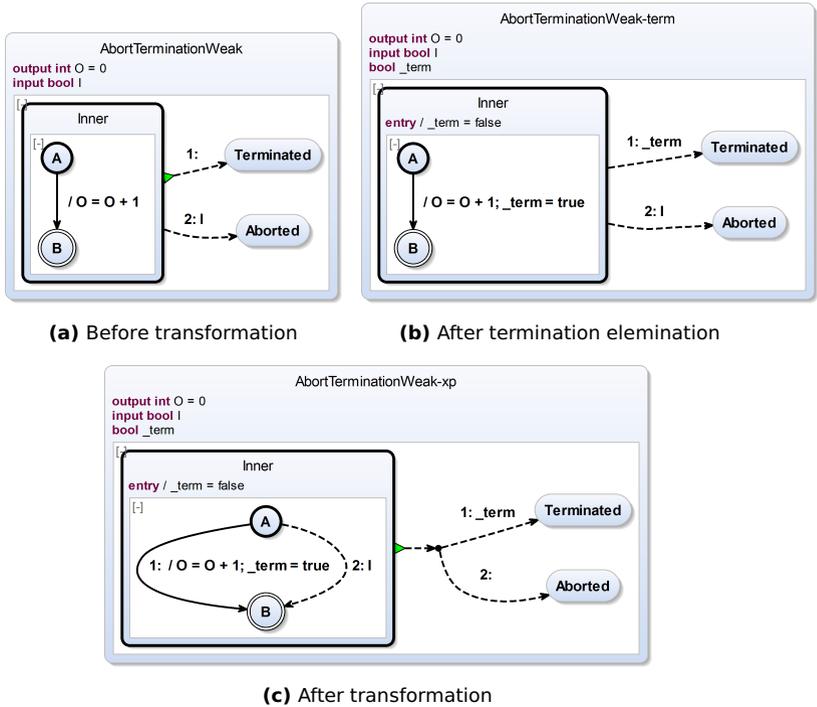


Figure 5.2.21. Abort transformation termination elimination

The revised example has no signals but variables and is shown in Figure 5.2.22a. When transforming aborts, first the termination elimination is applied as explained in the previous paragraph. Figure 5.2.22b shows the result after eliminating terminations. The optimization is already prepared here: Due to the fact that the final state cannot be reached by any immediate transition from an initial node (including the fact that it is not an initial final state) the auxiliary term flag is marked to be *delayed* and named `_termD`.

Figure 5.2.23a shows the further unoptimized transformation result. The drawback is that both regions are potentially instantaneous (by only viewing the structure) and the immediate feedback is problematic.

5.2. High-Level Compilation

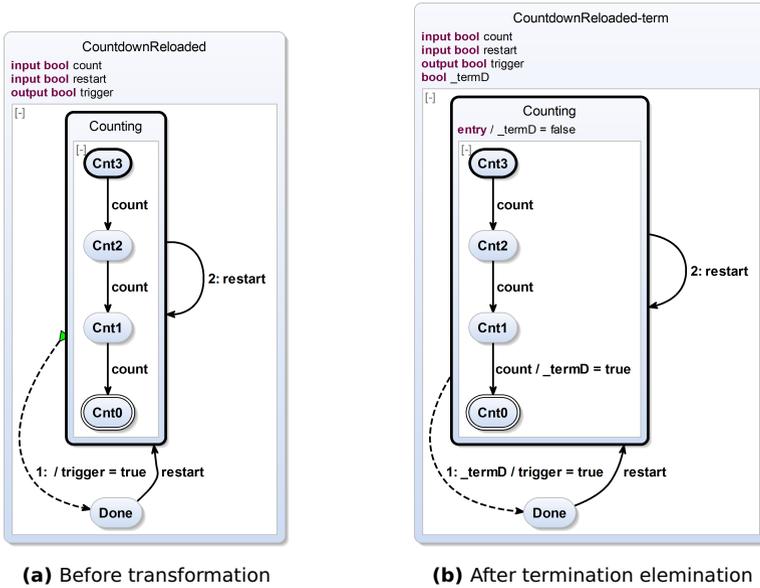


Figure 5.2.22. Abort transformation — further optimizations (I/II)

But using the information that the `_termD` flag cannot be set to true in the same tick when Counting is entered or re-entered, one can safely make the immediate transition in the control region from `_Run` to `_Done` non-immediate. Furthermore, because the original self-loop transition is a delayed weak abort and the initial state `Cnt3` has no incoming immediate transitions, one can safely make the aborting transition from `Cnt3` to `Cnt0` non-immediate too.

Both optimizations are applied in Figure 5.2.23b where none of the regions are able to terminate instantaneously any more and down-stream synthesis is eased.

5. Compiling SCCharts

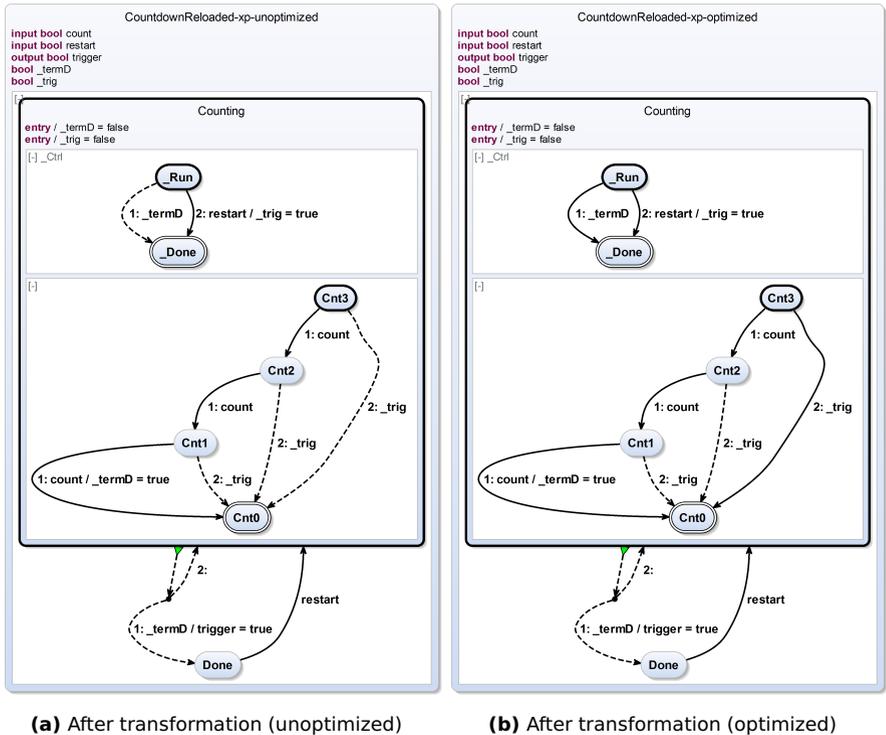


Figure 5.2.23. Abort transformation — further optimizations (II/II)

General Case: With the solution as discussed earlier the (fixed) general form (cf. Figure 5.2.24a) of the abort transformation now produces a result as shown in Figure 5.2.24b. Note that the previously published abort transformation (cf. Figure 5.2.25) could not handle terminations with a higher priority than a weak abort. The reason is the termination detection which is now enhanced as mentioned before. Recall that aborting transitions for strong aborts need to have the highest priority. However, in this case, not both transitions can have the highest priority. It is irrelevant which of these both transitions has the higher priority because it only matters that

5. Compiling SCCharts

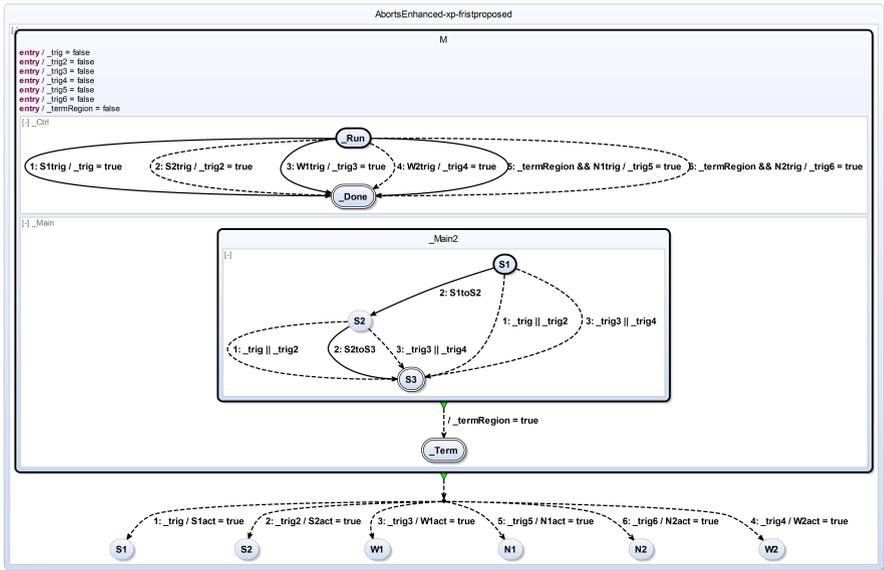


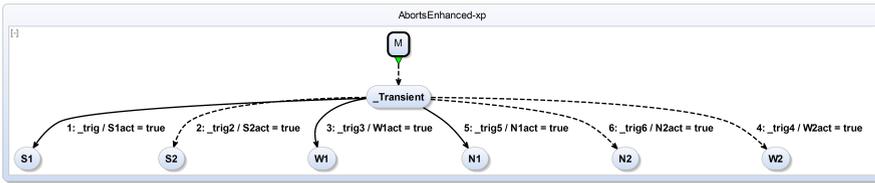
Figure 5.2.25. First proposed abort feature expansion (WTO, from [vHDM⁺13c])

such as `S3`. The auxiliary termination flag `_termD` is set when the transition triggered by `S2toS3` is taken. It is used within the control region to test for the original (conditional) terminations (transitions with priorities 5 and 6).

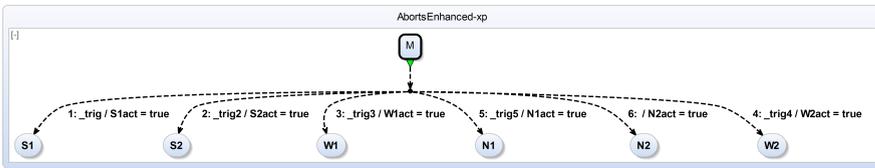
The same does not hold for weak abort transitions as explained in the previous paragraphs. Immediate (`W2trig`) and also delayed (`W1trig`) weak aborts are translated both into *immediate* abort transitions. The delay of `W1trig` is moved to the control region and an auxiliary variable `_trig` is used.

The Dynamic Transient Forking State: The resulting SCChart shown in Figure 5.2.25 of the first proposed abort transformation is syntactically problematic because of the following reason: A connector node must always be transient, i. e., control is not allowed to rest in a connector node. It must be left always in the same tick in which it is entered. The connector node

5.2. High-Level Compilation



(a) Transient State Option with M collapsed



(b) Default Transition Option with M collapsed (chosen)

Figure 5.2.26. Abort transformation: Dealing with the (dynamic) transient forking state. Note that superstate M is collapsed here.

that is entered by the termination from state M has the purpose of deciding which path to take to target either S1, S2, W1, W2, N1, or N2. These were the original targets of the outgoing transitions of state M. Semantically, we can only take the termination from state M to the forking connector state if at least one of the triggers S1trig, S2trig, W1trig, W2trig, N1trig, or N2trig is true. Hence, the connector is clearly left immediately when it is entered. However, syntactically this cannot be assured in general without a *default transition*, which therefore is mandatory for all connectors. Recall that only with a default transition, i. e., an outgoing transition with the implicit true trigger, the connector is guaranteed to be always transient.

For that reason, the abort transformation should also incorporate this syntactical requirement and produce one of the syntactically unproblematic solutions presented in Figure 5.2.26. The option of Figure 5.2.26a does not change anything but makes the connector an ordinary state. The fact that it is guaranteed to be transient semantically is lost but of course still control will never rest there. The option shown in Figure 5.2.26b omits the trigger

5. Compiling SCCharts

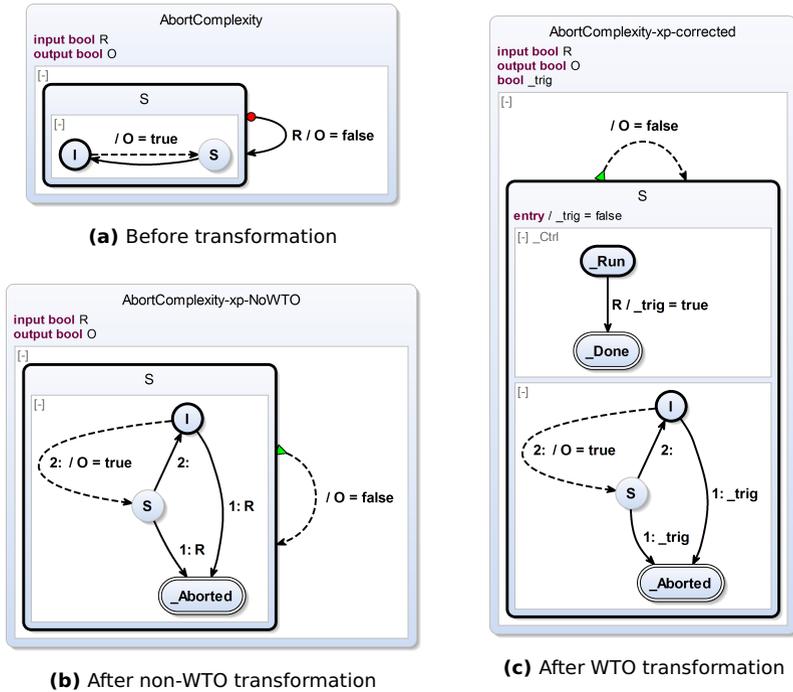


Figure 5.2.27. Further ease down-stream compilation and reduce complexity: WTO version (cf. Figure 5.2.18 on page 151) vs. non-WTO version, which does not always need a control region

of the lowest priority outgoing transition of the connector and makes this the explicit default transition. This can be done because the transformation ensures that one of the triggers must be true if the termination takes control to the connector. If all others triggers have been evaluated to false then the last trigger with the lowest priority is guaranteed to be true. The fact that the connector is transient is now encoded explicitly into the SCChart. Not losing this information seems preferable w. r. t. easy down-stream compilation. Hence, this option was chosen for the implementation of the abort transformation.

Non-WTO Option

There are cases when one does not want the control region to be created. This is the case if the abort trigger is “simple”, i.e., not a complicated expression that results in a lot of code or that is computationally expensive. In this case, evaluating the simple trigger should not cost more computation time than adding an auxiliary variable and an auxiliary concurrent control region in addition to setting this auxiliary variable according to the abort trigger in the control region and evaluating the auxiliary variable in the main region instead of the simple trigger. Another point of view is to leave such kind of optimizations (re-using of intermediate computation results) to lower-level compiler parts which perhaps can be done even more efficiently, e.g., on assembly or circuit level.

Hence, whenever the abort trigger is “simple enough” (see previous discussion) it makes sense to apply a *simpler* abort transformation that will neither introduce auxiliary variables, nor create a control region. It will act like the full abort transformation but whenever the full abort transformation would place a trigger referring to the auxiliary variable it will place the original abort trigger instead to the transition in question. This optimization will not only get rid of the auxiliary variable itself but also of the additional entry action resetting this variable to false.

The optimization may or may not be possible for the transformation of certain transitions: A necessary condition is that trigger evaluation does not have harmful side effects. Because of this, conservatively, we will not allow this optimization for any side effect trigger (e.g., a trigger which evaluates host code). Any pure SCCharts trigger is side effect free by construction (IUR). Another necessary condition is that there are no delayed weak abort transitions. These require an auxiliary variable which is delayed by a control region as explained above (cf. Section 5.2.8 on page 152). Yet another necessary condition is that terminations are immediate and have no trigger, i.e., only core terminations and no extended terminations are allowed. Otherwise a control region needs to handle the delay of a termination or possible extra conditions of a conditional termination. Further necessary conditions are 1. that there are no final states that do not have a corresponding termination transition and 2. that there are no

5. Compiling SCCharts

delayed strong abort transitions mixed with other types of transitions. Both conditions are explained by counter examples in the following two paragraphs.

Figure 5.2.27b shows the expanded version of Figure 5.2.27a where there is no control region and the trigger R is set directly instead of an auxiliary variable `_trig` that was used before in the WTO expansion shown in Figure 5.2.27c. Note that in this version (b) it would be *wrong* if the aborting transitions that now evaluate R directly would be immediate. Immediate abort transitions in the main region were only correct when they were triggered by an auxiliary variable whose setting to true is delayed by a control region (cf. Figure 5.2.18b on page 151). Also, this version (b) could be wrong if evaluating the trigger R would have harmful side effects, e. g., a host code function which not evaluates to the same boolean value throughout a tick computation.

On the one hand, as can be seen in Figure 5.2.27, the non-WTO variant tends to create less concurrency. Hence, it will create more compact code, it will therefore execute faster, and it will be easier to schedule in down-stream compilation. On the other hand, if triggers are more complex to evaluate or the absence of harmful side effects cannot be guaranteed then the WTO variant should be chosen.

There are further criteria which prevent also the non-WTO variant from getting rid of a control region such as delayed weak aborts or mixed strong aborts. In these cases the non-WTO variant acts for such transitions the same way as the WTO does (see above).

Mixed Immediate and Delayed Abort — **Earlier Termination:** There is a pitfall when choosing the simpler-looking non-WTO option, if immediate and delayed aborts are mixed and the inner behavior may be immediate (cf. Figure 5.2.28a). Then the transformation must not combine both types of aborts into a single transformation as in Figure 5.2.28b.

5.2. High-Level Compilation

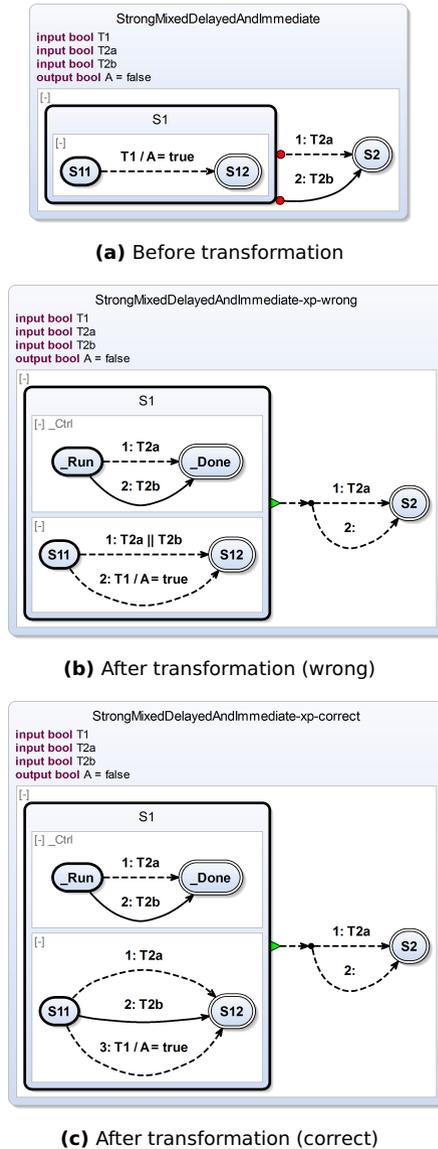


Figure 5.2.28. Abort (non-WTO): Mixing immediate and delayed aborts shall not lead to an earlier termination.

5. Compiling SCCharts

This was correct in the WTO case since there, auxiliary triggers were also delayed by the control region. In the non-WTO case, the original triggers may affect the SCChart directly. To prevent this, one must distinguish between immediate and delayed aborts, as done in Figure 5.2.28c.

There is a second pitfall revealed by this example (cf. Figure 5.2.28c). That is, the additional control region in this example is mandatory because S12 is a final state but S1 has no termination. If the transition to S12 is taken then the original SCChart in Figure 5.2.28a does not leave S1. Still “all regions” of S1 are in their final states. In the transformed model shown in Figure 5.2.28c one must prevent leaving S1 which is done by the control region which does not terminate until one of the aborts is triggered.

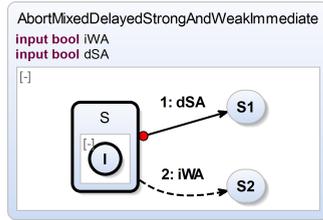
Mixed Delayed Strong Aborts — Priority Inversion: In the case of delayed strong aborts and other transition types, as shown in the example of Figure 5.2.29a, the priorities for the outgoing transitions should be respected in the first and all following ticks for the expanded version.

However, if omitting the control region in the non-WTO case then this leads to priority inversion in the first tick as shown in Figure 5.2.29b. If `iSA` and `dWA` are both present in the initial tick, then the original delayed transition triggered by `dWA` should not be taken because of the delay, although it has a higher priority. However, if the `_Aborted` state is reached and the termination from `S` is taken then the information is lost that the outgoing transition triggered by `dWA` is not yet a valid choice for the initial tick. Hence, this transition is wrongly taken because of its higher priority.

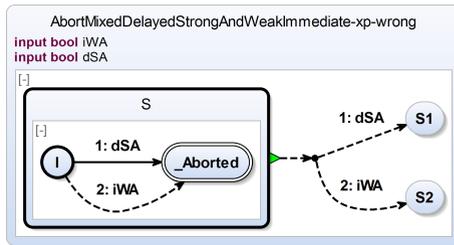
Summarizing, if there are both, delayed and immediate (strong) aborts, then a control region is required as depicted in Figure 5.2.29c. The auxiliary signal `_trig` is potentially set to true in the control region. However, as the delay of the original transition is also duplicated in the control region `_trig` can never be true in the first tick. The outgoing transition from `S` is now only triggered by `_trig` and not by `dWA` any more. Hence, in the above scenario of the initial tick, now correctly the transition leading to state `S2` would be taken.

Similarly, also a mix of a delayed strong abort and 1. an immediate one or 2. a termination are a problem. An exception (optimization) can be made for terminations if the state cannot terminate instantaneously.

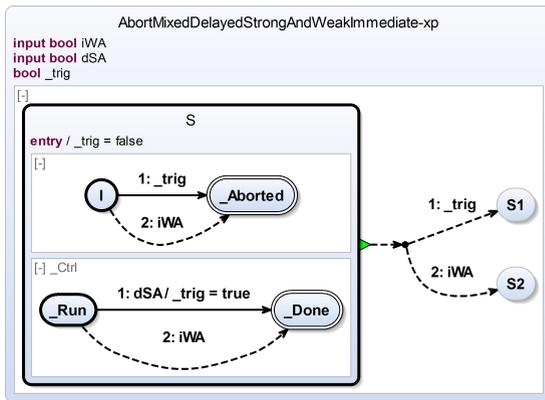
5.2. High-Level Compilation



(a) Before transformation



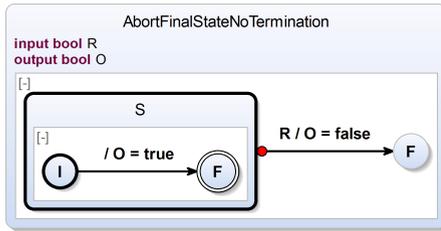
(b) After transformation (wrong)



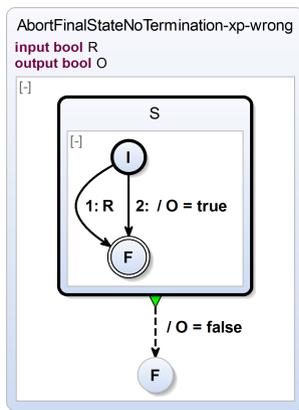
(c) After transformation (correct)

Figure 5.2.29. Abort (non-WTO): Mixing delayed strong aborts with other transformation types shall not lead to priority inversion.

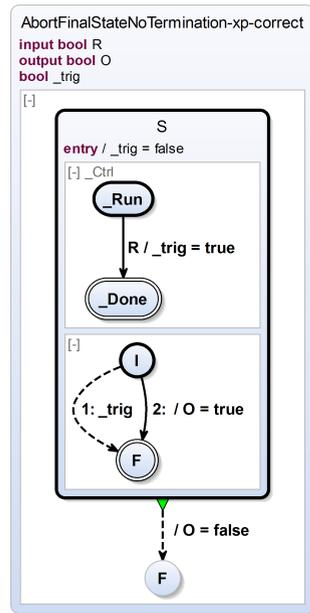
5. Compiling SCCharts



(a) Before transformation



(b) After transformation (wrong)



(c) After transformation (correct)

Figure 5.2.30. Abort (non-WTO): A final state but no termination shall not lead to an undesired leaving of a state.

Final States w/o Termination: In the case of none original outgoing termination transitions but a final state, as shown in the example of Figure 5.2.30a, the superstate should not be left even if the final state is reached.

If omitting the control region in the non-WTO case then this leads to a wrong SCChart as depicted in Figure 5.2.30b. The new termination transition will not only be triggered when the prior abort signal R is true but also when the final state F is reached by the internal transition, which sets O to true. This is incorrect.

Hence, the control region should not be omitted in the case of final states without termination transitions. If a control region exists (cf. Figure 5.2.30c) then after reaching F, the termination transition will not be taken because the control region still waits for signal R to become true. Only if R eventually becomes true then the control region will also transition to its final state and state S is correctly left.

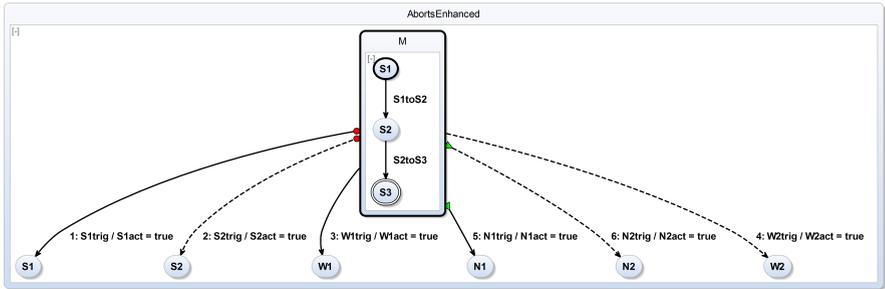
General Non-WTO Abort: The general form of the abort transformation with the mentioned non-WTO option is illustrated by Figure 5.2.31 and shows how the model in Figure 5.2.31a is expanded with non-WTO to the resulting model in Figure 5.2.31c.

Recall that even in the non-WTO case the control region is necessary if

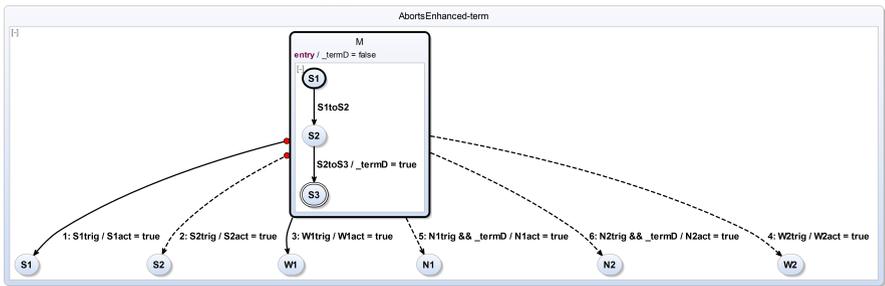
1. there are non-core termination transitions, i. e., termination transitions with triggers or a delay, or
2. there are weak delayed outgoing transitions, or
3. there are mixed delayed abort transitions and other (immediate) transition types, or
4. there are final states without a termination transition.

In any other case, the implementation will not create a control region as seen in the following example of Figure 5.2.32.

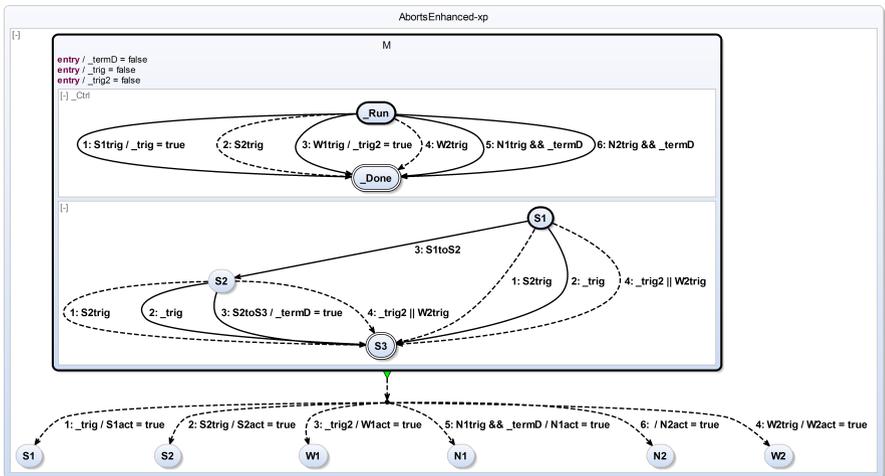
5. Compiling SCCharts



(a) Before transformation

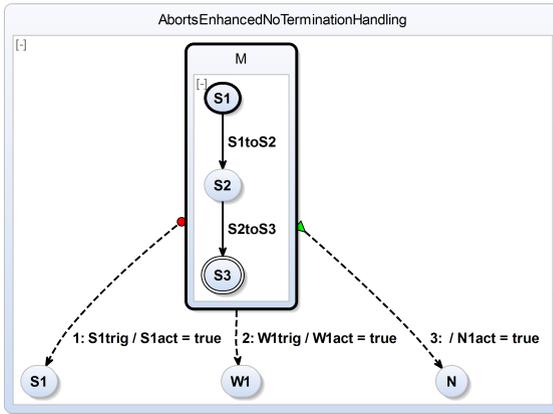


(b) After eliminating terminations

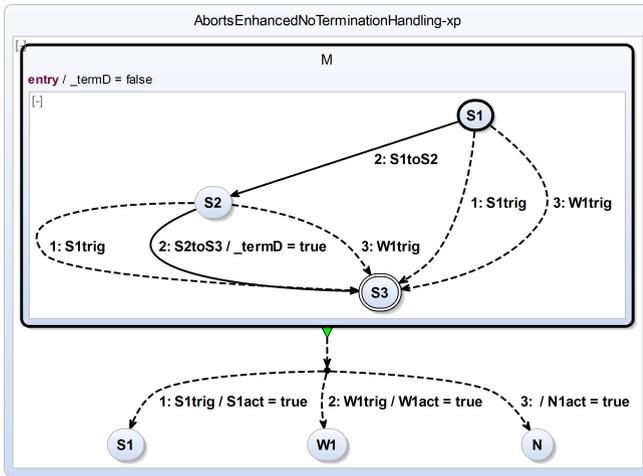


(c) After transformation

5.2. High-Level Compilation



(a) Before transformation



(b) After transformation

Figure 5.2.32. Enhanced abort transformation with correctly omitted control region (non-WTO)

5. Compiling SCCharts

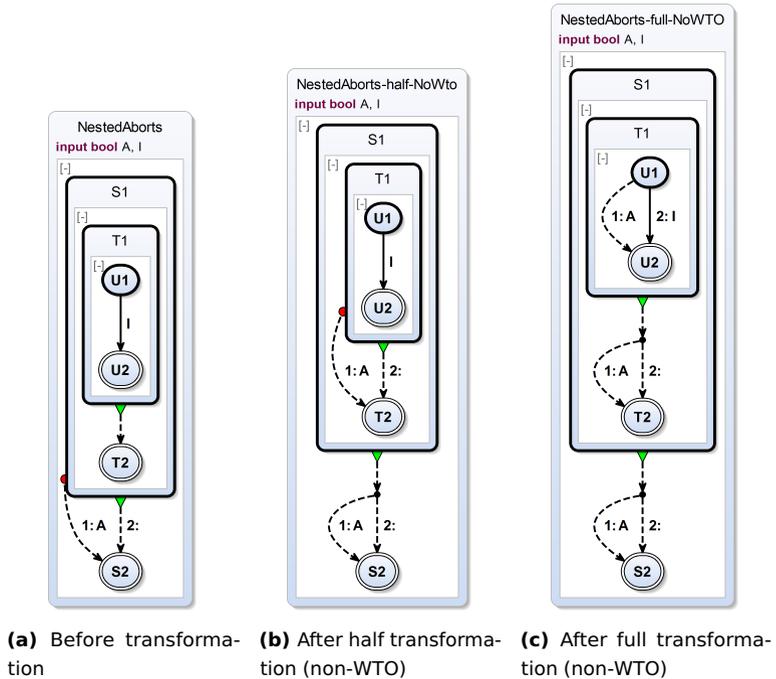


Figure 5.2.33. Nested abort example (original and non-WTO)

Nested Aborts

Regarding the WTO and the non-WTO option, nested aborts must be considered. An example is given in Figure 5.2.33a. It has an immediate strong abort outgoing transition from superstate S1 triggered by A. S1 contains another superstate T1, which can either be in U1 or U2. When the abort is triggered then the nested state structure of S1 must be left no matter which state inside S1 is active.

This can be easily seen in Figure 5.2.33c, which shows the result of the non-WTO version of the abort transformation. The trigger A is replicated several times inside the nested structure.

5.2. High-Level Compilation

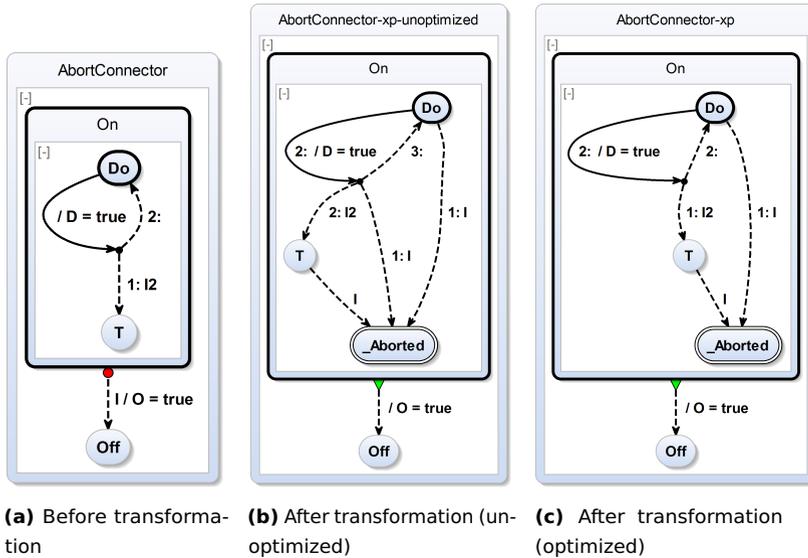


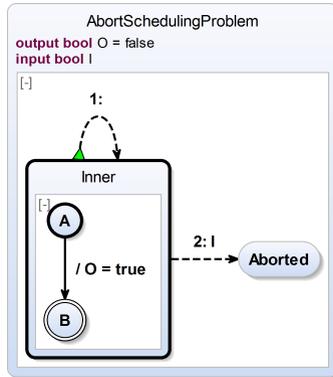
Figure 5.2.34. Abort feature expansion transformation with connectors

The abort transformation is applied from outside to inside. This means an abort from outer side is passed on to inner states and replicated there. Figure 5.2.33 shows the original, the half transformed, and the fully transformed model with the non-WTO option. Note that the strong abort from S1 to S2 (a) is passed on to the inside, now aborting T1 to T2 (b) and finally aborting U1 to U2 as a simple transition (if the source state is not a hierarchical one).

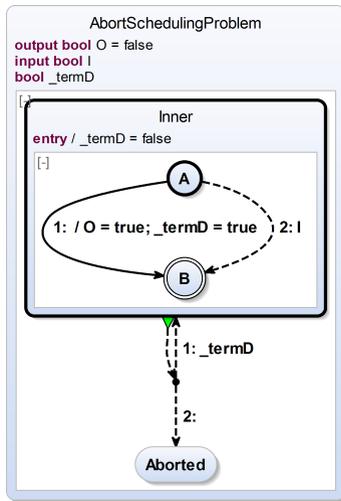
Aborts and Connectors

When expanding aborts, the transformation iterates over all inner states and adds explicit aborting transitions. Because connectors are special states, an explicit aborting transition is also added for each of them. Figure 5.2.34b shows such an expanded version of the SCChart shown in Figure 5.2.34a.

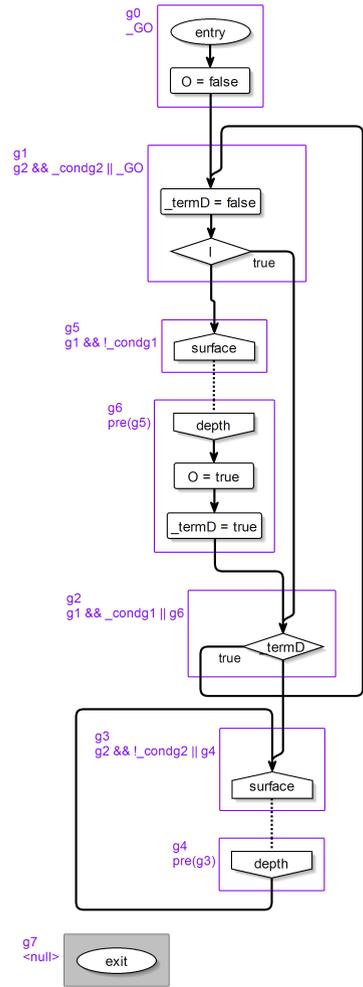
5. Compiling SCCharts



(a) Before transformation



(b) After transformation



(c) SCG contains an immediate cycle

Figure 5.2.35. Scheduling problem for self-loops: The `_termD` conditional can be traversed multiple times in a tick.

This is triggered by \perp in the same way as both other ordinary states Do and T are aborted. However, if a connector state is entered then, by definition, it is immediately left. Hence, a connector can be seen as a junction (split or join) for transitions in order to share common transition parts according to the WTO principle. Control can never rest in a connector state. Hence, on the one hand, if the source state is aborted strongly then the connector state is not reached and it does not change anything whether it has an aborting transition or not. On the other hand, if the source state is aborted weakly then the connector state may be reached but should be processed normally for the (immediate) “last will”. Also in this case, the aborting transition will never be taken because there is always an (immediate) default transition with a higher priority, by definition. Hence, in both cases, an aborting transition from a connector is superfluous *dead code* and can safely be omitted as shown in Figure 5.2.34c.

Aborts and Scheduling Difficulties

When transforming self-loop transitions or any similar (immediate) *cyclic construct*, this may result in a model which the current compiler is not able to schedule although the model is semantically correct. Consider the example of Figure 5.2.35 and the original model of Figure 5.2.35a. If this model is entered in the initial tick, it will enter state A. If, in the second tick, the weak abort \perp is triggered then first the inner behavior is permitted: 1. transitioning to B and setting O to true, 2. terminating Inner because B is a final state, 3. re-entering Inner immediately and 4. exiting Inner by the immediate weak abort transition to Aborted.

The semantically correctly expanded version of this model is shown in Figure 5.2.35b. In the second tick, first the transition with priority 1 from A to B is taken. This sets the auxiliary _termD flag to true which was set to false when Inner was entered in the initial tick. Then, the Inner state is left by the termination but re-entered because the _termD flag is true. Upon re-entry, the _termD flag is set to false again. Now, the immediate transition with priority 2 from A to B is taken. This does not modify the _termD flag. Hence, when the Inner state is left for the second time in this tick, the state Aborted is entered.

5. Compiling SCCharts

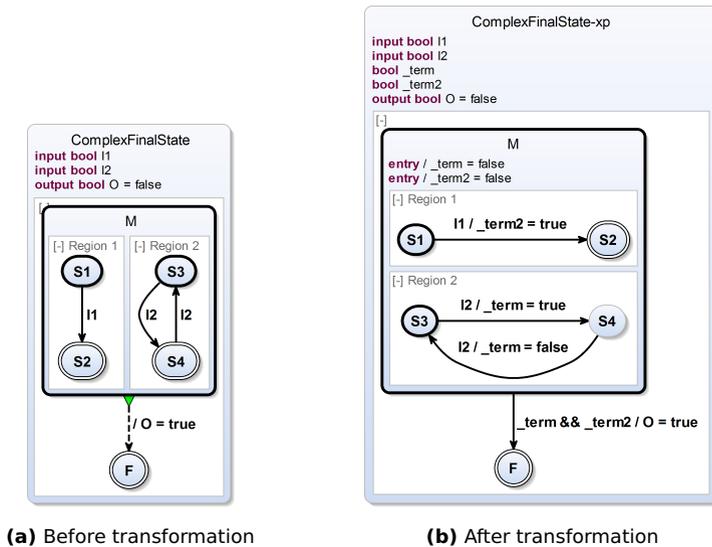


Figure 5.2.36. Complex final state feature expansion transformation

The SCG as shown in Figure 5.2.35c reveals the scheduling difficulty. There is schizophrenia involved here. The conditional node which decides either to re-enter Inner or to transition to Abort is passed twice in such a second tick. Due to the possibly immediate re-entry, there is an immediate dependency cycle which currently cannot be resolved down-stream.

5.2.9 Complex Final State

Complex final states (cf. pseudocode in Section 5.3.6 on page 244) are final states that have internal behavior such as regions, during actions, or exit actions or/and outgoing transitions. All this is not allowed for ordinary final states of Core SCCharts. In order to eliminate a complex final state, for any parent superstate of a complex final state, one must track the termination of its regions and then *terminate* the complex final state explicitly when all regions terminate. In order to terminate it, a weak abort transition is used.

5.2. High-Level Compilation

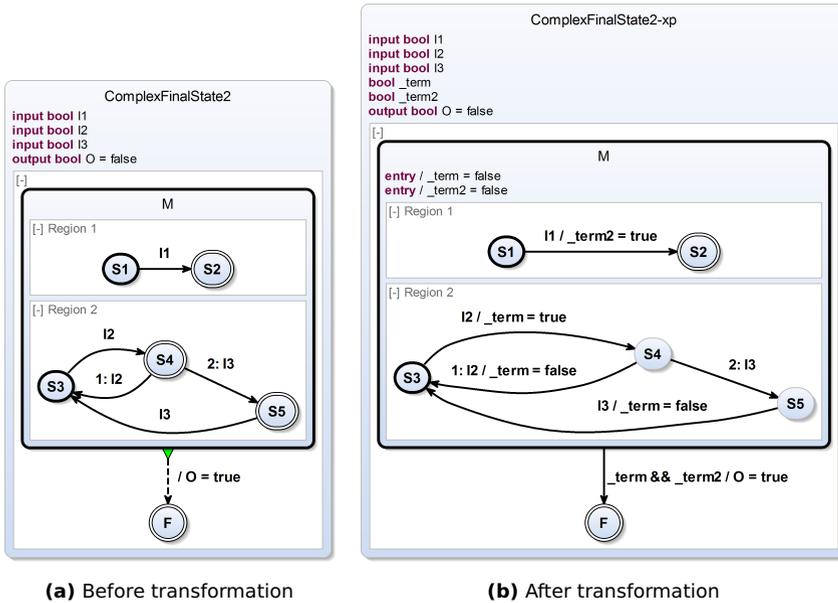
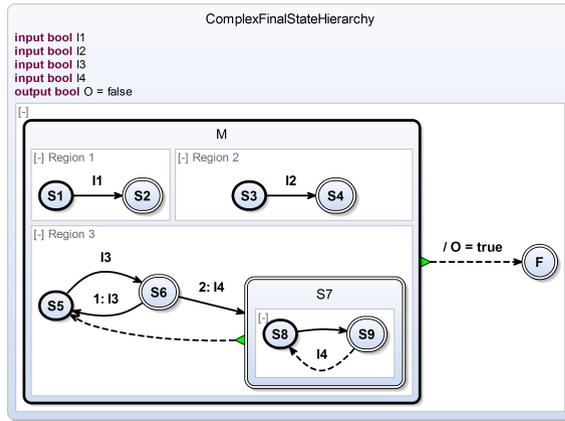


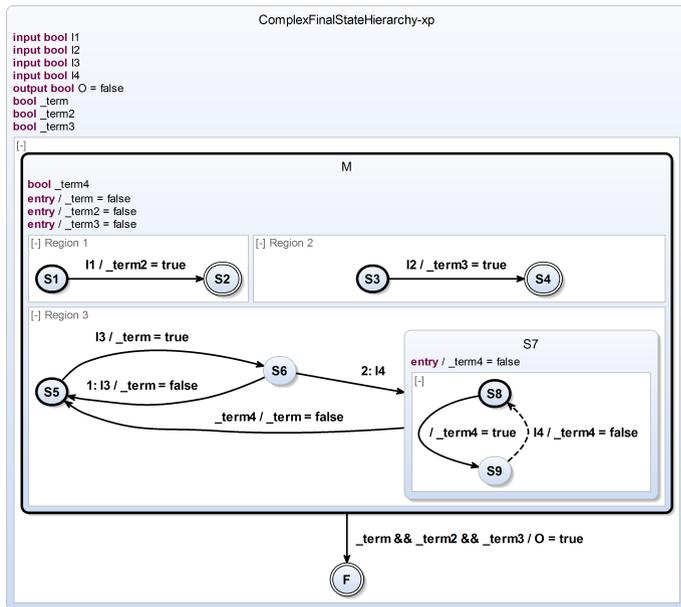
Figure 5.2.37. Complex final state feature expansion transformation for two complex final states

The complex final state thus becomes an ordinary state. Figure 5.2.36a shows a simple example where state S4 is a complex final state because it has an outgoing transition to state S3 triggered by I2. Figure 5.2.36b depicts the transformed version of the model where the complex final state has been eliminated. Hence, S4 has become a non-final state. The termination transition is changed to a weak abort and the trigger is modified such that the weak abort is taken when all regions of the superstate M have terminated, including the region with the complex final state. Therefore, there exists a `_term` variable for each region. These “termination flags” are set to true if a final state is entered. They are set to false if a final state is left. They are not set to another value if the transition goes from one complex final state to another or to some ordinary final state.

5. Compiling SCCharts



(a) Before transformation



(b) After transformation

Figure 5.2.38. Complex final state feature expansion transformation in the hierarchical case

5.2. High-Level Compilation

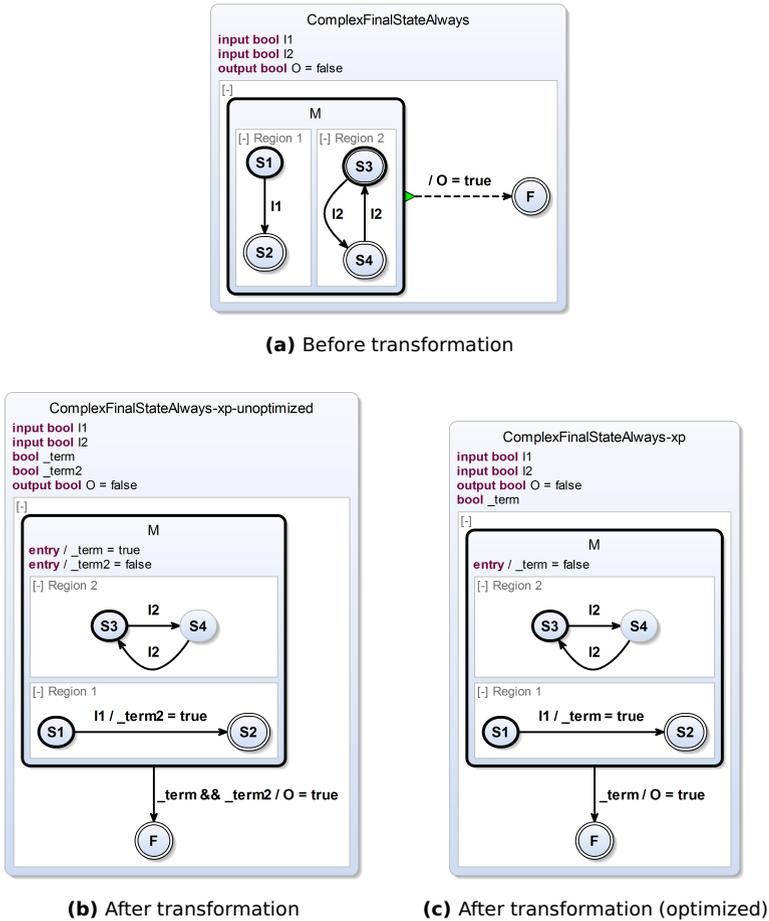


Figure 5.2.39. Complex final state feature expansion transformation optimizing regions with final states only

The `_term` variable is initially set to false because the initial state in all cases is not a final state in this example.

5. Compiling SCCharts

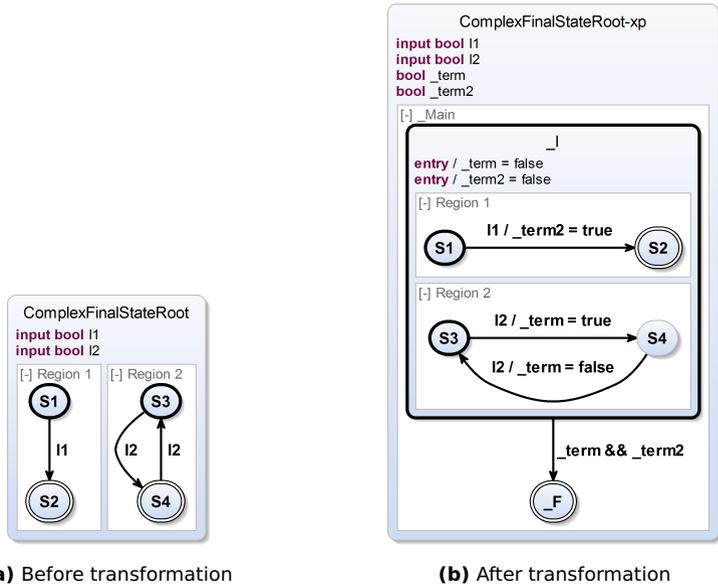


Figure 5.2.40. Complex final states in the root state

Another example is illustrated in Figure 5.2.37. This time, there are two complex final states S4 and S5. Note that the transformed SCChart shown in Figure 5.2.37b reveals that the termination flag `_term` is not modified if the transition from one complex final state to another (complex) final state, e. g., from S4 to S5, is taken. It also reveals that for each region of M , only one termination flag is necessary regardless of how many (complex) final states each region may contain.

A yet even more complex example is illustrated in Figure 5.2.38. In the example shown in Figure 5.2.38a, the superstate S7 is a complex final state and itself contains a region with two states S8 and S9. The *shallow termination* semantics is that the superstate M terminates if all its directly contained regions have reached a final state or a complex final state that they do not leave again in this tick, such as S7. Thus, it does not matter

5.2. High-Level Compilation

whether on a deeper hierarchy layer, as within S7, regions are in a final state or not. In this example it is even the contrary: If the inner region of S7 is in its final state S9 then the termination transition from S7 to S5 is triggered and taken which prevents the termination of M. Only if the inner region of S7 is or ends up in S8 and also Region 1 and Region 2 have terminated then superstate M is able to terminate. The transformed model is shown in Figure 5.2.38b. If S7 terminates, the abort to S5 is taken which sets `_term` to false and prevents the termination of M.

Figure 5.2.39a shows an example where the `_term` variable can be omitted safely if all states in a region are final states. Such a region cannot prevent its superstate from terminating. The `_term` variable of the transformed model (cf. Figure 5.2.39b) is initialized with true but never changed. Hence, there is no need to test for it in the weak abort termination. The optimized model, omitting the flag, is presented in Figure 5.2.39c.

In the case that a complex final state is defined in a region of the root state and all other root regions may also possibly terminate, then the SCChart should be able to terminate if the complex final state is entered and all other regions are in their final states. Therefore, all regions must be encapsulated in order to be able to weakly abort them as described above. This is illustrated by Figure 5.2.40. Figure 5.2.40a has a possibly terminating ordinary root region and another root region with a complex final state S4. Figure 5.2.40b shows the expanded version. Here, all regions of the SCChart are encapsulated into an auxiliary initial state `_I`. The weak abort leads to a new auxiliary final state `_F`, which terminates the SCChart.

5.2.10 During Action

When a *during action* (cf. pseudocode in Section 5.3.7 on page 246) for a state is defined then whenever the control is in this state, even if it is left by a weak abort or termination in the same tick, the during action is performed.

A during action can optionally carry a trigger which additionally guards this action. If no trigger is defined, the implicit true trigger enables the during action. *Immediate during actions* execute in the same tick when the defining state is entered as well as in subsequent ticks. *Non-immediate during actions* can execute only in subsequent ticks and do not execute in the tick

5. Compiling SCCharts

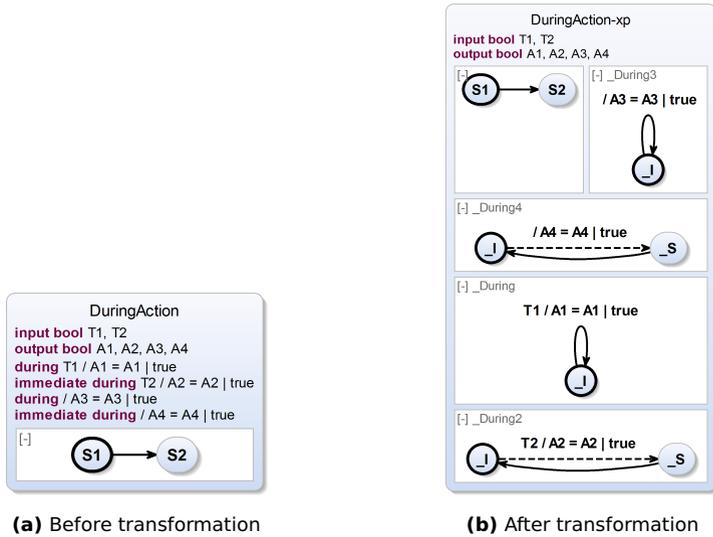


Figure 5.2.41. During action feature expansion transformation

when the state is entered. A during action can neither prevent a state nor the SCChart from terminating.

Figure 5.2.41a shows several examples for during action definitions within a state. Some are immediate, others are non-immediate. Some have triggers, others have the implicit true trigger.

The transformation for during action is conceptually pursuing the following approach:

1. Create a separate auxiliary region R for each during action.
2. Create an initial auxiliary state $_I$ in R .
3. Now, the behavior differs for immediate and non-immediate during actions:
 - Immediate: Additionally create a second auxiliary state $_S$ in R . Connect $_I$ with $_S$ using an immediate transition which performs the

during action. Loop back from `_S` to `_I` with a delayed transition that has no trigger or effects.

- ▶ Non-immediate: Add a delayed self-loop transition from `_I` to `_I` which performs the during action.

Figure 5.2.41b shows the expanded versions of all four different during actions from Figure 5.2.41a.

During Action Termination Pitfall

The above translation works well if there is no outgoing termination transition from the parent superstate. Consider Figure 5.2.42 where there is such a termination transition.

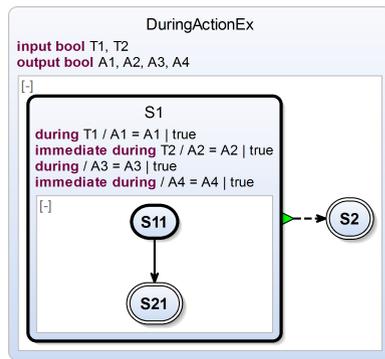


Figure 5.2.42. During action examples with termination

In this scenario, the (simple) translation from above would not be correct because it would prevent the termination so that it could never be taken. Figure 5.2.43a shows the incorrect simple translation.

Once `S21` is entered, the outgoing termination transition from state `S1` should be taken. However, the simple during action translation has created an auxiliary region for each during action that has no final state. With only one of these regions, `S1` can never be exited by a termination transition.

5. Compiling SCCharts

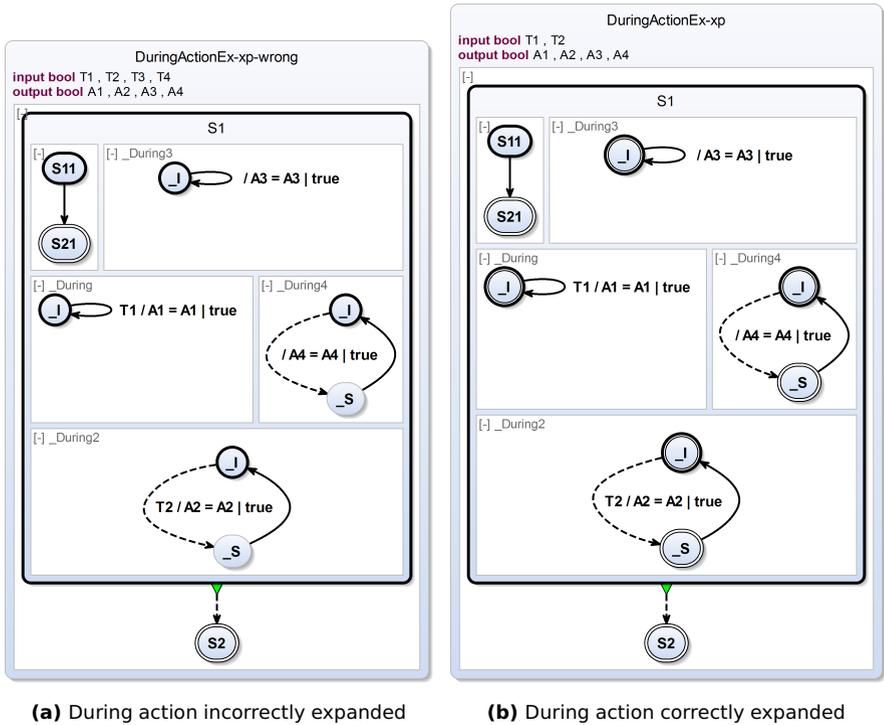


Figure 5.2.43. During action examples with termination expanded with simple during action transformation (left) and with advanced during action transformation (right).

Hence, if there are termination transitions outgoing then a more advanced translation is necessary which respects a possible termination. It makes use of *complex final states* as covered in Section 5.2.9 on page 176.

Note that it is not sufficient to just make `_S` a final state because if the trigger of a *conditional* during action (e. g., `_During2`) evaluates to false, the control would rest in `_J`, which is not a final state. This again could prevent a termination transition to be taken.

5.2. High-Level Compilation

The (general) advanced translation for during actions is conceptually pursuing the approach listed on page 182. However, it is extended such that if there exist outgoing termination transitions of the parent superstate then make `_I` (and `_S`) final.

Note that because `_I` (and `_S`) both have one outgoing transition, these become *complex final states*.

Figure 5.2.43b shows the correctly translated during actions. Observe that there is an outgoing termination transition to state `S2`. Hence, all auxiliary states created by the during action transformation are also final states. Note that the complex final state transformation will handle regions that only contain final states in an optimized way by not introducing a termination flag (see Figure 5.2.39 on page 179).

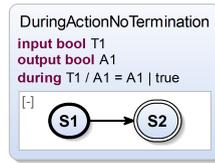
Creating Complex Final States

Since the advanced transformation for during actions creates complex final states, it is desirable to avoid the advanced transformation whenever it is possible. This is the case if there is no outgoing termination transition. Hence, the parent superstate is either never left or it is left by a strong or weak abort only. In case the state is not left at all, the simple during transformation can be applied. In case of an abort, the abort feature is handled in down-stream compilation.

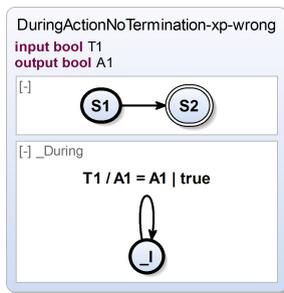
Consequently, we only have to consider using the advanced during action transformation if there is an outgoing termination transition from the declaring state of the during action.

Additionally, we can still stick to the simple transformation in case a possibly existing termination can never be triggered. This is the case if there exists a region with no final state. Note that this is an optional optimization but it simplifies down-stream compilation because a termination, which can never occur, does not require any auxiliary termination-detection-flags (cf. Section 5.2.9 on page 176).

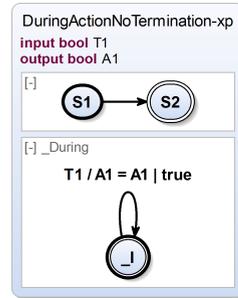
5. Compiling SCCharts



(a) During action with no (explicit) termination transition defined in the root state



(b) During action incorrectly expanded



(c) During action correctly expanded

Figure 5.2.44. Simple during transformation (b) and advanced during transformation (c) for a root state that has an implicit termination transition. The simple during action transformation is wrong here because the during action would prevent the SCChart from terminating.

Root State During Action

There is an interesting design decision associated with the question when to use the simple and when to use the advanced during action transformation:

Should the during action of a (a) state or (b) SCChart terminate if all of its regions terminate?

It is not obvious how this should be handled. Consider the following case (b) of Figure 5.2.44a. Here, a during action is defined for the SCChart in the root state. Should the during action further be executed if S2 is entered? There are several options as listed in Table 5.2.1.

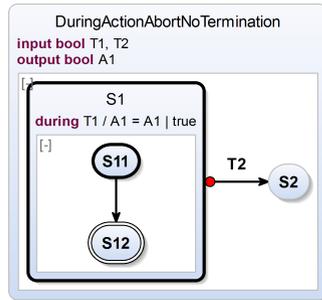


Figure 5.2.45. During action should continue as long as state S1 is active and not aborted

An SCChart should be able to terminate if all regions of the root superstate terminate. Semantically, this means assuming an *implicit* termination transition leaving the SCChart and leading to *program termination* where this transition is neither modeled nor visible explicitly. But logically such a transition always exists for every SCChart. It basically only leads to termination of the SCChart, once all concurrent regions have terminated. A during action should not prevent this behavior. Hence, if all regions have at least one final state, the advanced during action transformation should be applied even for the root state.

Figure 5.2.44a depicts why preventing an SCChart from terminating seems wrong (Figure 5.2.44b) and it also depicts the design decision for considering an implicit termination transition for the root state as an “exception” compared to other superstates (Figure 5.2.44c).

First Proposed Expansion

The first proposed expansion for during actions is illustrated by Figure 5.2.46. In the example of Figure 5.2.46a, a simple conditional during action is modeled. Note that the outer state is considered a root state or a superstate that could be left by a termination. The result of the originally proposed [vHDM⁺13c] transformation can be seen in Figure 5.2.46c. It does not make use of complex final states in contrast to the current during action

5. Compiling SCCharts

Table 5.2.1. During actions in root state: Possible design decisions and chosen (*) decision

Option	Rationale	Pro	Contra
Simple Transformation	The root state has no termination transition going out. Hence, the simple transformation rule can be applied.	Consistency with other superstates.	An SCChart cannot terminate.
Advanced Transformation	For all superstates (including a root state), if their regions terminate, also terminate the during actions. Do not require any explicit or implicit termination transition.	Consistency with other superstates, an SCChart can terminate.	A during action (of a superstate which is aborted) is possibly terminated before any abort occurs when there is no termination transition going out but all other regions have terminated (cf. Figure 5.2.45).
Implicit Termination*	Terminate during actions if a termination transition is taken. The root state is considered to have an <i>implicit termination transition</i> .	The SCChart can terminate, a during action of a superstate can run until it is aborted.	Exception for the root state.

transformation result as shown in Figure 5.2.46b. Hence, it seems to be less compact. In fact, the original during action transformation was designed to be processed *after* the abort transformation. Therefore, it could not make

5.2. High-Level Compilation

use of higher-level features and need to handle the termination detection by itself. Besides a less compact result and a more complex transformation, the first proposed during transformation had another drawback: It could not be strongly aborted because aborts have already been expanded before. For these reasons we decided to handle during actions before aborts and let the during action transformation use complex final states (cf. Figure 5.2.46b) to gain a more compact result, a less complex transformation, and to allow preemption.

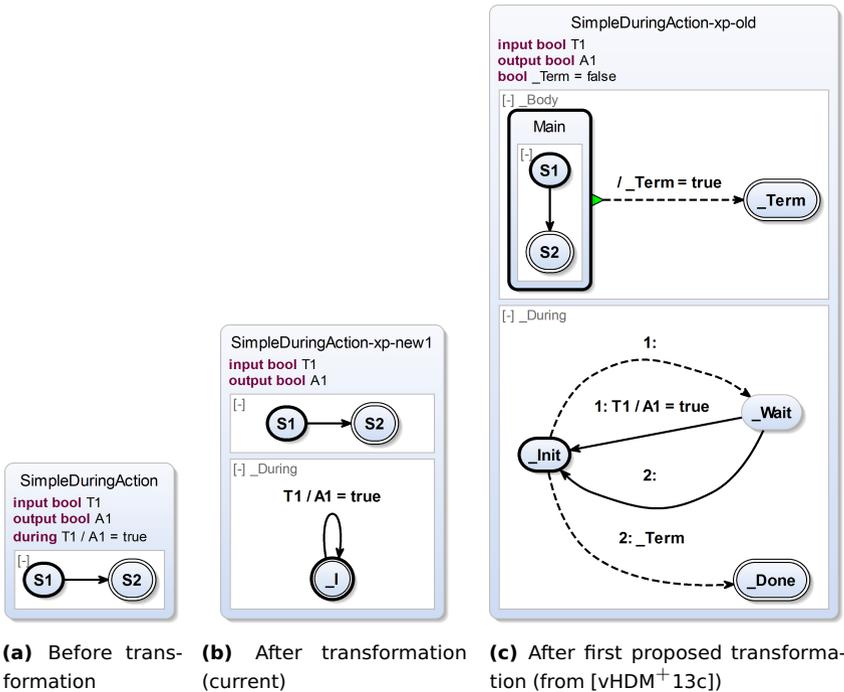


Figure 5.2.46. Simple during action example: Figure 5.2.46c shows the first proposed expansion (from [vHDM⁺13c]) and Figure 5.2.46b shows the current expansion.

5. Compiling SCCharts

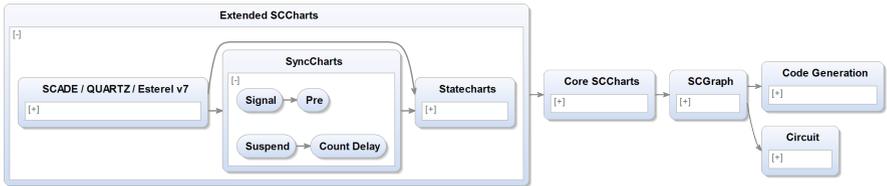


Figure 5.2.47. SyncCharts feature transformations in KiCo compiler selection: Signal, Pre, Count Delay, and Suspend

5.2.11 Signal

Synchronous *signals* (cf. pseudocode in Section 5.3.8 on page 247) are borrowed from SyncCharts and Esterel where these are primary elements for communicating with concurrent model parts and with the environment. In contrast, signals are syntactic sugar in SCCharts but variables are the primary elements for communication. Signals and variables share the same name space. Pure signals are eliminated by transforming them into boolean variables where a true value corresponds to a present status of a signal and a false value corresponds to an absent status of a signal. A valued signal is eliminated by transforming it into a boolean variable for its presence status and a second variable which holds the value. The second variable has the same type as the valued signal.

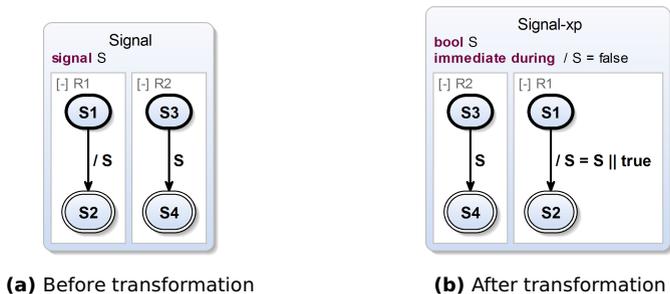


Figure 5.2.48. Signal feature expansion transformation

Pure Signal

Figure 5.2.48 shows the transformation of a local *pure signal* S . In Figure 5.2.48a, S is used for communicating between two concurrent regions $R1$ and $R2$. In region $R1$, the signal S is emitted when transitioning from state $S1$ to $S2$. In region $R2$, the signal S is tested in the trigger of the transition from state $S3$ to $S4$.

Figure 5.2.48b shows the expanded model where signal S is replaced by a variable. All read occurrences for S do not have to be modified but all emissions of S are replaced by a relative write $S = S \parallel \text{true}$.

The declaration is changed from a signal to a boolean variable. To represent the presence status correctly, the new variable representing the presence status of the signal must be initialized to false in every tick. This is done using an immediate during action and an absolute write $S = \text{false}$. The immediate action ensures an initialization even in the first tick when the scope of the signal is entered. The absolute write ensures that the signal's presence status variable is reset to false in each tick before a possible emission occurs, i. e., a relative write to the signal's presence status variable which sets it to true (cf. Section 2.6 on page 38).

Typically, when a signal scope is left and re-entered again within the same tick in SyncCharts or Esterel, one has to deal with *schizophrenia* [Ber00b] problems. The emulation of signals in SCCharts naturally solves these problems by construction. Whenever the scope of S is re-entered, the immediate during action will reset the signal presence status variable to false.

Note that in case of a possible entry action which could emit a declared signal, the *absent during action* must occur on an outer layer to ensure that it will occur sequentially before such a possible entry action. If there are no such actions declared in the state then this extra layer is not necessary and can be omitted.

Valued Signal

Valued signals in Esterel or SyncCharts are pure signals that additionally carry a typed value. To represent a valued signal, two variables are needed: One for the presence status and another one for the value. The value is

5. Compiling SCCharts

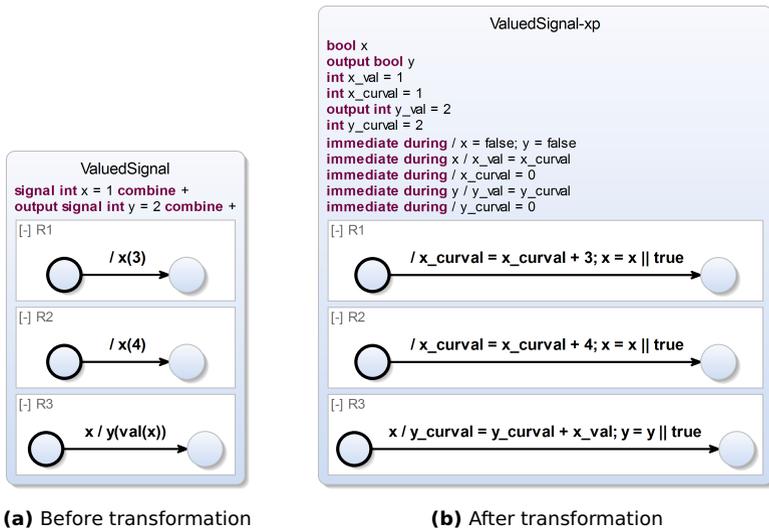


Figure 5.2.49. Valued signal feature expansion transformation

persistent across tick boundaries until it is changed. The value can only be changed when the signal is emitted. If, in a tick, several emits to a valued signal are executed then a *combination function* is required in order to combine these values. The order in which these concurrent emissions occur is undefined and must not influence the result to prevent any kind of race condition and non-deterministic behavior. This is solved by the required combination function since it must be associative and commutative. Another auxiliary variable is used to evaluate the combined value in the current tick. This auxiliary current value variable is used to update the value variable of the signal in ticks where an emission occurs.

Figure 5.2.49 shows the transformation for two valued signals x and y . Note that both are declared with a combination function “+”. y is declared as an output signal. x is emitted in region R1 with value 3. Concurrently, x is also emitted in region R2 with value 4. In this example, + is used as an associative and commutative combination function. The resulting value

5.2. High-Level Compilation

of x , if both emissions $x(3)$ and $x(4)$ are executed in the same tick, will be seven, independent from the exact execution order. Region R3 waits until the valued signal x is present and emits the other valued signal y with the (final) value of x retrieved with $\text{val}(x)$. Since there is only one modeled emission of y , there seem to exist no concurrent emissions. However, the value of y_curval is still reset to the neutral element (w. r. t. the combination function and value type) in each tick such that a combination function is necessary for y too.

The transformation turns x and y into boolean variables as for pure signals. Each boolean variable represents a signal status. Also, these status variables are reset to false by immediate during actions at the beginning of each tick. In addition, variables to hold the persistent value are created, namely x_val and y_val . Furthermore, auxiliary signals for computing the combined value in a tick are added: x_curval and y_curval . Immediate during actions reset these auxiliary current value variables to the neutral value, in this case 0. Another immediate during action is responsible for updating the persistent value of a valued signal with the combined value of the auxiliary current value variable whenever the signal status variable is true, i. e., the signal is present/was emitted in this tick. Note that y_val is declared as an output variable to be observable like y , which was modeled as an output. The emissions of the signals are replaced by two assignments. The first assignment sets the status variable to true with a relative write likewise to the pure signal transformation. The second assignment combines the new emitted value with the auxiliary current value variable using the defined combination function, e. g., $x_curval = x_curval + 3$.

5.2.12 Pre

The *pre* operator (cf. pseudocode in Section 5.3.9 on page 249) of SyncCharts or Esterel allows to access the value or presence status of the previous tick. In SCCharts, *pre* is also syntactic sugar and can be eliminated using M2M transformations. In order to eliminate *pre* and access a value of the previous tick, an auxiliary variable must be introduced to hold the previous value, lets say $_pre_x$ for a variable called x .

Figure 5.2.50 shows the transformation for the *pre* operator. The SCChart

5. Compiling SCCharts

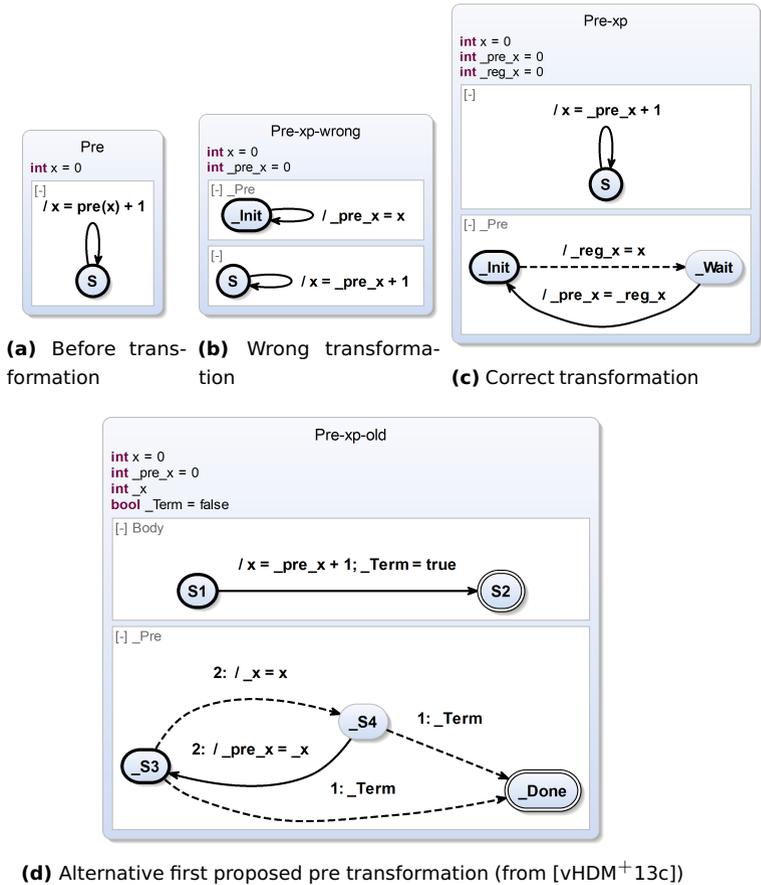


Figure 5.2.50. Pre operator feature expansion transformation, wrong (b), correct (c), and alternative (d)

in Figure 5.2.50a has one variable x . In each tick, other than the initial tick, it increments its value by accessing the previous tick value of x and adding 1 to it. One might expect that the following could then be enough: 1. Replace

5.2. High-Level Compilation

all expression occurrences of $\text{pre}(x)$ with $\text{_pre_}x$ and 2. concurrently update $\text{_pre_}x$ with the value of x at the beginning of each tick so that it can be accessed in the current tick with x 's value of the previous tick. Figure 5.2.50b demonstrates this simple but wrong approach. The problem is that the sequential constructiveness will enforce any assignment which contains $\text{_pre_}x$ to happen after any writing x to $\text{_pre_}x$. Hence, $\text{_pre_}x$ will effectively be a copy of x and not the value of the previous tick. And even more worse: If $\text{_pre_}x$ is contained in an assignment to x , like in the example, then this effectively is a causality cycle which leads to rejection of the SCChart.

To overcome this, another auxiliary variable $\text{_reg_}x$ for the current value is introduced, which is shown in the correct transformation in Figure 5.2.50c. Now, in every tick, $\text{_pre_}x$ is set not to x but to the value of $\text{_reg_}x$ and only after that $\text{_reg_}x$ is updated to x . Sequentiality is essential and makes sure 1. to collect the old value $\text{_reg_}x$ and to store it to $\text{_pre_}x$, 2. to possibly execute any writes to x , and 3. to update $\text{_reg_}x$ with the final value of x of this tick.

Figure 5.2.50d shows the result of our first proposed pre transformation implementation. As can be seen, it did not make use of complex final states, i. e., it did only produce ordinary final states. The downside of this transformation was that keeping track of termination of concurrent region needed to be implemented in the pre transformation a second time, e. g., using a _Term flag, although the complex final state and the abort transformations already do something very similar. According to WTO, we decided to keep track of region termination only in the complex final state transformation and in the abort transformation and to take advantage of re-using this functionality implicitly by producing complex final states in the pre transformation. This further allows to simplify the pre transformation as shown in the result of the current implementation (cf. Figure 5.2.50c).

Another problem of the simple and wrong approach (cf. Figure 5.2.50b) is that the new concurrent region must not prevent the superstate from possibly terminating. To address this, both auxiliary states _init and _wait are possibly set to be final, similarly to the during action transformation (cf. Section 5.2.10 on page 181). _init and _wait are set to final iff the surrounding state has a termination transition which is possibly enabled (cf. Figure 5.2.51) or if it is the root state.

5. Compiling SCCharts

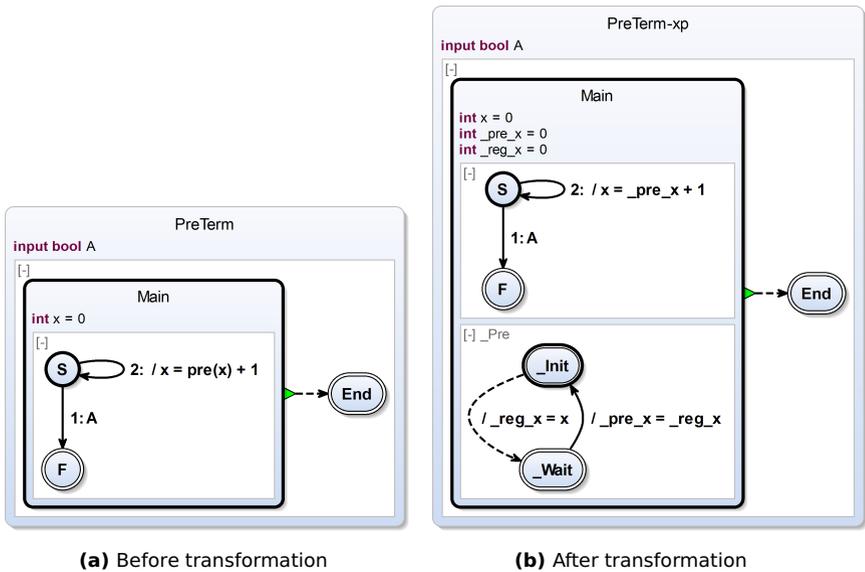


Figure 5.2.51. Pre operator for states with a termination transition

Nested Pre: The pre operator can even be used in a nested fashion to access values of even former ticks than the previous one. In this case, the transformation needs to be applied more than once according to the (finite) maximum nesting level. This is shown in Figure 5.2.52 for the nesting level of two. Figure 5.2.52b shows the result of applying the pre transformation once and Figure 5.2.52c shows the result after applying it twice.

Technically, one can easily detect if, after expanding pre, the feature is still contained in the model. This means the nesting level of pre was greater than one and at least another pass of the pre transformation is necessary. Since the nesting level is finite, only a finite number of passes are necessary for fully eliminating the pre extended pre operators. However, the current implementation does not assume nested pre operators.

5.2. High-Level Compilation

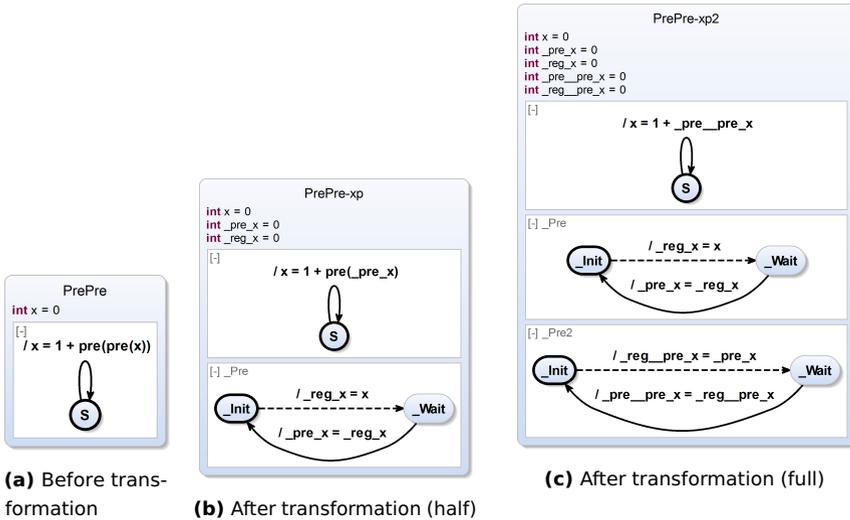


Figure 5.2.52. Nested pre operator feature expansion transformation

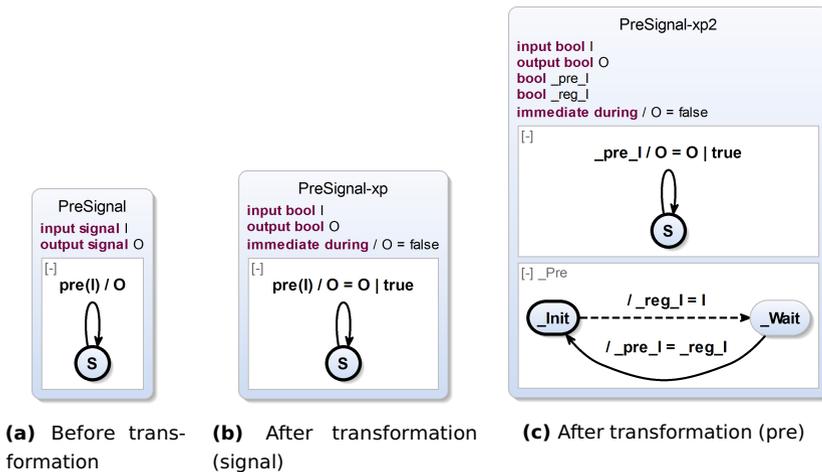


Figure 5.2.53. Pre operator feature expansion with signals

5. Compiling SCCharts

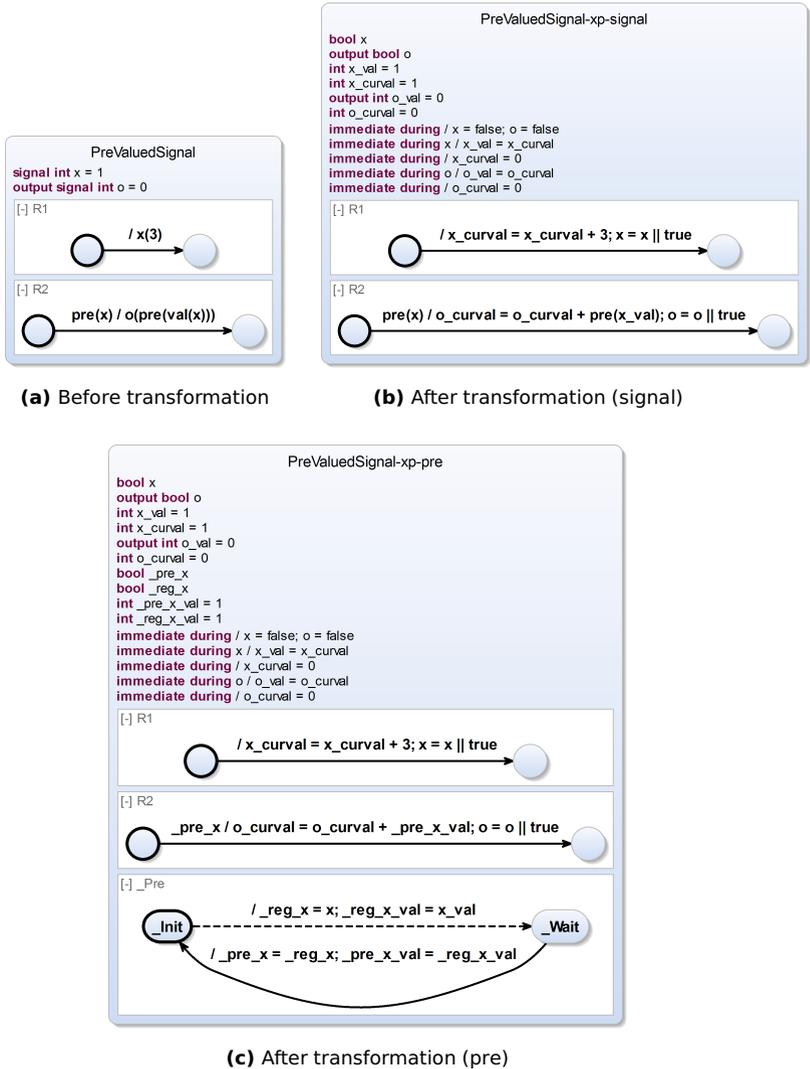


Figure 5.2.54. Pre operator feature expansion with valued signals

Pre with Signals: The pre operator can also be used in combination with signals. Natively, pre operates on variables only and uses auxiliary variables of the same type to store previous tick values. One possibility would be to extend the transformation to also handle signals.

This would require a case differentiation. In case of signals, auxiliary signals would be created in place of auxiliary variables. This complicates the pre transformation. Another possibility is to declare that pre cannot handle signals and hence to enforce that the signal transformation must run before. This design decision was chosen for the SCCharts compiler.

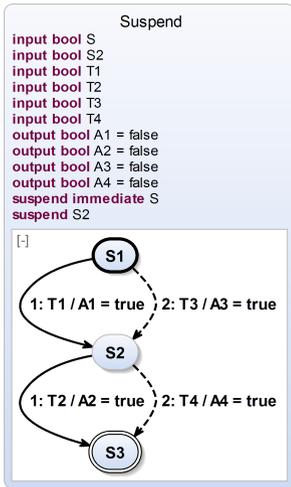
Figure 5.2.53 shows how pre is used in combination with signals. Figure 5.2.53b shows the result after transforming signals into variables and Figure 5.2.53c shows the result after also applying the pre transformation.

Pre with Valued Signals: The pre operator can also be used in combination with valued signals as Figure 5.2.54 illustrates. The model in Figure 5.2.54a has two valued signals: A local integer signal x and an output integer signal o . In the second tick, signal x is emitted with value 3. Hence, in the third tick, $\text{pre}(x)$ will become true and will trigger the transition in region R2. This sets the output signal o to the value of x of the previous tick which was 3. As can be seen in Figure 5.2.54b after applying the signal transformation, again there are two variables for each signal: Variable o represents the presence status of the original signal o and o_val represents the value of the original valued signal o . This means o will become true in the third tick, when the value of $\text{pre}(x_val)$ is assigned to o_val . The pre transformation itself is applied afterwards and produces the SCChart shown in Figure 5.2.54c. It produces a new region $_Pre$ where the pre values of the presence status ($_reg_x$) and of the value of x ($_reg_x_val$) are calculated, memorized, and written back in the next tick to dedicated $_pre_*$ variables that are used in the other regions in place of $\text{pre}(*)$.

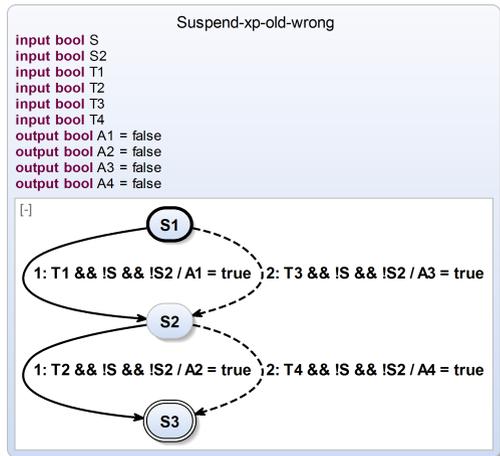
5.2.13 Suspend

The *suspend* extended feature (cf. pseudocode in Section 5.3.10 on page 251) allows to freeze the internal behavior of a state upon some trigger. This feature is borrowed from SyncCharts and Esterel.

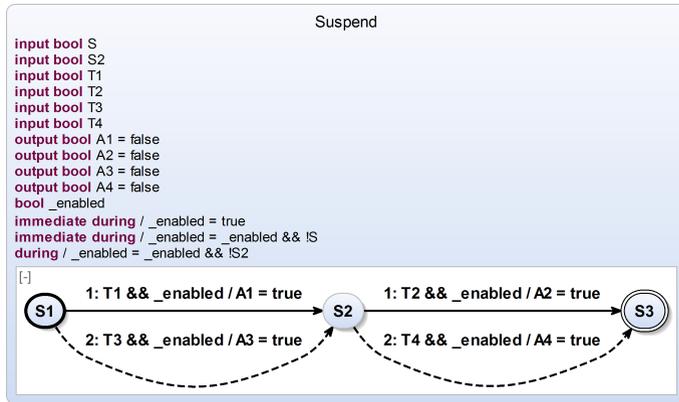
5. Compiling SCCharts



(a) Before transformation



(b) Wrong transformation (adapted from [vHDM⁺13c])



(c) Correct transformation

Figure 5.2.55. Suspend feature expansion transformation

5.2. High-Level Compilation

One approach for implementing this feature is to add guards to all internal actions in conjunction to existing guards. This was the first proposed approach [vHDM⁺13c]. The additional guards are the negation of the suspension trigger as illustrated in Figure 5.2.55.

The SCChart in Figure 5.2.55a has two suspensions: One suspend with trigger S and another immediate suspend with trigger S2. Figure 5.2.55b shows the transformed result. However, this is incorrect for immediate transitions, which now are suspended immediately even for a (delayed) suspend such as S2. Also, as can be seen, the suspension triggers have to be replicated to every transition.

This violates the WTO principle: Triggers and their negation may need to be evaluated multiple times if a state has more than one outgoing transition.

To overcome the drawbacks of the first proposal, a variant where an auxiliary flag is used for each suspension is now proposed. This flag is set to true initially in every tick with an absolute write. Then, each during action updates it in a conjunction with the negated suspension trigger. The update during action for a suspension is immediate iff the suspension is immediate. The evaluation of the suspension trigger now takes place within the during action only once for a tick to meet the WTO principle. The flag can then be used to additionally guard inner actions and transitions similar to the first proposed transformation.

Figure 5.2.55c shows the result of the current proposed transformation for suspend and immediate suspend using a shared boolean auxiliary variable to implement the `_enabled` flag. Note that the enabled flag needs to be added to immediate and non-immediate transitions (and actions). Because the flag is initialized with true in every tick, it does not harm for non-immediate suspensions where the resulting during actions, which listen to the original suspension triggers, may set the flag to false only in non-initial ticks (e. g., S2).

5.2.14 Count Delay

A *count delay* (cf. pseudocode in Section 5.3.11 on page 252) is used as a modifier for a trigger to wait for the n-th occurrence of this trigger to possibly take a transition. Count delay is an extended feature borrowed

5. Compiling SCCharts

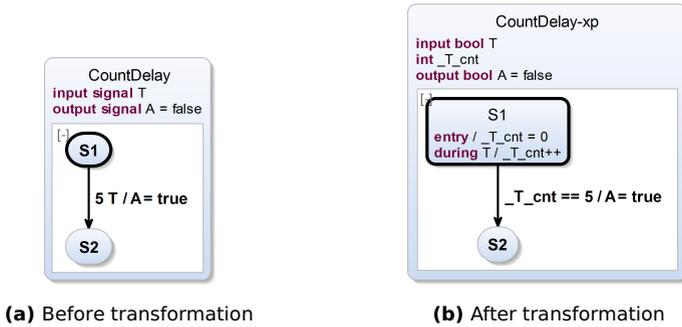


Figure 5.2.56. First proposed count delay feature expansion transformation (from [vHDM⁺13c])

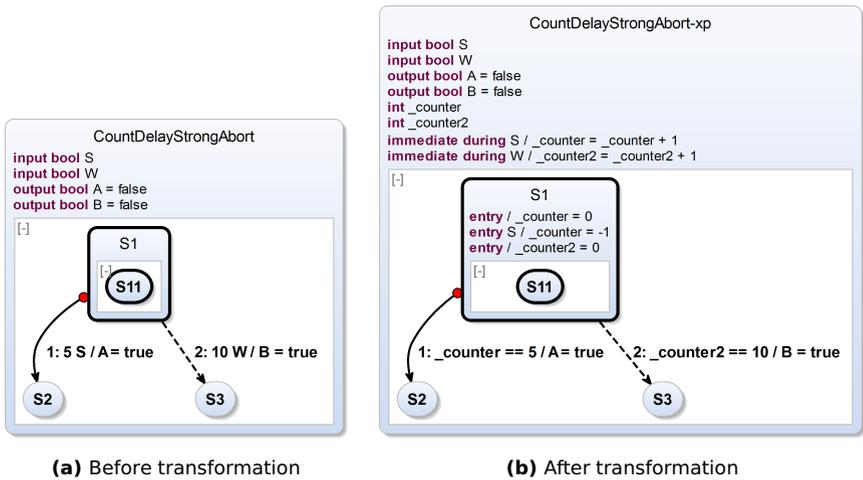


Figure 5.2.57. Current count delay feature expansion transformation also handling strong abort transitions

from SyncCharts. Basically, a count delay is replaced by an explicit counting variable and a concurrent counting conditioned by the original transition

trigger. More specifically, for each count delay trigger T , a counting variable $_T_cnt$ is created in the parent superstate. The counting variable $_T_cnt$ is reset in the source state of the corresponding transition using an entry action. For counting the occurrences of T , a during action is introduced: $T / _T_cnt++$. It is immediate iff the dedicated transition is immediate.

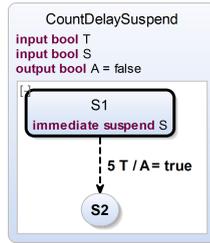
Figure 5.2.56 illustrates our first proposed transformation [vHDM⁺13c] for the count delay feature. The small example (cf. Figure 5.2.56a) waits for the 5th occurrence of trigger T . Note that the occurrences not necessarily need to take place in consecutive ticks. The transformation result is shown in Figure 5.2.56b. The auxiliary counting variable $_T_cnt$ is declared in the superstate. When entering state $S1$ this counting variable is reset to 0. The during action is declared within $S1$. Whenever the counter reaches 5, meaning trigger T had occurred five times since state $S1$ was entered (not necessary in consecutive ticks), then the transition is triggered. Hence, the original transition trigger was replaced by a new trigger that only checks if the counter has reached the declared limit.

Our first proposed transformation could not handle strong aborts. As Figure 3.2.5 on page 60 showed, a strong abort must not be triggered from within a state that may be preemptively aborted. Hence, counting of the trigger which may lead to abort of the state cannot be done within the state in case of a strong abort. Our current count delay transformation implementation now handles this case correctly by counting one hierarchy level up in the parent superstate. This is illustrated in Figure 5.2.57. The new transformation also correctly handles a delayed *and* an immediate variant of the count delay.

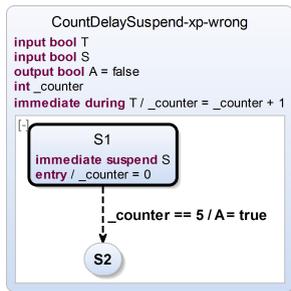
Figure 5.2.58 shows the combination of suspend and count delay. It basically illustrates why suspend should be transformed before count delays are. The reason in essence is that otherwise the entry action, which resets the counter, may be preempted which would be wrong.

Figure 5.2.58a shows the example with an immediate suspend and a count delay feature before expansion. Figures 5.2.58b and 5.2.58c show the result of first transforming the count delay and then transforming the suspend which is wrong. Figures 5.2.58d and 5.2.58e show the result of first transforming the suspend and then transforming the count delay which is correct. The small but crucial difference between Figures 5.2.58c and 5.2.58e

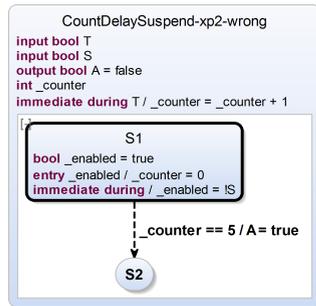
5. Compiling SCCharts



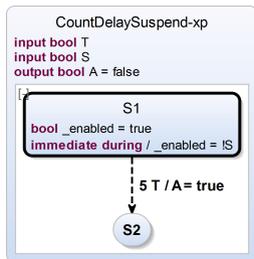
(a) Before transformation



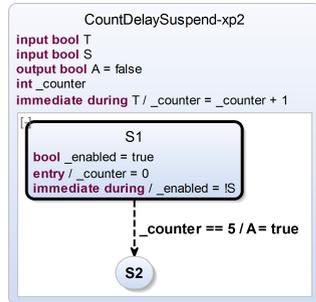
(b) Wrong transformation (half)



(c) Wrong transformation (full)



(d) Correct transformation (half)



(e) Correct transformation (full)

Figure 5.2.58. Count delay combined with suspend feature expansion: Need to transform suspend first

5.2. High-Level Compilation

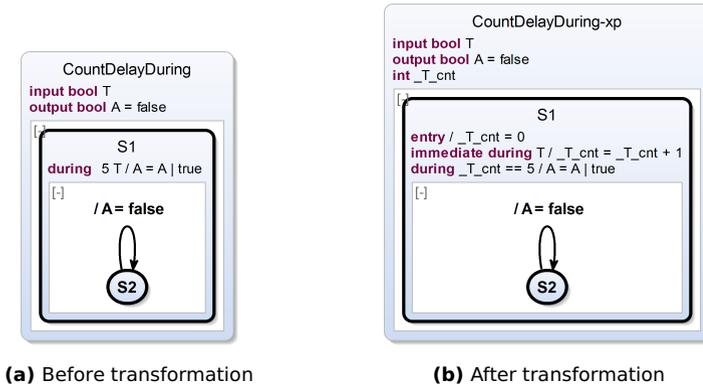


Figure 5.2.59. Count delay combined with during action feature expansion transformation possibility

is that in Figure 5.2.58c also the entry action is guarded by the `_enabled` flag simply because the (immediate) suspend also was applied to the already expanded count delay counter reset part. However, the reset of the counter must happen independently from any suspend. Actually, the reset action as well as the counting is more associated with the superstate.

To account for this and to enforce that the suspend transformation will happen before the count delay transformation, the count delay transformation is declared to not handle the suspend feature (cf. Figure 4.1.5 on page 99).

General Count Delay Triggers In the current KIELER SCCharts implementation only transition triggers allow to specify a count delay. However, in general, the count delay feature could be desirable for other triggers too. Figure 5.2.59 shows an example for a during action with a count delay and one possible expansion.

5. Compiling SCCharts

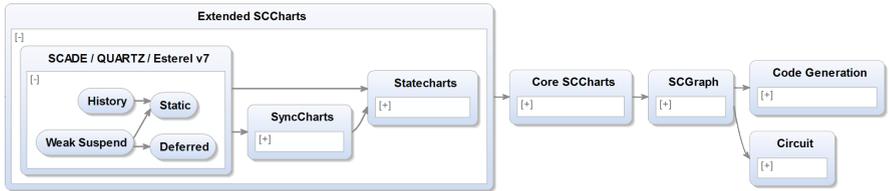


Figure 5.2.60. SCADE / QUARTZ / Esterel v7 feature transformations in KiCo compiler selection: History, Static, Deferred, and Weak Suspend

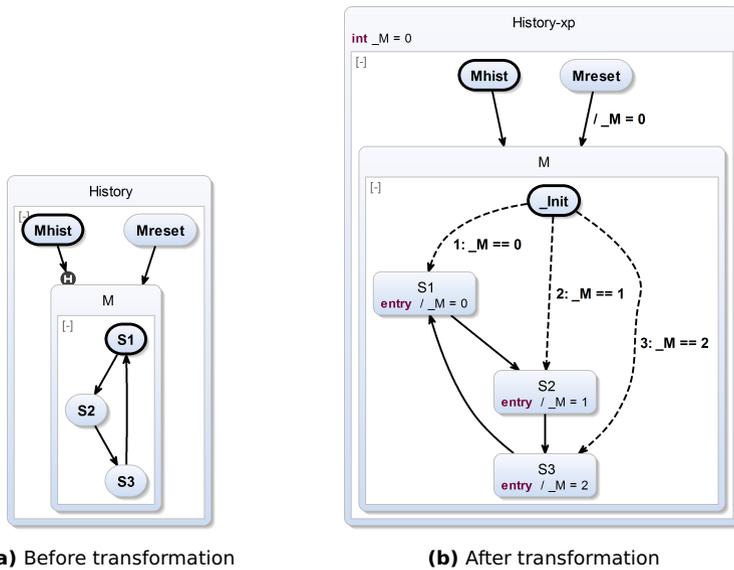


Figure 5.2.61. History feature expansion transformation

5.2.15 History

When inner regions of a superstate are entered, usually each region starts in its one, distinct initial state — no matter which state was active when the

5.2. High-Level Compilation

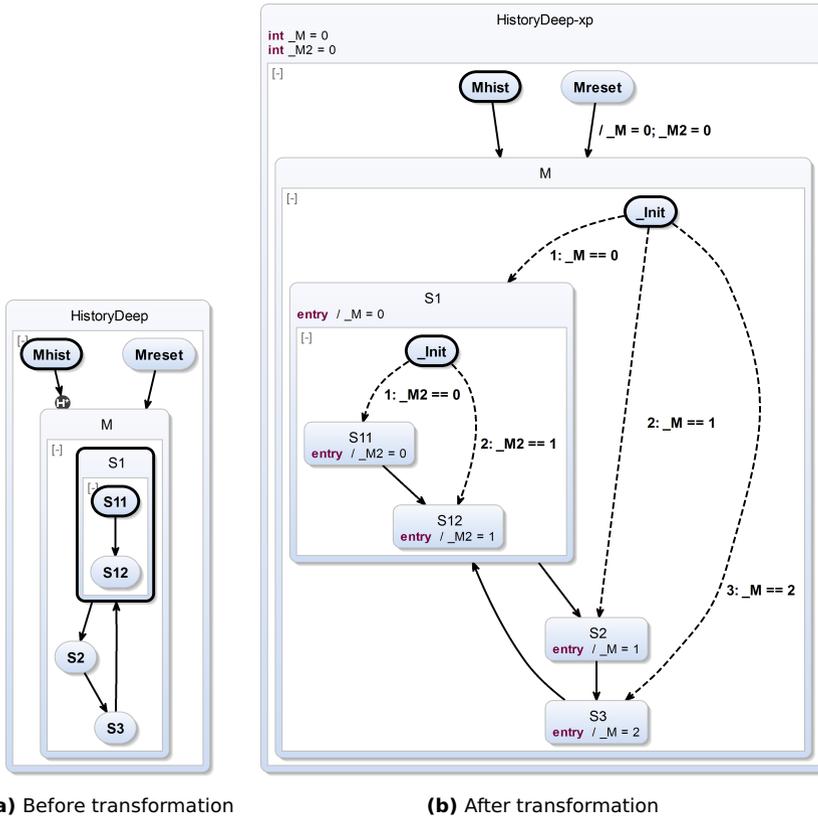


Figure 5.2.62. Deep history feature expansion transformation

superstate may have been left earlier. In contrast, if re-entering a superstate via a history transition, then each region will restart in the state it was in before the parent state of these regions was left. The initial states only matter the first time the superstate and its regions are entered.

History transitions (cf. pseudocode in Section 5.3.12 on page 253) require to remember state information for inactive parts of a model during runtime.

5. Compiling SCCharts

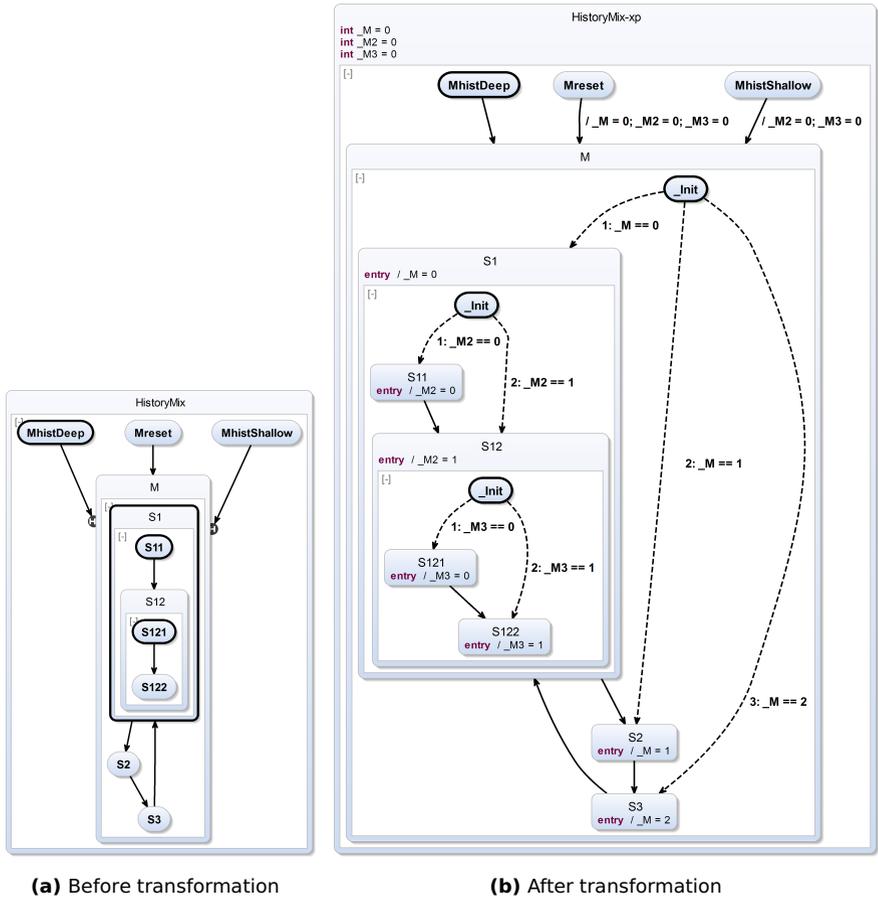


Figure 5.2.63. Mixed deep history and shallow history features

Hence, on the one hand, the use of this feature increases the complexity of the model and it should thus be used with care. On the other hand, this extended feature hides an enormous amount of complexity if this behavior must be achieved and one does not have to model it explicitly

5.2. High-Level Compilation

with Core SCCharts features. There are two variants of history transitions proposed for SCCharts: 1. A *shallow history* feature where state information is only remembered and possibly recovered only in the hierarchy layer of the direct child regions of the target superstate. 2. A *deep history* feature where the state information is remembered on all nested inner hierarchy layers of all regions contained by the superstate. The deep history feature is considered the default for SCCharts. The proposed transformation exposes the complexity of this feature and is illustrated in Figure 5.2.61. Note that Mreset is not reachable and the incoming transitions are just meant to keep the model simple. Actually, an incoming history transition only makes sense if there is also a way to leave the superstate in question which is not part of this and the following illustrations.

If a superstate M should be re-entered via a history transition, then outside of M , i. e., in its parent superstate, an auxiliary state variable $_M$ needs to be added. $_M$'s value will encode the state that M was in last. This is the state that M should start in when entered via a history transition. This could be encoded as an integer where 0 encodes the original initial state. Consequently, all non-history transitions that target state M need to reset $_M$ to 0. All history transition leave the value of $_M$ untouched. An auxiliary initial state $_init$ is added and depending of the value of $_M$ it immediately dispatches to the desired state. Whenever the state changes, clearly $_M$ must be updated in order to “remember” the last active state. This is done by adding entry actions to all states that set $_M$ to the unique number that represents this state, e. g., 0 for the original initial state.

If state M is not a superstate at all or M has only (stateless) entry/during/exit actions then using history transitions to target M makes no sense as there is nothing to remember. If M is a superstate but all regions of M only contain simple states then there is no difference between shallow history and deep history transitions targeting M . However, if M has at least one region with at least one superstate then there is a difference. The shallow history transformation will ignore these inner superstates and proceed as if all inner superstates are entered by a non-history transition.

In contrast, the deep history transformation will recursively traverse all inner superstates and apply the transformation to them too. Figure 5.2.62 illustrates the transformation result for the deep history feature expansion,

5. Compiling SCCharts

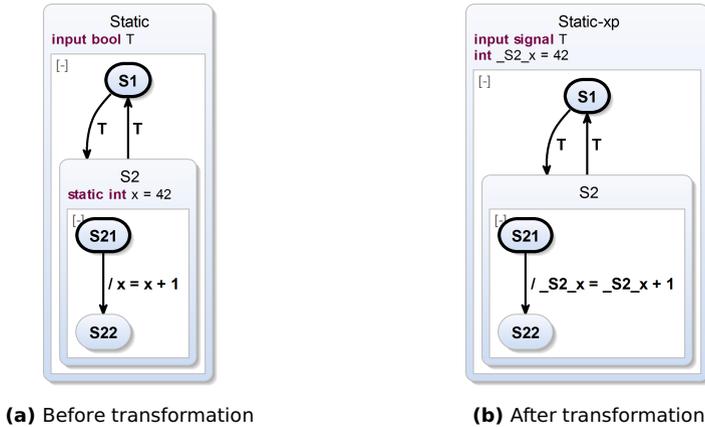


Figure 5.2.64. Static feature expansion transformation

where S1 is such an inner superstate of a region of M. As shown, for all inner superstates there needs to be another new auxiliary state variable, here `_M2`. Also, this needs to be reset when entering M via a non-history transition. `_M2` will then be updated when this inner superstate changes its state, again using an entry action. When (re-)entering S1 then the value `_M2` decides where to go, immediately.

History and deep history transitions can even be mixed. Figure 5.2.63 shows an SCChart with a three-level-deep superstate M that is entered via a shallow history, via a deep history, and via a non-history transition. When entering M via the shallow history transition then for all inner hierarchy layers the auxiliary state variables `_M2` and `_M3` need to be reset but not `_M`, which is on the level that is remembered. The rest follows from the transformations as described above.

5.2.16 Static Variables

Variables declared in a superstate only live in the scope of this superstate. When the superstate is re-entered the variables are possibly re-initialized.

Similarly to the history feature for states, there is a feature to persist the value of variables and prevent their re-initialization: *Static variables* (cf. pseudocode in Section 5.3.13 on page 255).

In contrast to history, this is a quite lightweight and not complex feature and transformation. Static variables of a superstate are uniquely renamed according to their scope. Their declaration is raised to the root state. Also, their initialization is raised to the root state.

Figure 5.2.64 illustrates the static variable feature expansion. State S2 in Figure 5.2.64a has a static variable x , which is initialized to the value 42. x can be incremented internally inside S2. The transformation result shown in Figure 5.2.64b clarifies the semantics of the static variable feature. x is renamed to $_s2_x$ and its declaration is moved to the root state of the SCChart. Also, the initialization to 42 is done in the root state. All read and write accesses within S2 must be updated to access the renamed variable $_s2_x$. Note that although $_s2_x$ is visible globally after transformation, it should not be accessed outside its original scope of S2 if scoping was correct before the transformation was applied.

Note that if one would like to allow *static (valued) signals* then there should be a not-handled-by dependency from signals to static such that the static feature gets eliminated first.

5.2.17 Deferred

Deferred transitions (cf. pseudocode in Section 5.3.14 on page 256) can be used to enforce a tick boundary and to prevent instantaneous behavior of the target state as introduced in Section 3.2.15 on page 73. Likewise to history and suspension, this is a rather complex extended feature.

It is no option to statically replace all immediate transitions by non-immediate transitions and do the same for actions. Deferring is a dynamic feature that is only enabled if a state is entered by a deferred transition. If the same state is entered by some non-deferred transition it must still be able to react instantaneously (if it had possible immediate behavior or immediate outgoing transitions).

The transformation creates an auxiliary flag variable, which is initialized with false. This is set to true by all deferred transitions that target a state.

5. Compiling SCCharts

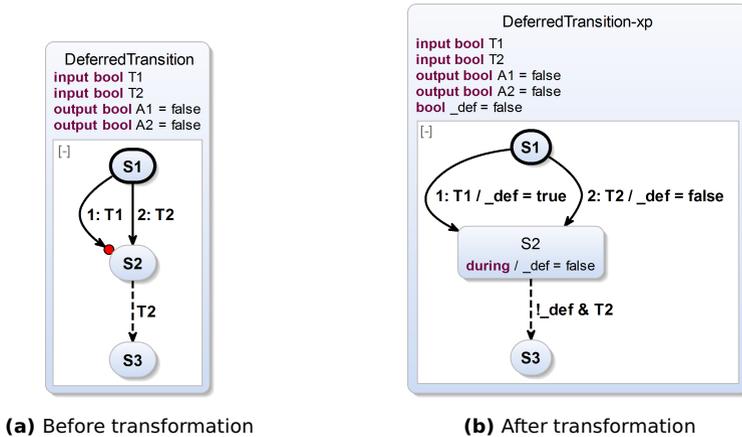


Figure 5.2.65. Deferred feature expansion transformation

When this state is entered, a (delayed) during action resets the flag in the following tick.

There are two variants of deferred transitions, a *shallow* (deferred) variant and a *deep* variant. The deep variant is syntactically indicated by an additional asterisk in the SCCharts diagram. In the deep variant, the flag is added deeply to all internal immediate actions and transitions and also to all outgoing immediate transitions. In the shallow variant the flag is only added to the outgoing immediate transitions of the target state.

The deferred transformation is illustrated in Figure 5.2.65. A simple example is shown in Figure 5.2.65a where the target state S2 is a simple state that has an immediate outgoing transition. This immediate transition is blocked in the tick when S2 is entered via the deferred transition. It is not blocked in case it is entered via a non-deferred transition.

The transformed version makes the semantics of deferred explicit and is shown in Figure 5.2.65b. A boolean auxiliary deferred flag `_def` is added to the superstate of S2 and initialized with false. It is set to true when entering S2 via a deferred transition. A delayed during action resets the flag to false again in the next tick after entering S2. As long as the flag is true,

5.2. High-Level Compilation

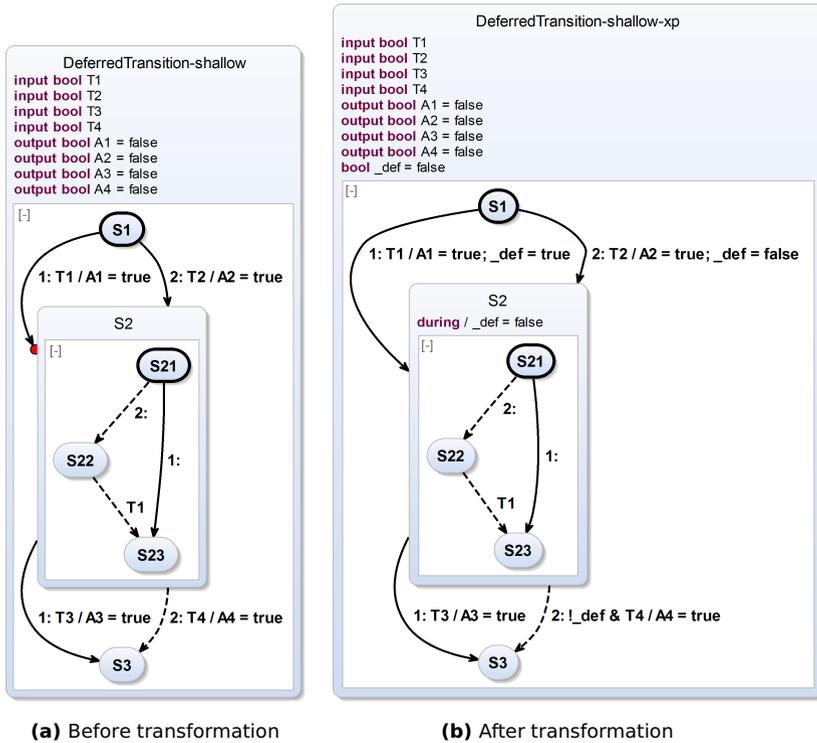


Figure 5.2.66. Shallow deferred feature expansion transformation for a state with possibly immediate behavior

which is the case just in the tick when S2 is entered, the immediate outgoing transition to S3 is disabled and cannot be taken in the same tick. It may be taken in consecutive ticks when the flag has been reset to false again. If S2 is entered via the non-deferred transition, then the flag is set to false and the immediate outgoing transition could possibly be taken if its trigger T2 is true. Note that the flag must be actively set to false because it cannot be guaranteed that the superstate has not been aborted immediately after it has been entered via a deferred transition earlier in the execution. Hence,

5. Compiling SCCharts

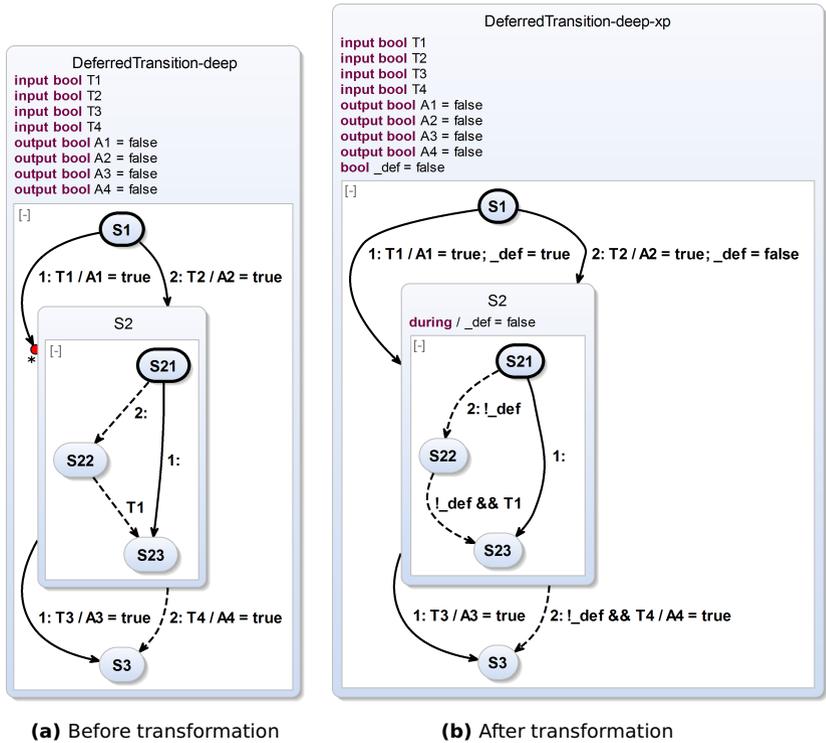


Figure 5.2.67. Deep deferred feature expansion transformation for a state with possibly immediate behavior

the deferred flag can possibly still be true which would lead to erroneous behavior.

Figures 5.2.66 and 5.2.67 show another example of the deferred feature expansion where the distinction between shallow and deep deferred transitions is exposed. In the deep deferred transition transformation, also internal immediate behavior of S2 is blocked using the the auxiliary `_def` flag. Note that the auxiliary flag is only added to immediate transitions. The same would hold for immediate actions in case of a deep deferred transition.

Non-immediate transitions or non-immediate actions will anyhow never be executed when S2 is entered so guarding them is not necessary.

One could think about defining (deep) deferred using suspend. Although this should be at least partly possible it might not be a good idea because this would increase computational complexity unnecessarily since non-immediate transitions will be guarded as well. As immediate outgoing transitions would not be blocked by suspension, the transformation must anyhow take care of them by itself. Hence, implementing the deferred feature separately is reasonable.

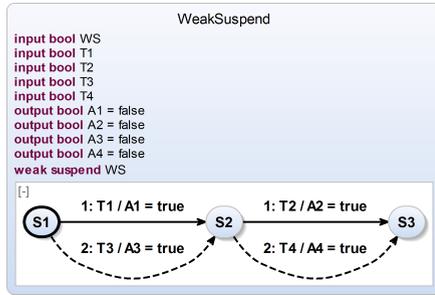
5.2.18 Weak Suspend

The *weak suspend* (cf. pseudocode in Section 5.3.15 on page 257) is a subtle special extended feature that is borrowed from QUARTZ and Esterel v7. The basic idea is that the weak suspend behaves like the suspend but with the following difference: It allows a “last will” for the current tick if the weak suspend is triggered, but as with the “strong” suspend, it, in the next tick, starts execution in the same state where control started for the previous tick when the weak suspension took place. We here discuss how to implement weak suspend for sake of completeness and to show that it can be done. However, as it turns out, the transformation for weak suspend is quite complex. Furthermore, even though weak suspend is useful, e. g., to implement multi-clock design, it appears to be rarely used. We therefore consider this as an optional language feature that may or may not be implemented by an SCCharts compiler.

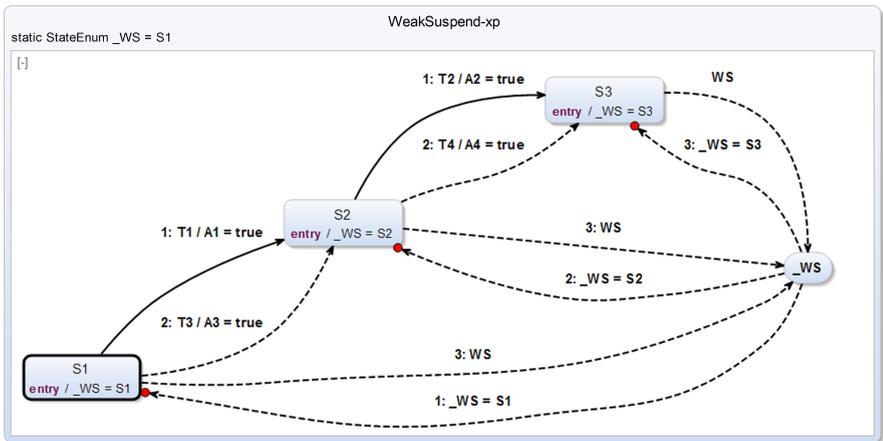
Figure 5.2.68 exposes the weak suspend feature expansion. In Figure 5.2.68a, three internal states S1, S2, and S3 exist together with immediate and delayed transitions connecting them. S3 is a final state. A weak suspend is triggered by a boolean WS.

Figure 5.2.68b presents our first expansion of weak suspend, which still had limitations. The stateEnum variable is an integer indicating which state the control should start in the next tick. It is updated using entry actions. However, these entry actions should not execute if the weak suspend is triggered; in this case we do want to execute the “last will”, but not to update the stateEnum variable.

5. Compiling SCCharts



(a) Before transformation



(b) After (obsolete) transformation (from [vHDM⁺13c])

Figure 5.2.68. Weak suspend expansion transformation

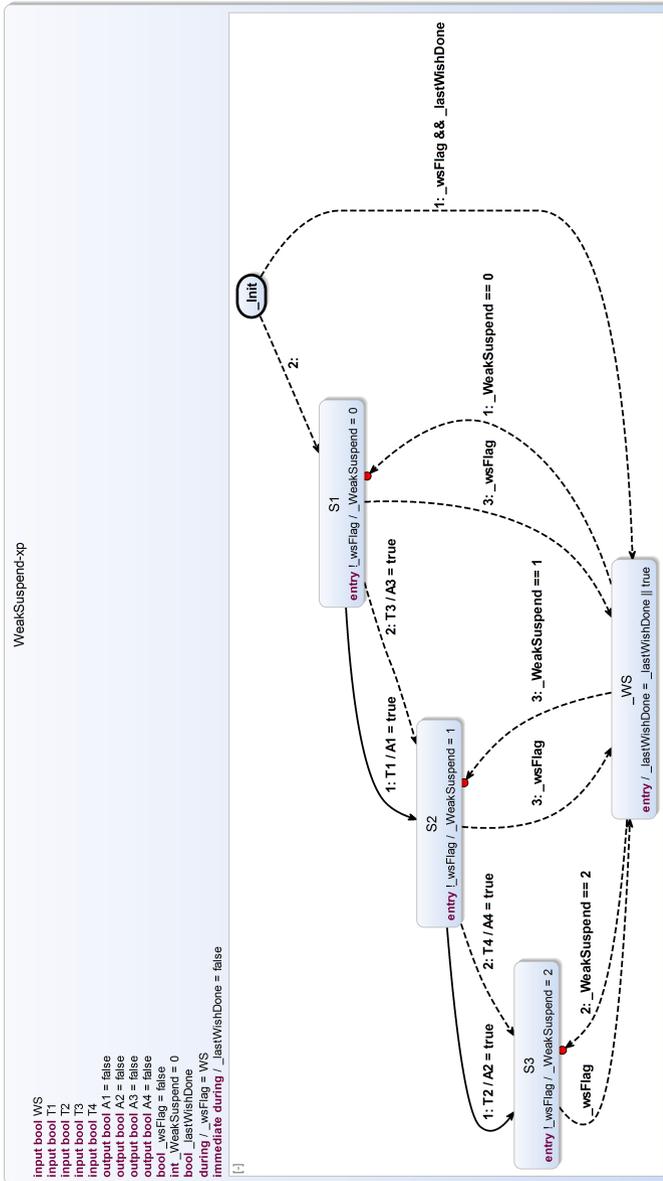


Figure 5.2.69. After current (evolved) weak suspend transformation

5. Compiling SCCharts

The central `_WS` state is where to go after performing the “last will” from every state with lowest priority in order to make sure to complete the “last will”. Furthermore, the `_WS` state is transient. From this state we immediately branch to the desired state and enter it “hidden” utilizing a (shallow) deferred transition. Due to the “hidden entry”, in the next tick, it seems as if we already were in the state before (we actually already are but just prevented any immediate behavior because of the deferred entry) and are able to execute (delayed) transitions as usual.

This initially proposed transformation has the following limitations: 1. It does not work for hierarchy because of when re-entering a superstate it will not distinguish between executing the “last will” and the “hidden re-entry”. 2. Making `S3` final would be problematic because a “last will” might (i) involve actions after a termination transition triggered by such a final state or (ii) the scope of the weak suspend should not be left. 3. The weak suspend trigger `WS` is used directly. This makes it impossible to distinguish between the immediate and a possible delayed variant. The delayed variant cannot be expressed.

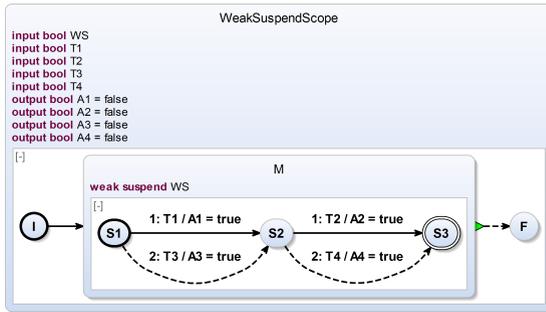
To overcome these flaws, an enhanced but slightly more complex expansion for weak suspend is proposed as shown in Figure 5.2.69. The additional `_wsFlag` flag is introduced to solve problem 3. In each tick, it is set by an immediate or a delayed during action according to the immediate or delayed weak suspend. This also satisfies the WTO principle.

Note that even in the *immediate* case of the suspend, the during action which sets the `_wsFlag` to the suspend condition does not need to be a relative write, even if it may occur in the same tick when the initialization to false occurs.

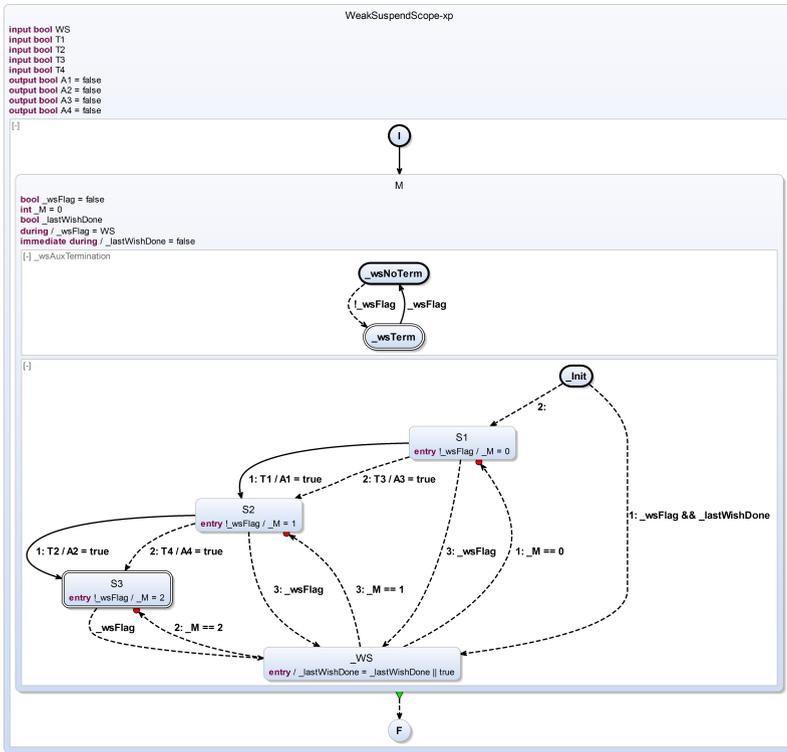
The reason is that the initialization is sequentially ordered before entering the state that contains the (immediate) during action. Further note that the initialization to false is required for the non-immediate weak suspend case. In the immediate weak suspend case, it would be sufficient to omit the (extra) initialization which is overwritten immediately by the weak suspend trigger.

The transformation still has the auxiliary `_WS` state but this can now be entered in two ways. An additional initial state `_Init` is added for every region and an additional `_lastWishDone` flag is introduced.

5.2. High-Level Compilation



(a) Before transformation



(b) After transformation

Figure 5.2.70. Weak suspend expansion and scoping

5. Compiling SCCharts

The purpose of this flag is to distinguish whether 1. the superstate is entered while performing the “last will” (`_lastWishDone == false`) or 2. the “hidden re-entry” is taking place in order to set back the control point for the next tick (`_lastWishDone == true`). This flag is defined on every layer and each auxiliary `_WS` state will set it to true. Hence, the ordinary immediate behavior is only executed if the superstate is entered and additionally either the `_wsFlag` is false (which means the weak suspend does not hold) or the `_lastWishDone` is still false on this layer for this tick. However, if it has already been executed and the weak suspend holds, then the `_lastWishDone` flag is true and the transition from state `_Init` to state `_WS` will do the “hidden re-entry” on this layer. The (shallow) deferred transitions prevent immediate cycles after performing the “hidden re-entry”. This solves problem 1.

Weak Suspend and Scoping: In order to not run into problem 2 and be able to perform a “last will” that involves actions after a possible termination on the upper hierarchy level, the transition of a final state to the `_WS` state is removed if the final state’s region is not a direct child region of the state in which the weak suspend is defined in.

Such a region should not terminate if the weak suspend trigger holds because it defines the scope of the weak suspend. This is illustrated by the example in Figure 5.2.70. Another auxiliary region is created on the scoping level of a weak suspend if this state may terminate with a termination. This region prevents termination of the superstate in case the weak suspend trigger holds. Otherwise, it permits a possible termination by being in a final state.

Note that removing transitions from a final state to the `_WS` state if the final state is in some inner region that may be part of a “last will” is a semantic design choice. It has the consequence is that if control ends up in a (concurrent) final state and the termination is not triggered then this region is terminated and, weak-suspending, it will not cause any further actions. Another and different option would be to reconsider the priorities for complex final states and leave this transition in. It is future work to evaluate these both or other design choices for a “last will” that includes taking terminations.

5.2. High-Level Compilation

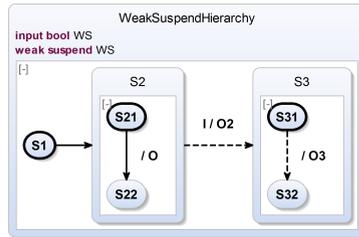


Figure 5.2.71. Weak suspend and hierarchy

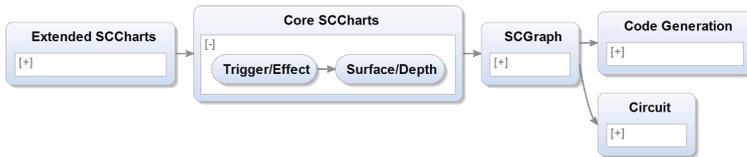


Figure 5.2.72. Normalization transformations in KiCo compiler selection: Trigger/Effect and Surface/Depth

Weak Suspend and Hierarchy: When hierarchy is involved (cf. Figure 5.2.71) the “last will” inside may be executed but the deferred transitions prevent any immediate cycles. However, proceeding with executing the “last will” may happen even by an outer abort or a termination (within the scope of the weak suspend). Later, the (shallow) deferred transition is taking control back to where it has started inside the aborted or terminated inner superstate. Recall that the shallow variant only prevents immediate outgoing transitions but allows inner behavior. This inner behavior is necessary to restore the correct control point by the “hidden re-entry”.

5.2.19 Normalization

Figure 5.2.72 shows the Core SCCharts transformations Trigger/Effect and Surface/Depth (cf. pseudocode in Section 5.3.16 on page 260). These transformations help to further ease down-stream compilation and together constitute the *normalization* of Core SCCharts which reduces the necessary

5. Compiling SCCharts

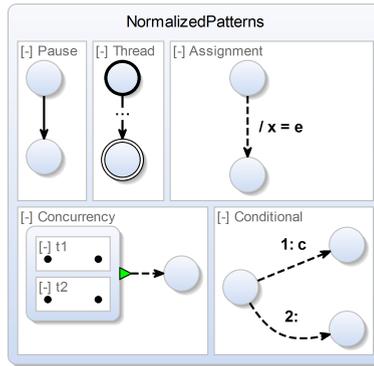


Figure 5.2.73. The five allowed patterns for Normalized (Core) SCCharts ease a direct mapping to SCG elements (cf. Figure 5.0.2 on page 115).

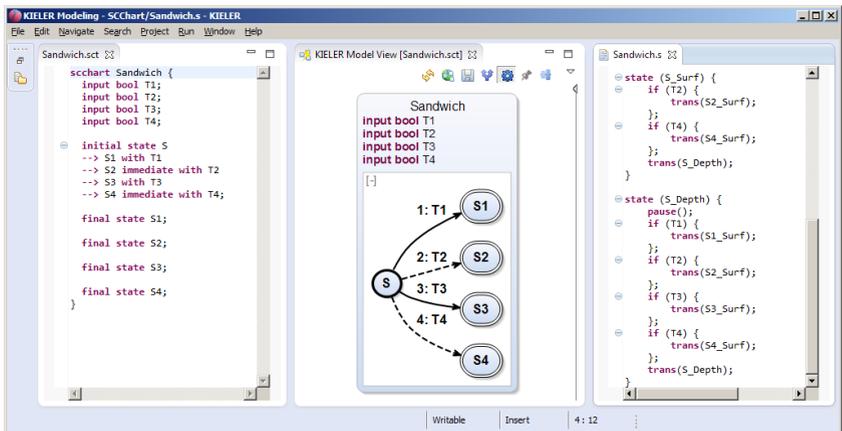


Figure 5.2.74. Sandwich SCCharts example and S code generation (originally from SyncCharts) which is the predecessor of the current normalization transformation

patterns to not more than *five*, as discussed Section 5.0.1 on page 113. These patterns are recapitulated in Figure 5.2.73.

5.2. High-Level Compilation

Pause: When entering a pause state, the one and only outgoing (delayed) transition is taken in the next tick. This represents the tick boundary. The one and only transition has no actions and an implicit true-trigger.

Thread: A thread begins with exactly one initial state and ends with one or more final states. In between (indicated by “...” in Figure 5.2.73), only pause, concurrency, assignment, and conditional are allowed patterns.

Concurrency: Concurrency implicitly creates hierarchy with a fork and join. The fork is expressed as a hierarchical state with concurrent regions (threads). The join is expressed as an unconditional and immediate termination transition with no action.

Assignment: The assignment is the only pattern that expresses an action. It never has a trigger and always executes immediately.

Conditional: The conditional pattern is a state with exactly two outgoing immediate transitions where one of them carries a trigger *c*. The one with the trigger has highest priority. It expresses the if branch. The other has only the implicit true-trigger, the lowest priority, and it expresses the else branch.

If a Core SCChart only uses these five patterns, it is called a *Normalized Core SCChart* or *Normalized SCChart* in short.

History of Normalization

The normalization transformation step evolved from the SyncCharts to S code generation step, which is part of the SyncCharts compiler released together with KIELER version 0.8.0 in 2012 (cf. Section 5.1 on page 116).

In this predecessor of the normalization transformation step, for each SyncChart state, all outgoing transitions are transformed into conditional goto statements. Between the transition-handling conditional goto statements, the priority could be modified. This is a step that is now separated and done later when scheduling the dedicated SCG elements.

A relevant snippet of the SyncCharts2S code generator is presented in Listing 5.2.1. It reveals that this compiler explicitly dealt with strong and weak preemptions for superstates. The fact that Core SCCharts do not have

5. Compiling SCCharts

```
340         // create a pause instruction only iff no HALT or TERM instruction
341         // halt == no outgoing transition
342         // term == final state
343         if (!state.finalState && !joinInstruction) {
344             // Before pausing, ensure the correct priority for possible preemption after wake up
345             sState.addHighestStrongPrio(state);
346             // Now insert the Pause
347             sState.instructions.add(SFactory.eINSTANCE.createPause());
348         }
349
350         // first handle all strong preemptions
351         for (transition : regardedTransitionListStrong) {
352             sState.addStrongPrio(state, transition);
353             transition.handleTransition(sState);
354         }
355
356         // lower priority (to allow a possible body to be executed)
357         sState.addHighestWeakPrio(state);
358
359         // then handle all weak preemptions
360         for (transition : regardedTransitionListWeak) {
361             sState.addWeakPrio(state, transition);
362             transition.handleTransition(sState);
363         }
```

Listing 5.2.1. Snippet from SyncCharts2S compiler, handling transformation for the depth of a state

any weak or strong preemptions any more but only terminations simplifies the normalization step of the current compiler significantly.

Figure 5.2.74 shows the resulting S code for the Sandwich SCCharts example. S is an intermediate language for generating code. It is introduced in Section 5.4.1 on page 265. A partly adapted and limited version of the SyncCharts2S code generator was used before the normalization step evolved and now is completely done on the level of SCCharts. The S code distinguishes the surface and depth of state S. A state is always entered by transitioning to its surface. In the S code, this is done by using the `trans()` construct. This represents an immediate transition. The explicit `pause()` statement at the beginning of the depth is also visible. Afterwards, the outgoing transition triggers are tested by Conditional constructs in the order of the dedicated transition priorities. If no transition can be taken, the depth calls itself ending up in the `pause()` statement which implements the tick

5.2. High-Level Compilation

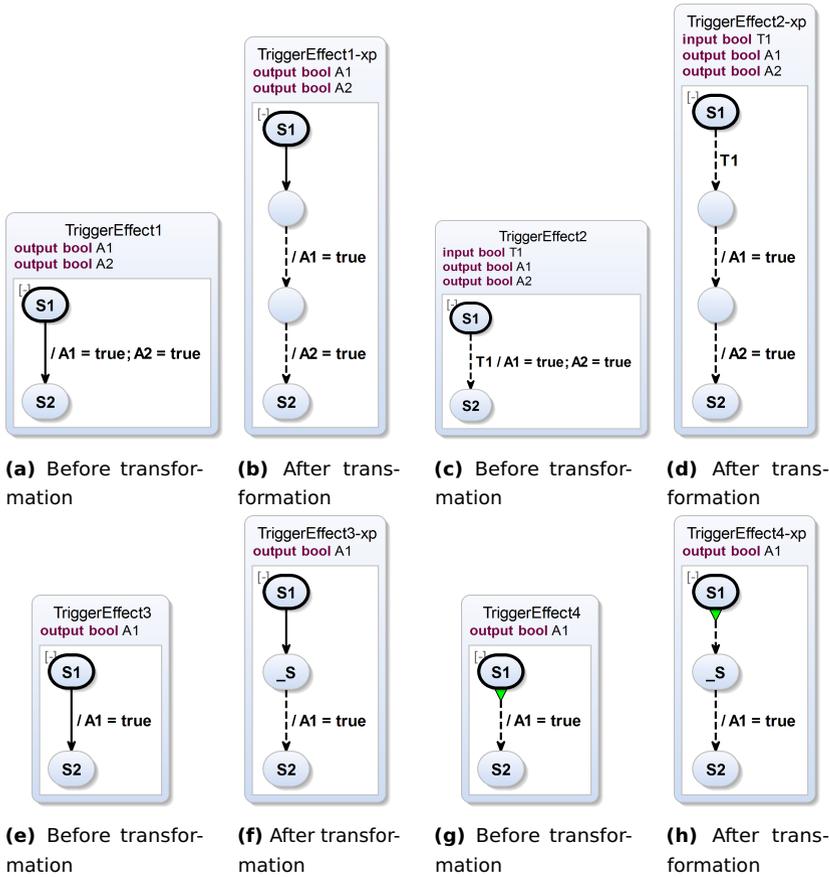


Figure 5.2.75. Trigger/effect normalization transformation examples

boundary. All immediate outgoing transition triggers are already tested in the surface part of the state and also in the order of their priorities.

This can be done similarly by using the five SCCharts patterns from Figure 5.2.73 on SCCharts level, which is the task of the evolved and current normalization transformation. It is described next.

5. Compiling SCCharts

Current Normalization Transformation

The normalization transformation comprises two relevant parts: 1. The triggers are separated from the effects, which is done by the *Trigger/Effect* transformation. 2. The surface testing for outgoing (immediate) transitions is explicitly separated by a Pause construct from the depth, testing for outgoing (non-immediate and immediate) transitions. This is done by the *Surface/Depth* transformation. Both transformations are explained in the following.

Trigger/Effect Transformation

The first step is to separate triggers and/or delays (due to non-immediate transitions) from effects. Various examples are presented in Figure 5.2.75.

Generally speaking, this transformation inserts additional auxiliary states in at least one of the following cases:

1. Immediate and a trigger and one or more effects (c),
2. immediate and no trigger and more than one effect,
3. non-immediate and a trigger and one or more effects, or
4. non-immediate and no trigger but an effect (a+e),
5. a termination (as a special trigger) and one or more effects (g).

The transformation is applied until none of these cases are true any more.

The last case is necessary since a termination pattern results in a fork and a join only, while any effects must be handled separately. The trigger and effect transformation takes care of this separation by introducing auxiliary states for all the above cases.

After the separation of triggers and effects took place, the surface and depth transformation is applied to complete the normalization.

Surface/Depth Transformation

The second step of the normalization is the separation of the surface and the depth of a state. The surface comprises the behavior of the state that is possibly executed when the state is entered, i. e., these are immediate outgoing transitions.

5.2. High-Level Compilation

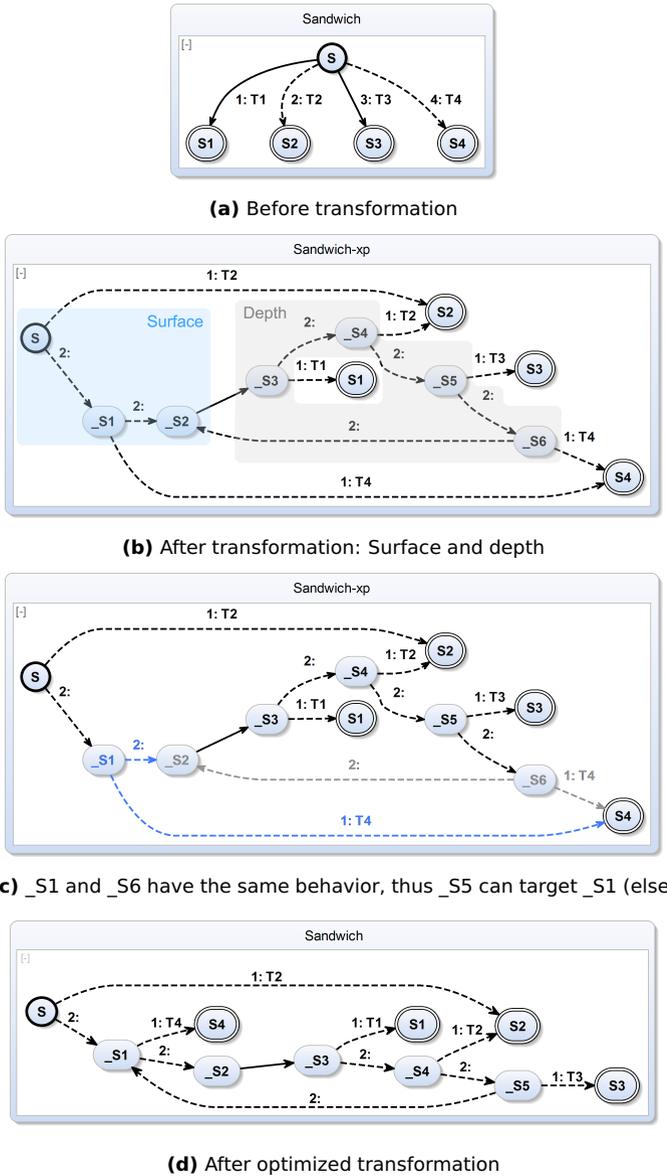


Figure 5.2.76. Surface/depth expansion and optimization [vHDM⁺13b]

5. Compiling SCCharts

The depth comprises the behavior of the state that is possibly executed when the state was entered in an earlier tick and control for the current tick started in this state, i. e., these are immediate and non-immediate outgoing transitions.

Figure 5.2.74 on page 222 showed the translation to intermediate S code from SCCharts. Similarly, the surface and depth transformation uses Conditional constructs to test for a transition or more specifically for its trigger in the if branch. In the else branch, the transition with the next lower priority is tested as another Conditional construct with its own if branch and else branch. This continues for all transitions. The if branch for each transition is connected with the possible actions (from the trigger effects transformation) and finally leads to the destination state of the original transition. The surface and depth are separated by a Pause construct. In the surface, all immediate outgoing transitions are tested in the order of their priorities and in the surface, all immediate and non-immediate outgoing transitions are tested in the order of their priorities.

Figure 5.2.76 demonstrates the surface and depth transformation for the sandwich SCChart example (cf. Figure 5.2.76a). After applying the transformation to the sandwich example, the result makes the surface (blue) and depth (gray) explicitly visible as shown in Figure 5.2.76b. Between the blue and gray part, the Pause construct is clearly visible.

Optimized Surface/Depth Transformation

Figure 5.2.76c shows also the result of the (unoptimized) surface and depth transformation similarly to Figure 5.2.76b. It can be seen that state `_S6` (gray) and state `_S1` (blue) have the same if branch and else branch targets with also the same condition `T4`. Hence, it is valid to re-route the else branch of state `_S5` to `_S1` instead of `_S6`. Consequently, the Conditional construct with state `_S6` is not necessary and can be eliminated without changing the behavior. The resulting optimized Normalized SCChart without `_S6` is shown in Figure 5.2.76d.

This kind of optimization was first studied by Smyth [Smy13] on the level of SCGs. I adapted and implemented it for Core SCCharts. In general, it works as follows for all normalized states S that have a particular state P

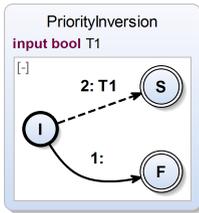
5.2. High-Level Compilation

with an outgoing delayed transition, implementing an SCG Pause pattern, such as `_S2` in Figure 5.2.76b:

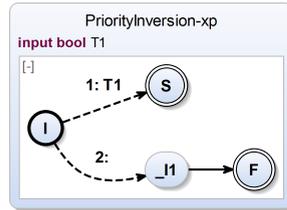
1. Let P be a state with an outgoing delayed transition (`_S2` in the example). If P has two incoming transitions we call these T_{AP} and T_{BP} where T_{AP} is coming from the surface (from `_S1` in the example) and T_{BP} is coming from the feedback of the depth (from `_S6` in the example).
2. Follow T_{AP} back to its source state A (`_S1` in the example) and T_{BP} back to its source state B (`_S6` in the example).
3. Now, if A 's or B 's incoming transition is delayed, stop. Otherwise, compare **all** outgoing transitions T_{AX} of A and T_{BX} of B (both have a transition to `_S4` in the if branch and to `S2` in the else branch) including triggers and effects (in the example, the triggers of the if branches are both `T4`, there are not other triggers of effects to compare).
 - (a) If these are equal then A (`_S1`) becomes the new P state. Incoming transitions to B (`_S6`) now are redirected to the new P (`_S1`), i. e., the former A . B (`_S6`) and all its outgoing transitions T_{BX} including T_{BP} are eliminated. Recursively continue the algorithm with the new state P .
 - (b) If the outgoing transitions including triggers and effects are not equal then the recursion and the optimization algorithm stops.

Apparent Priority Inversion: Figure 5.2.77 shows two interesting example normalizations for two very similar models that only differ by their transition priorities. Consider the model in Figure 5.2.77a. If entering state `I` and the input `T1` is true then we immediately end up in state `S` because the delayed transition is not yet tested. If entering state `I` and the input `T1` is false then we stay in `I`. In the next tick, we do not check for `T1` again because the delayed transition to state `F` has a higher priority. Hence, we definitely take this transition to state `F` if `T1` was not true in the tick before. This behavior is fully reflected by the optimized normalized version of the model shown in Figure 5.2.77b. If we enter `I` and the input `T1` is true then we immediately end up in state `S`. If we enter `I` and the input `T1` is false then

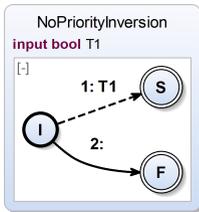
5. Compiling SCCharts



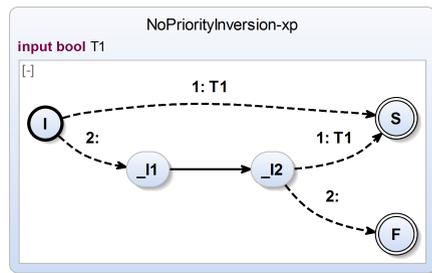
(a) Before transformation



(b) After transformation: It seems that priorities have been inverted but they have not.



(c) Before transformation



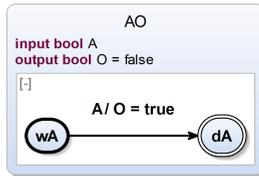
(d) After transformation: No apparent priority inversion in this case

Figure 5.2.77. *Apparent priority inversion* after optimizing normalized SCCharts: At first glance it may surprise that the priorities of Figure 5.2.77b are correct.

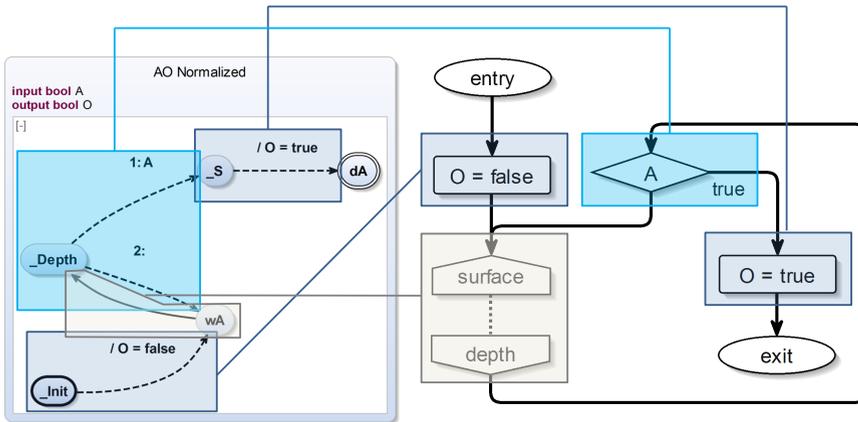
we go to `_I1` where we wait for the next tick. In the next tick we will not check for `T1` and end up in the final state `F`.

The other model, shown in Figure 5.2.77c, starts with inverted priorities compared to the model in Figure 5.2.77a. Here, the optimization cannot be performed as the `_I1` state is not entered from the depth. Hence, the immediate transition trigger `T1` has to be tested twice: One test takes place immediately after entering `I` and the other takes place in the next tick when entering `_I2`.

5.2. High-Level Compilation



(a) AO example as Extended SCChart



(b) Mapping between Normalized SCCharts pattern constructs and SCG elements: Two Assignment constructs (dark blue), one Pause construct (gray), and one Conditional construct (cyan) are mapped into the according SCG elements that constitute the AO SCG.

Figure 5.2.78. AO example as Extended SCChart, Normalized SCChart and SCG control-flow graph representation

5.2.20 Constructing the SCG

Constructing SCGs (cf. Section 2.7 on page 41) from Normalized SCCharts basically is reduced to a straight forward bijective mapping from normalized pattern constructs (cf. Figure 5.2.73 on page 222) to SCG elements according to the SCG mapping table shown in Figure 5.0.2 on page 115.

Figure 5.2.78b gives an example of how the pattern constructs can be

5. Compiling SCCharts

directly mapped to elements of an SCG for the AO SCChart shown in Figure 5.2.78a in its extended version. The behavior of AO is fairly simple. AO starts in the initial state *wA*. In the initial tick it will not react but initialize the boolean output to false. In the second tick and further ticks it may react once to the boolean input *A* being true with taking the transition from state *wA* to state *dA* and setting the output *O* to true. Once it has reached the final state *dA*, AO terminates.

The initialization of *O* as well as the action when taking the transition are represented as an Assignment construct (dark blue). The delayed transition is only tested after entering the state *wA*, and the delay is represented as a Pause construct (gray). The transition is finally represented as a Conditional construct (cyan) with an if branch for taking and an else branch for not taking the transition.

As there is only one thread of control in AO, i. e., there is no concurrency involved in this example. Hence, the resulting SCG does not contain any forks or joins. An example SCG for a simple but concurrent SCChart was given earlier for ALDO (cf. Figure 5.2.4 on page 132).

5.3 Pseudocode for High-Level Transformations

In the following, pseudocode for the transformations (cf. Section 5.2 on page 123) used to expand the SCCharts high-level features is presented.

The pseudocode is based on the SCCharts meta model, see Section 3.4 on page 81. For example, this means for a state *S* that it has a list of outgoing transitions. In the pseudocode this list is referred to as *S.outgoingTransitions*. If an outgoing transition *T* should be attached to state *S* then this is written as *S.outgoingTransitions.add(T)* in the pseudocode. Also this means that for example a state comes with certain properties such as a boolean initial flag which can be set to true or false and which indicates whether the state is an initial state. In the pseudocode this is often abbreviated when such a state *S* is created: `create State S and set initial`. This creates a state *S* and besides this sets its initial boolean flag to true. Note that outgoingTransitions are typically ordered by their priority.

Note that the SCCharts meta model has containment relations, e. g., a

5.3. Pseudocode for High-Level Transformations

state can be contained in only one region. If this state (and not a copy of it) is added to *another* region then it is only contained in this other region. In fact, this is simply a move-operation from one region to another.

To improve readability of the pseudocode, variables carry an index to indicate their type. E. g., a state named A appears as A_s in the pseudocode, where the subscript s suggests that A is a state. Similarly, r stands for a region, t stands for a transition, a stands for an action, v stands for a valued object (i. e., a signal or a variable), and e stands for an expression.

If a transformation pseudocode has the signature $\text{ENTRY}(\text{State } S_s)$ then this means that the function ENTRY is called for every state of the model. The iteration over all applicable model elements is not part of the pseudocode.

5.3.1 Connector

Algorithm 1 Connector Transformation

```
1: function CONNECTOR(STATE  $S_s$ )
2:   if  $S_s.\text{stateType} == \text{CONNECTOR}$  then
3:      $S_s.\text{stateType} := \text{NORMAL}$ 
4:     for all  $S_s.\text{outgoingTransitions}$  as  $T_t$  do
5:        $T_t.\text{immediate} := \text{TRUE}$ 
```

The connector feature transformation is described and illustrated by examples in Section 5.2.4 on page 135. The transformation pseudocode does the following:

If S is a connector state (line 2) then set S to be a normal state (line 3). Additionally, go through all outgoing transitions (line 4) and set them to be immediate explicitly by setting the immediate flag to true (line 5). Recall that as long as transitions are outgoing from a connector state, they are immediate implicitly, no matter the immediate flag of the transition is set to true or not.

5. Compiling SCCharts

Algorithm 2 Entry Transformation

```
1: function ENTRY(STATE  $S_s$ )
2:   if  $S_s$ .entryActions.size > 0 then
3:     State  $F_s$ 
4:     State  $L_s$ 
5:     if  $S_s$ .isFinal then
6:       create Connector  $C_s$  in parent region of  $S_s$ 
7:       for all  $S_s$ .incomingTransitions as  $T_t$  do
8:          $T_t$ .targetState :=  $C_s$ 
9:          $F_s$  :=  $C_s$ 
10:         $L_s$  :=  $S_s$ 
11:      else if  $S_s$ .regions.size == 0 then
12:        create State  $E_s$  in parent region of  $S_s$ 
13:        for all  $S_s$ .outgoingTransitions as  $T_t$  do
14:           $E_s$ .outgoingTransitions.add( $T_t$ )
15:        create Transition from  $S_s$  to  $E_s$  as termination
16:        create Region  $R_r$  in  $S_s$ 
17:        create State  $I_s$  in  $R_r$  and set initial
18:        create State  $T_s$  in  $R_r$  and set final
19:         $F_s$  :=  $I_s$ 
20:         $L_s$  :=  $T_s$ 
21:      else if  $S_s$ .regions.size == 1 then
22:        Region  $R_r$  :=  $S_s$ .regions.first
23:         $L_s$  := initial state of  $R_r$ 
24:         $L_s$ .initial := false
25:        create State  $I_s$  in  $R_r$  and set initial
26:         $F_s$  :=  $I_s$ 
27:      else
28:        create Region  $R_r$  in  $S_s$ 
29:        create State  $I_s$  in  $R_r$  and set initial
30:        create State  $T_s$  in  $R_r$  and set final
31:        create State  $M_s$  in  $R_r$ 
32:        for all  $S_s$ .regions as  $R'_r$  with  $R'_r \neq R_r$  do
33:           $M_s$ .regions.add( $R'_r$ )
34:         $F_s$  :=  $I_s$ 
35:         $L_s$  :=  $M_s$ 
36:        create Transition from  $L_s$  to  $T_s$  and set termination
```

```

37:   for all  $S_s$ .entryActions as  $E_a$  do
38:     State  $C_s := L_s$ 
39:     if  $E_a$  not the last entry action then
40:       create State  $C'_s$  in parent region of  $F_s$ 
41:        $C_s := C'_s$ 
42:       create Transition  $T_t$  from  $F_s$  to  $C_s$  and set immediate
43:        $T_t$ .effects :=  $E_a$ .effects
44:       if  $E_a$  has trigger then
45:          $T_t$ .trigger :=  $E_a$ .trigger
46:         create Transition from  $F_s$  to  $C_s$  and set immediate
47:        $F_s := C_s$ 
48:        $S_s$ .entryActions.remove( $E_a$ )

```

5.3.2 Entry Action

The entry feature transformation is described and illustrated by examples in Section 5.2.5 on page 137. The transformation pseudocode does the following:

First check if there are any entry actions (line 2). If this is the case, the code denotes a particular first state F and a particular last state L based on the following four cases (lines 3 and 4): 1. The state S is a final state (lines 5 to 10): Then a connector is introduced and put “before” S by re-routing all incoming transitions of S . The first state F is the connector and the last state L is S itself. 2. The state S has no regions (lines 11 to 20): Then a new exiting state E is introduced “after” S by re-routing all outgoing transitions of S . The first state F is a new initial state I and the last state L is a new final state T , both created in a new region inside S . 3. The state S has exactly one region (lines 21 to 26): The region is re-used but a new initial state (F) is replacing the original initial state (L). 4. The state S has several regions (lines 27 to 36): A new main state M is used to aggregate all regions of S . A new initial state is becoming F and M is becoming L . Afterwards, the entry actions are hooked in-between states F and L in the same order as they are declared (lines 37 to 47). All entry actions which have a trigger need to

5. Compiling SCCharts

be connected by an additional (unconditional) default transition (lines 44 to 46). Additional connectors are inserted for two or more entry actions. Finally, remove the entry action just handled (line 48).

5.3.3 Exit Action

The exit feature transformation is described and illustrated by examples in Section 5.2.6 on page 140. The transformation pseudocode does the following:

First check if there are any exit actions (line 2). If this is the case the code denotes a particular first state F and a particular last state L based on the following three cases (lines 3 and 4): 1. The state S has no regions (lines 5 to 12): The first state F is a new initial state I and the last state L is a new final state T , both created in a new region inside S . A new exiting conditional state E is introduced “after” S by re-routing all outgoing transitions of S and a new decide conditional state D is introduced inside of S . A valued object V memorizes the exit transition which is taken such that the correct transition from E can be chosen. A new termination transition from S to E is created. 2. The state S has exactly one region (lines 13 to 18): The region is re-used but a final state (L) is replacing one original final state (L). Other final states are forwarded to the first state F (lines 19 to 21). 3. The state S has several regions (lines 22 to 30): A new initial main state M is used to aggregate all regions of S . A new final state is becoming L and M is becoming F . If S had no regions but outgoing transitions, remember which transition internally is taken to know which of the outgoing transitions should trigger (lines 31 to 42). Afterwards, the exit actions are hooked in-between states F and L in the same order as they are declared (lines 43 to 55). All exit actions which have a trigger need to be connected by an additional (unconditional) default transition. Additional connectors are inserted for two or more entry actions. Finally, remove the exit action just handled (line 56).

5.3.4 Initialization

The initialization feature transformation is described and illustrated by examples in Section 5.2.7 on page 144. The pseudocode does the following:

Algorithm 3 Exit Transformation

```

1: function EXIT(STATE  $S_s$ )
2:   if  $S_s$ .exitActions.size > 0 then
3:     State  $F_s$ 
4:     State  $L_s$ 
5:     if  $S_s$ .regions.size == 0 then
6:       if  $S_s$  has no outgoing transitions then
7:         Remove all exit actions and return
8:       create Region  $R_r$  in  $S_s$ 
9:       create State  $I_s$  in  $R_r$  and set initial
10:      create State  $T_s$  in  $R_r$  and set final
11:       $F_s := I_s$ 
12:       $L_s := T_s$ 
13:    else if  $S_s$ .regions.size == 1 then
14:      Region  $R_r := S_s$ .regions.first
15:       $F_s :=$  some final state of  $R_r$ 
16:       $F_s$ .final := FALSE
17:      create State  $T_s$  in  $R_r$  and set final
18:       $L_s := T_s$ 
19:      for all region.finalStates as  $T'_s$  with  $T'_s != L_s$  do
20:        create Transition from  $T'_s$  to  $F_s$  and set immediate
21:         $T'_s$ .final := FALSE
22:    else
23:      create Region  $R_r$  in  $S_s$ 
24:      create State  $T_s$  in  $R_r$  and set final
25:      create State  $M_s$  in  $R_r$  and set initial
26:      for all  $S_s$ .regions as  $R'_r$  with  $R'_r != R_r$  do
27:         $M_s$ .regions.add( $R'_r$ )
28:       $F_s := M_s$ 
29:       $L_s := T_s$ 
30:      create Transition from  $M_s$  to  $T_s$  and set termination
31:    if  $S_s$  had no regions but  $S_s$  has outgoing transitions then
32:      create Connector  $D_s$  in parent region of  $F_s$ 
33:      create Connector  $E_s$  in parent region of  $S_s$ 
34:      create IntValuedObject  $V_v$  in parent state of  $S_s$ 
35:      for all  $S_s$ .outgoingTransitions as  $T_t$  do
36:        create Transition  $T'_t$  from  $E_s$  to  $T_t$ .targetState and set im-
diate

```

5. Compiling SCCharts

```
37:         Set trigger of  $\tau_t$  to  $V_v == \text{ID}(\tau_t)$ 
38:         Set actions of  $\tau_t$  to the ones of  $\tau_t$ 
39:          $F_s.outgoingTransitions.add(\tau_t)$ 
40:          $\tau_t.targetState := D_s$ 
41:         Set action of  $\tau_t$  to  $V_v := \text{ID}(\tau_t)$ 
42:         create Transition from  $S_s$  to  $E_s$  and set termination
43:     for all  $S_s.exitActions$  as  $E_a$  do
44:         State  $C_s := L_s$ 
45:         if  $E_a$  not the last exit action then
46:             create State  $C'_s$  in parent region of  $F_s$ 
47:              $C_s := C'_s$ 
48:             create Transition  $\tau_t$  from  $F_s$  to  $C_s$  and set immediate
49:             if  $F_s.isSuperstate$  then
50:                  $\tau_t.transitionType := \text{TERMINATION}$ 
51:              $\tau_t.effects := E_a.effects$ 
52:             if  $E_a$  has trigger then
53:                  $\tau_t.trigger := E_a.trigger$ 
54:                 create Transition from  $F_s$  to  $C_s$  and set immediate
55:              $F_s := C_s$ 
56:              $S_s.exitActions.remove(E_a)$ 
```

Algorithm 4 Initialization Transformation

```
1: function INITIALIZATION(STATE  $S_s$ )
2:   ValuedObjectList  $L_v := S_s.valuedObjects$  with an initialValue
3:   Int  $i := 0$ 
4:   for all  $L_v$  as  $V_v$  do
5:       create EntryAction  $E_a$  in  $S_s$  at position  $i$ 
6:        $i++$ 
7:        $E_a.effects.add.(V_v.assign(V_v.initialValue))$ 
```

5.3. Pseudocode for High-Level Transformations

Collect all valued objects of S that have an initial value set (line 2). Traverse through this list (line 4) and create entry actions for each initialization (line 5). Each entry action has no trigger but an effect which assigns the initial value to the valued object (line 7). Note that all new entry actions must be placed in the same order as their initializations are declared but before all possibly existing entry actions. This is ensured by index position i (line 3 and line 6).

5.3.5 Abort

Termination

The abort feature transformation is described and illustrated by examples in Section 5.2.8 on page 146. The termination transformation pseudocode is part of the abort transformation and does the following:

First test if there are terminations if there are no terminations then return (line 3). Set up a variable `triggerExpression` which will hold the termination expression for state S (line 4). Then go through all regions of state S as R (line 5) and do the following:

Create a boolean valued object `finishedValuedObject` (line 6) which will indicate the termination of the particular region. Create an entry action `resetFinished` which resets `finishedValuedObject` to false whenever state S is entered (lines 7 and 8). Then go through all final states F of region R and do the following (line 9): Add the assignment `finishedValuedObject := TRUE` to all incoming transitions of F (lines 17 and 18). In case there are more than one incoming transitions then optimize this by creating an auxiliary connector state C and re-route all incoming transitions to this connector node (lines 11 to 13) and add only one transition which sets `finishedValuedObject := TRUE` from C to F (lines 14 and 15). Furthermore, for each region add the `finishedValuedObject` boolean flag to `triggerExpression` which is the conjunction of all termination flags (line 19).

Finally, for all outgoing transitions T of state S , do the following (line 20): If this is a termination transition (line 21) then change its type to be a weak abort (line 22). Furthermore, set T to be immediate (line 23) and add `triggerExpression` in conjunction to its possibly existing trigger (line 24).

5. Compiling SCCharts

Algorithm 5 Termination Transformation

```
1: function TERMINATION(STATE  $S_s$ )
2:   if  $S_s$  has no termination then
3:     return
4:   Expression triggerExpressionv := NULL
5:   for all  $S_s$ .regions as  $R_r$  do
6:     create BoolValuedObject finishedValuedObjectv
7:     create EntryAction resetFinisheda in  $S_s$ 
8:     resetFinisheda.effects.add(finishedValuedObjectv := FALSE)
9:     for all  $R_r$ .finalStates as  $F_s$  do
10:      if  $F_s$  has more than one incoming transition then
11:        create Connector  $C_s$  in  $R_r$ 
12:        for all  $F_t$ .incomingTransitions as  $T_t$  do
13:           $T_t$ .targetState :=  $C_s$ 
14:          create Transition  $T'_t$  from  $C_s$  to  $F_s$ 
15:           $T'_t$ .effects.add(finishedValuedObjectv := TRUE)
16:        else
17:          for all  $F_t$ .incomingTransitions as  $T_t$  do
18:             $T_t$ .effects.add(finishedValuedObjectv := TRUE)
19:          triggerExpressionv := triggerExpressionv AND finishedValuedObjectv
20:   for all  $S_s$ .outgoingTransitions as  $T_t$  do
21:     if  $T_t$ .type == TERMINATION then
22:        $T_t$ .type := WEAKABORT
23:        $T_t$ .immediate :=  $T_t$  not conditionally delayed
24:        $T_t$ .trigger :=  $T_t$ .trigger AND triggerExpressionv
```

5.3. Pseudocode for High-Level Transformations

Abort

The abort feature transformation is described and illustrated by examples in Section 5.2.8 on page 146. The abort transformation pseudocode does the following:

First check if there are any *untransformed transitions* (uT line 2) that need to be handled by the abort transformation if there exist inner behavior (line 3). In case there are not, then the else case in lines 71 to 73 replaces all strong aborts by (default) weak aborts. Note that weak aborts (without the red circle transition marker) are the default for SCCharts. This might only influence cases where there is non-abortable behavior inside the state like for entry or exit actions.

Remember *untransformed aborts* in uA (line 4). In case uT or uA is true then the abort transformation first denotes some boolean flags to simplify further case-handling (lines 7 to 9). dwA denotes whether the state S has any delayed weak aborts in which case the transformed state needs a watcher control region. msA denotes whether the state S has delayed strong aborts and also other transformations of other types in which case the transformed state also needs a watcher control region. nCR denotes whether the transformed state S needs a watcher control region which is additionally the case if S had final states in all of its regions but no outgoing termination transition.

The first part from lines 12 to 15 creates a control region if this is needed. This includes an initial Run and a final Done state. Note that the transitions are not created yet. They will be created later in lines 61 to 69. The second part from lines 16 to 18 creates three expressions which are later used in lines 38 to 49 to explicitly abort the internals of state S using *aborting transitions*. These are a disjunction of triggers of all delayed strong aborts (sATrigger), a disjunction of triggers of all immediate strong aborts (sAITrigger), and a disjunction of triggers of all weak aborts (wAITrigger).

Now, in lines 19 to 25, two specialties are handled: 1. If there are delayed weak aborts then an auxiliary trigger variable is introduced for each delayed weak abort trigger. The mapping is conserved for later usage. 2. If there are mixed immediate and delayed strong aborts then also an auxiliary trigger variable is introduced for each delayed strong abort trigger. The mapping is conserved for later usage.

5. Compiling SCCharts

Algorithm 6 Abort Transformation

```

1: function ABORT(STATE  $S_s$ )
2:   Bool  $uT := ((S_s \text{ has more than one outgoing transitions}) \text{ OR } (S_s \text{ has one outgoing AND NOT a termination transition without any trigger}))$ 
3:   if  $uT$  AND  $S_s$  has inner behavior then
4:     Bool  $uA := S_s$  has outgoing transitions NOT of type termination
5:     TransitionList  $sO_t := S_s.outgoingTransitions$ 
6:     RegionList  $sR_r := S_s.regions$ 
7:     Bool  $dwA := S_s$  has delayed weak aborts
8:     Bool  $msA := S_s$  has mixed delayed strong aborts
9:     Bool  $nCR := (dwA \text{ OR } msA \text{ OR } S_s \text{ has final states but no termination})$ 
10:    if  $uT$  OR  $uA$  then
11:      State  $Run_s, State$   $Done_s$ 
12:      if  $nCR$  then
13:        create Region  $ctrlRegion_r$  in  $S_s$ 
14:        create State  $Run_s$  in  $ctrlRegion_r$  and set initial
15:        create State  $Done_s$  in  $ctrlRegion_r$  and set final
16:        Expr  $sATrigger := \bigvee$  triggers of strong aborts of  $S_s$ 
17:        Expr  $sAITrigger := \bigvee$  triggers of immediate strong aborts of  $S_s$ 
18:        Expr  $wAITrigger := \bigvee$  triggers of weak aborts of  $S_s$ 
19:        if  $dwA$  OR  $msA$  then
20:          create BoolValuedObject  $dATrig_v$  and set Bool
21:          create EntryAction  $dATrig_v := \text{FALSE}$  in  $S_s$ 
22:          if  $dwA$  then
23:            Replace trigger of delayed weak abort  $A_t$  in weakAbort-
                Trigger by  $dATrig_v$  and remember mapping of  $A_t$  to  $dATrig_v$ 
24:          if  $msA$  then
25:            Replace trigger of any delayed abort  $A_t$  in strongAbort-
                Trigger by  $dATrig_v$  and remember mapping of  $A_t$  to  $dATrig_v$ 
26:          Expression  $terminationTrigger := \text{TRUE}$ 
27:          for all  $S_s.regions$  as  $R_r$  do
28:            create State  $RA_s$  in  $R_r$  and set final
29:            for all  $R_r.states$  as  $RS_s$  with  $RS_s$  not a connector do
30:              if  $RS_s \neq RA_s$  AND  $RS_s$  NOT final then
31:                if  $sATrigger$  exists then
32:                  create Transition  $T_t$  from  $RS_s$  to  $RA_s$ 

```

5.3. Pseudocode for High-Level Transformations

```
33:         if  $RS_s$  has inner behavior then
34:              $T_t.type := STRONGABORT$ 
35:          $T_t.priority := HIGHEST$ 
36:          $T_t.trigger := sATrigger$ 
37:          $T_t.immediate := FALSE$ 
38:     if  $sATrigger$  exists then
39:         create Transition  $T_t$  from  $RS_s$  to  $RA_s$ 
40:         if  $RS_s$  has inner behavior then
41:              $T_t.type := STRONGABORT$ 
42:              $T_t.priority := HIGHEST$ 
43:              $T_t.trigger := sATrigger$ 
44:              $T_t.immediate := TRUE$ 
45:         if  $wATrigger$  exists then
46:             create Transition  $T_t$  from  $RS_s$  to  $RA_s$ 
47:              $T_t.priority := LOWEST$ 
48:              $T_t.trigger := wATrigger$ 
49:              $T_t.immediate := TRUE$ 
50:     if  $nCR$  then
51:         for all  $S_s.outgoingTransitions$  as  $O_t$  do
52:             create Transition  $cT_t$  from  $Run_s$  to  $Done_s$ 
53:              $cT_t.priority := LOWEST$ 
54:              $cT_t.immediate := O_t.immediate$ 
55:              $cT_t.trigger := O_t.trigger$ 
56:             Get  $dATrig_v$  from mapping for  $O_t$  if exists
57:             if  $dATrig_v$  exists then
58:                  $cT_t.effects.add(dATrig_v := TRUE)$ 
59:         create Connector  $C_s$  in parent region of  $S_s$ 
60:         create Transition  $CT_s$  from  $S_s$  to  $C_s$  and set termination
61:         for all  $S_s.outgoingTransitions$  as  $O_t$  do
62:              $O_t.sourceState := C_s$ 
63:             if  $O_t$  is the default/lowest prio transition then
64:                  $O_t.trigger = NULL$ 
65:             else
66:                 Get  $dATrig_v$  from mapping for  $O_t$  if exists
67:                 if  $dATrig_v$  exists then
68:                      $O_t.trigger := dATrig_v$ 
```

5. Compiling SCCharts

```
69:         Ot.type := WEAKABORT
70:     else
71:         for all Ss.outgoingTransitions as Ot do
72:             if Ot.type == STRONGABORT then
73:                 Ot.type := WEAKABORT
```

Lines 26 to 49 modify the internals of S to explicitly abort each state in case a strong or weak abort triggers. Note that connector or final states are skipped here. Also note that the delay for delayed weak aborts happens in the control region. Note that the delay for delayed strong aborts happens in S itself but also in the control region in case of mixed immediate and delayed strong aborts. The immediate strong aborts get the highest priority over strong aborts. The weak aborts get the lowest priority and are always set to be immediate.

In case a control region is needed, the transitions for that are created in lines 50 to 58. If auxiliary variables for the trigger exist in the mapping then the control region will set them to true if according transitions are taken. Note that all transitions of S are handled in the original order which preserves their priority by setting each priority of the next transformation to the lowest one (line 53).

Finally, lines 61 to 69 create the forking branch for deciding where to go after terminating the transformed S . Note that all outgoing transitions are immediate. To handle correctly delayed weak aborts or mixed delayed and immediate strong aborts, the triggers are exchanged by auxiliary variables on existence.

5.3.6 Complex Final State

The complex final state feature transformation is described and illustrated by examples in Section 5.2.9 on page 176. The transformation pseudocode does the following:

First check in line 2 whether any complex final states must be eliminated

Algorithm 7 ComplexFinalState Transformation

```

1: function COMPLEXFINALSTATE(STATE  $S_s$ )
2:   if there exist regions of  $S_s$  that cannot terminate then
3:     return
4:   if  $S_s$  is root state then
5:     create Region  $M_r$  in  $S_s$ 
6:     create State  $l_s$  in  $M_r$ 
7:     create State  $T_s$  in  $M_r$ 
8:     create Transition from  $l_s$  to  $T_s$  and set immediate termination
9:     for all  $S_s$ .regions as  $R'_r$  with  $R'_r \neq M_r$  do
10:       $l_s$ .regions.add( $R'_r$ )
11:      $S_s := l_s$ 
12:     ValuedObjectList termVariablesv
13:     for all  $S_s$ .regions as  $R_r$  do
14:       if not all states of  $R_r$  are final then
15:         create BoolValuedObject termv in parent state of  $S_s$ 
16:         if  $R_r$ .initialState.final then
17:           create EntryAction termv := TRUE in  $S_s$ 
18:         else
19:           create EntryAction termv := FALSE in  $S_s$ 
20:         termVariablesv.add(termv)
21:         for all  $R_r$ .finalStates as  $F_s$  do
22:           for all  $F_s$ .incomingTransitions as  $T_t$  do
23:             if  $T_t$  not from a final state then
24:                $T_t$ .effects.add(termv := TRUE)
25:             for all  $F_s$ .outgoingTransitions as  $T_t$  do
26:               if  $T_t$  not to a final state then
27:                  $T_t$ .effects.add(termv := FALSE)
28:         for all  $R_r$ .finalStates as  $F_s$  do
29:           if  $R_r$  is a complex final state then
30:              $R_r$ .final := FALSE
31:     for all  $S_s$ .outgoingTransitions that are terminations as  $T_t$  do
32:        $T_t$ .type := WEAKABORT
33:     for all termVariablesv as termv do
34:        $T_t$ .trigger :=  $T_t$ .trigger AND termv

```

5. Compiling SCCharts

at all. If there are any regions that can never terminate then this is not the case so the transformation can skip (line 3).

If S is the root state then encapsulate every region of S inside a new state I and have an explicit new termination transition from I to a new final state T (lines 4 and 10). If T is reached then the SCChart terminates. Now, consider state I instead of S (line 11).

First declare a list of termination variables `termVariables` in line 12.

Line 13 is the optimization which prevents from creating a `termVariable` for regions where all states are final.

Then go through all regions and for every region do the following (lines 13 to 30): First add a termination variable `term` (line 15) which will indicate the termination of a region. This termination variable is initialized with `false` unless the initial state is also a final state (lines 16 to 19). Add `term` to the list of all termination variables `termVariables` (line 20).

Go through all final states F of the region R . For all incoming transitions of F add `term := true` and for all outgoing transitions of F add `term := false` unless the other state is also a (complex) final state. In this case the `term` is already set correctly and does not need modification (lines 22 to 27).

Now, remove the final flag for all complex final states (lines 28 to 30).

Finally, all outgoing termination transitions are transformed to weak abort transitions where the trigger is a conjunction of all termination variables (lines 31 to 34).

5.3.7 During Action

The during feature transformation is described and illustrated by examples in Section 5.2.10 on page 181. The transformation pseudocode does the following:

First decide whether the during action needs to respect a possible termination of state S (lines 2 and 3). If a termination needs to be respected then set `complexDuring` to `true`.

Go through all during actions of state S (lines 4 to 17). For each during action D do the following:

Create a new region R and a state I in this region (lines 5 and 6). Create a self-loop transition from I to I (line 7) and copy all attributes from the

Algorithm 8 During Transformation

```

1: function DURING(State  $S_s$ )
2:   Bool complexDuring := ( $S_s$  has outgoing terminations OR  $S_s$  root state)
3:   complexDuring := (complexDuring AND all regions of  $S_s$  may terminate)
4:   for all  $S_s$ .duringActions as  $D_a$  do
5:     create Region  $R_r$  in  $S_s$ 
6:     create State  $l_s$  in  $R_r$ 
7:     create Transition  $T_t$  from  $l_s$  to  $l_s$ 
8:     Copy attributes, trigger, effects from  $D_a$  to  $T_t$ 
9:     if  $D_a$ .immediate then
10:      create State  $S'_s$  in  $R_r$ 
11:      create Transition from  $S'_s$  to  $l_s$ 
12:       $T_t$ .targetState :=  $S'_s$ 
13:      if complexDuring then
14:         $S'_s$ .final := TRUE
15:      if complexDuring then
16:         $l_s$ .final := TRUE
17:      $S_s$ .duringActions.remove( $D_a$ )

```

during action to this transition (line 8).

Now, if it is an immediate during action, then we need a second state S' (line 10). The self-loop transition is re-routed to this state S' as a new target (line 12) and a delayed transition is created back to state l (line 11).

If a termination needs to be respected, possibly make states l and S' final states (lines 13 to 14 and lines 15 to 16).

Finally, remove the during action from all during actions (line 17).

5.3.8 Signal

The signal feature transformation is described and illustrated by examples in Section 5.2.11 on page 190. The transformation pseudocode does the following:

First create a commonly used immediate during action A to (later) reset local or output boolean present status variables (lines 2 to 4).

5. Compiling SCCharts

Algorithm 9 Signal Transformation

```
1: function SIGNAL(STATE  $S_s$ )
2:   DuringAction  $A_a$ 
3:   if  $S_s$  has local or output signals then
4:     create DuringAction  $A_a$  in  $S_s$  and set immediate
5:   if  $S_s$  has local actions then
6:     create Region  $R_r$  in  $S_s$ 
7:     create State  $S'_s$  in  $R_r$  and set initial
8:     Move actions and regions of  $S_s$  to  $S'_s$ 
9:   for all  $S_s$ .signals as  $P_v$  do
10:    if  $P_v$  is a valued signal then
11:      create ValuedObject  $V_v$  in  $S_s$ 
12:      create ValuedObject  $C_v$  in  $S_s$ 
13:      create DuringAction  $U_a$  in  $S_s$  and set immediate
14:       $U_a$ .effects.add( $V_v := C_v$ )
15:       $U_a$ .trigger :=  $P_v$ 
16:      create DuringAction  $R_a$  in  $S_s$  and set immediate
17:       $R_a$ .effects.add( $C_v :=$  neutral element of valued signal  $P_v$ )
18:      Copy attributes and type from  $P_v$  to  $V_v$  and to  $C_v$ .
19:       $V_v$ .input :=  $P_v$ .input
20:       $V_v$ .output :=  $P_v$ .output
21:      In all actions inside  $S_s$ , to all emissions of  $P_v$ :
22:      Add an assignment to  $C_v$  (using combine operator of  $P_v$ ).
23:      In all value tests of  $P_v$  in any expressions inside  $S_s$ :
24:      Replace the test of  $P_v$  by a test of  $V_v$ .
25:      Remove initial value and combine operator of  $P_v$ .
26:      Set  $P_v$  to be a boolean variable.
27:      In all actions inside  $S_s$ :
28:      Replace emissions of  $P_v$  by an assignment  $P_v := (P_v \text{ OR } \text{TRUE})$ .
29:      if  $P_v$  is not an input then
30:         $A_a$ .effects.add( $P_v := \text{FALSE}$ );
```

5.3. Pseudocode for High-Level Transformations

Ensure that in case of local actions, the state is encapsulated properly by an additional hierarchy layer such that reset/absent action always occurs even before possibly declared entry actions (lines 5 to 8).

Now, go through all signals P of state S (line 9). For valued signals: Create a variable V which will hold the value of P (line 11). Create a variable C which will collect the value of P in each tick (line 12). Create an update during action which will update V to the collected new value of C in each tick when P is emitted (lines 14 and 15). Create a value reset during action R that resets C to the neutral element in each tick before possibly assigning relative value updates using the combination function (lines 16 and 17). Make sure that V and C have the correct types (line 18). If P is an input or output, also V should be an input or output (lines 19 and 20). Wherever P is emitted (with a value), add a new assignment with a relative write to C using the combine operator of P (lines 21 and 22). Wherever a value test of P was done, instead test for the value of variable V (lines 23 and 24). Remove the initial value and the combine operator from the declaration of P because P will become a boolean variable indicating the present status (line 25).

Then turn signal P into a boolean variable (line 26). Replace emissions of P by relative writes, setting the present status to “present” (TRUE) (lines 27 and 28). If P is a local or output variable then add an effect to the commonly used during action A which resets P to “absent” (FALSE) (lines 29 and 30).

5.3.9 Pre

The pre feature transformation is described and illustrated by examples in Section 5.2.12 on page 193. The transformation pseudocode does the following:

First decide whether the pre transformation needs to respect a possible termination of state S (lines 2 and 3). If a termination needs to be respected then set `complexPre` to true.

First get a list L of all valued objects of state S that are referenced by a pre operator (line 4). Declare a new region `preRegion` and two states `preInit` and `preWait` as well as two transitions `transInitWait` and `transWaitInit` that are used commonly for all valued objects in state S (lines 5 to 9). `preInit` is an initial and final state and `preWait` is a final state.

5. Compiling SCCharts

Algorithm 10 Pre Operator Transformation

```
1: function PRE(STATE  $S_s$ )
2:   Bool complexPre := ( $S_s$  has outgoing terminations OR  $S_s$  root state)
3:   complexPre := (complexPre AND all regions of  $S_s$  may terminate)
4:   ValuedObjectList  $L_v$  :=  $S_s$ .valuedObjects which are referenced by a pre
   operator
5:   Region preRegionr
6:   State preInits and set initial and final
7:   State preWaits and set final
8:   Transition transInitWaitt
9:   Transition transWaitInitt
10:  for all  $L_v$  as  $V_v$  do
11:    if preRegion not yet set then
12:      create Region preRegionr in  $S_s$ 
13:      create State preInits in preRegionr
14:      create State preWaits in preRegionr
15:      create Transition transInitWaits from preInits to preWaits
16:      create Transition transWaitInits from preWaits to preInits
17:      transInitWaits.immediate := TRUE
18:      create ValuedObject Prev in  $S_s$ 
19:      create ValuedObject Regv in  $S_s$ 
20:      Copy attributes and type from  $V_v$  to Prev
21:      Copy attributes and type from  $V_v$  to Regv
22:      transInitWaitt.effects.add(Regv :=  $V_v$ )
23:      transWaitInitt.effects.add(Prev := Regv)
24:      for all  $S_s$ .actions as  $A_a$  do
25:        Replace pre expression pre( $V_v$ ) in  $A_a$  by Prev
26:        Replace pre value expression pre( $?V_v$ ) in  $A_a$  by ?Prev
27:      if complexPre then
28:        preInits.final := TRUE
29:        preWaits.final := TRUE
```

5.3. Pseudocode for High-Level Transformations

Now, walk through all valued objects of L (line 10). For each valued object V do the following: If the common `preRegion` is not yet set then create it as a new region inside state S . Create also two states `preInit` and `preWait` inside `preRegion` (lines 12 to 14). Connect the states by an immediate transition `transInitWait` from `preInit` to `preWait` and a delayed transition `transWaitInit` from `preWait` to `preInit` (lines 15 to 17). In any case, create two variables `Pre` and `Reg` for each V and copy all attributes of V to the new variables (lines 18 to 21). Then add an effect setting `Reg := V` to `transInitWait` and an effect setting `Pre := Reg` to `transWaitInit` (lines 22 and 23). Finally, replace all `pre(V)` expressions in all actions inside S by `Pre`. Additionally, replace all `pre(?V)` value expression in all actions inside S by `?Pre` (lines 24 to 26).

In lines 27 to 29, the auxiliary states `preInit` and `preWait` are set to be final states in case the state S can be left by a termination transition.

5.3.10 Suspend

Algorithm 11 Suspend Transformation

```
1: function SUSPEND(STATE  $S_s$ )
2:   if  $S_s$  has no suspends then
3:     return
4:   create BoolValuedObject  $enabled_v$  in  $S_s$ 
5:   for all actions inside  $S_s$  as  $A_a$  with  $A_a$  not a suspend action do
6:      $A_a.trigger := (A_a.trigger \text{ AND } enabled_v)$ 
7:   create DuringAction  $R_a$  in  $S_s$  and set immediate
8:    $R_a.effects.add(enabled_v := TRUE)$ 
9:   for all  $S_s.suspends$  as  $B_a$  do
10:    create DuringAction  $D_a$  in  $S_s$ 
11:     $D_a.immediate := B_a.immediate$ 
12:     $D_a.effects.add(enabled_v := enabled_v \text{ AND } (NOT (B_a.trigger)))$ 
13:     $S_s.suspends.remove(B_a)$ 
```

The suspend feature transformation is described and illustrated by examples in Section 5.2.13 on page 199. The transformation pseudocode does the following:

5. Compiling SCCharts

If there are no suspensions then do nothing in this transformation (lines 2 and 3).

Otherwise, create a common valued object enabled in state S (line 4).

For all actions of and inside S including any transitions, add the enabled flag in conjunction to possible triggers (lines 5 and 6).

Create an immediate during action R which resets the common enabled flag to true in each tick (lines 7 and 8).

Now, go through all suspends B of state S and do the following (lines 9 to 13): Create a during action D (line 10) and set it to be immediate iff the suspend was immediate (line 11). The during action has no trigger but the effect is a relative write update of the enabled flag setting it to the conjunction of enabled and the negated trigger of the suspend (line 12).

Finally, remove the suspend B from the list of suspends in S (line 13).

5.3.11 Count Delay

Algorithm 12 CountDelay Transformation

```
1: function COUNTDELAY(STATE  $S_s$ )
2:   for all  $S_s.outgoingTransitions$  as  $O_t$  do
3:     if  $O_t$  has a delay > 1 then
4:        $P_s := S_s.parentRegion.parentState$ 
5:       create IntValuedObject  $C_v$  in  $P_s$ 
6:       create EntryAction  $E_a$  in  $S_s$ 
7:        $E_a.effects.add(C_v := 0)$ 
8:       if NOT  $O_t$  is immediate then
9:         create EntryAction  $E'_a$  in  $S_s$ 
10:         $E'_a.trigger := O_t.trigger$ 
11:         $E'_a.effects.add(C_v := -1)$ 
12:       create DuringAction  $D_a$  in  $P_s$  and set immediate
13:        $D_a.effects.add(C_v := C_v + 1)$ 
14:        $D_a.trigger := O_t.trigger$ 
15:        $O_t.trigger := (C_v == O_t.delay)$ 
16:        $O_t.delay := 1$ 
```

5.3. Pseudocode for High-Level Transformations

The count delay feature transformation is described and illustrated by examples in Section 5.2.14 on page 201. The transformation pseudocode does the following:

For every count delayed transition *O* of state *S* (lines 2 and 3) do the following (lines 4 to 16): Remember the parent state of *S* in *P* (line 4). Create a counting integer valued object *C* in *P*. Create an entry action in *S* which resets the counter (lines 6 and 7). Prevent immediate counting in the non-immediate case (lines 8 and 11). Create an incrementing during action in *P* (lines 12 to 14). Modify the trigger of *O* to only depend on whether the *C* has reached *O.delay* (line 15). Set the delay of *O* to 1 (line 16).

5.3.12 History

The history feature transformation is described and illustrated by examples in Section 5.2.15 on page 206. The transformation pseudocode does the following:

First create lists to remember history transitions, outgoing from state *S*, in *H* (line 2) and non-history transitions in *N* (line 5).

If there are no history transitions then return (lines 3 and 4).

The `initialValue` is the integer which encodes the modeled initial state in case of entering state *S* by a non-history transition (line 6).

`stateEnumsAll` is an enumeration of all integer valued objects, one for each region, to encode the history state (line 7 and lines 15 and 16). `stateEnumsDeep` is a similar enumeration for the states in regions which are hierarchically deeper than the shallow, direct regions of *S* (lines 8 and 18). This is important to remember to later reset the valued objects for deeper regions in the shallow history case.

If there are no deep history transitions then an optimization is to not remember these deeper states (lines 11 and 12).

Now, go through all regions as *R* (line 13) and do the following (lines 14 to 31): For every region, create a counter to encode the possible state to jump to when entering *S* (line 14) via a history transition. Create a valued object `stateEnum` to encode and remember the current state (line 15) and add it to the list (line 16). If this is a deeper state then also add it to the deep list (lines 17 and 18). Remember the modeled original initial state

5. Compiling SCCharts

Algorithm 13 History Transformation

```
1: function HISTORY(STATE  $S_s$ )
2:   TransitionList  $H_t :=$  incoming history transitions of  $S_s$ 
3:   if  $H_t$  empty then
4:     return
5:   TransitionList  $N_t :=$  incoming non-history transitions of  $S_s$ 
6:   Int initialValue
7:   ValuedObjectList stateEnumsAllv
8:   ValuedObjectList stateEnumsDeepv
9:   RegionList regionsr :=  $S_s$ .regions
10:  RegionList regionsDeepr := all contained regions of  $S_s$  (hierarchically)
11:  if  $S_s$  has no deep history transitions then
12:    regionsDeepr := regionsr
13:  for all regionsDeepr as  $R_r$  do
14:    Int counter := 0
15:    create IntValuedObject stateEnumv in  $S_s$ 
16:    stateEnumsAllv.add(stateEnumv)
17:    if NOT regionsr.contains( $R_r$ ) then
18:      stateEnumsDeepv.add(stateEnumv)
19:     $O|_s := R_r$ .initialState
20:     $O|_s$ .initial := false
21:    StateList subStatess :=  $R_r$ .states
22:    create State  $N|_s$  in  $R_r$  and set initial
23:    for all subStatess as subStates do
24:      create Transition  $T_t$  from  $N|_s$  to subStates and set immediate
25:       $T_t$ .trigger := (stateEnumv == counter)
26:      create EntryAction  $E_a$  in subStates
27:       $E_a$ .effects.add(stateEnumv := counter)
28:      if subStates ==  $O|_s$  then
29:        initialValue := counter
30:        stateEnumv.initialValue := counter
31:        counter := counter + 1
32:    for all  $H_t$  as  $T_t$  do
33:      if  $T_t$  is NOT a deep history then
34:        for all stateEnumsDeepv as stateEnumv do
35:           $T_t$ .effects.add(stateEnumv := initialValue)
```

5.3. Pseudocode for High-Level Transformations

```
36:     Tt.history := not a history transition
37:   for all Nf as Tf do
38:     for all stateEnumsAllv as stateEnumv do
39:       Tt.effects.add(stateEnumv := initialValue)
```

in OI (line 19) and set to be non-initial (line 20). Remember all substates in a separate list subStates (line 21) and add a new initial state (line 22). For every substate, create an immediate transition which connects the new initial state and the substate (line 24). It is triggered in case stateEnum has the counter value (line 25). Note that the counter is incremented (line 31) for every substate. Additionally, for every state, create an entry action which sets stateEnum to the counter value (lines 26 and 27). If the substate is the modeled original initial state then set the initial value of stateEnum and initialValue to its counter value (lines 28 to 30). This value is needed to enter S by a non-history transition.

Finally, go through all history transitions of H and if it is a shallow history transition, add an effect which resets all deeper stateEnum values (lines 33 to 35). Set it to be a typical, non-history transition (line 36). Also, go through all typical, non-history transitions of N and add an effect which resets all stateEnum values (lines 37 to 39).

5.3.13 Static Variables

Algorithm 14 Static Transformation

```
1: function STATIC(STATE Ss)
2:   ValuedObjectList Lv := Ss.valuedObjects with isStatic flag set to TRUE
3:   for all Lv as Vv do
4:     Vv.name := Vv.getUniqueHierarchicalName
5:     Vv.isStatic := FALSE
6:     Ss.rootState.valuedObjects.add(Vv)
```

The static variables feature transformation is described and illustrated

5. Compiling SCCharts

by examples in Section 5.2.16 on page 210. The transformation pseudocode does the following:

Collect all valued objects of S that are declared as static (line 2). Traverse through this list and do the following for each valued object (line 3): First, rename the valued object according to the hierarchy of its original declaration (line 4). Then, set the valued object declaration to be not static any more (line 5). Finally, move its declaration to the root state (line 6).

5.3.14 Deferred

Algorithm 15 Deferred Transformation

```
1: function DEFERRED(STATE  $S_s$ )
2:   TransitionList incomingDeferredTransitions $t$  := incoming deferred transitions of  $S_s$ 
3:   TransitionList incomingNonDeferredTransitions $t$  := incoming non-deferred transitions of  $S_s$ 
4:   if incomingDeferredTransitions $t$  is empty then
5:     return
6:   create BoolValuedObject deferVariable $v$  in  $S_s$ .parentRegion.parentState
7:   deferVariable $v$ .initialValue := FALSE
8:   create DuringAction  $R_a$  in  $S_s$ 
9:    $R_a$ .effects.add(deferVariable $v$  := FALSE)
10:  for all incomingDeferredTransitions $t$  as  $T_t$  do
11:     $T_t$ .deferred := FALSE
12:     $T_t$ .effects.add(deferVariable $v$  := TRUE)
13:  for all incomingNonDeferredTransitions $t$  as  $T_t$  do
14:     $T_t$ .effects.add(deferVariable $v$  := FALSE)
15:  for all  $S_s$ .outgoingTransitions as  $T_t$  do
16:    if  $T_t$ .immediate then
17:       $T_t$ .trigger := ( $T_t$ .trigger AND (NOT deferVariable $v$ ))
```

The deferred feature transformation is described and illustrated by examples in Section 5.2.17 on page 211. The transformation pseudocode does the following:

5.3. Pseudocode for High-Level Transformations

First remember all incoming deferred transitions to *S* in a list `incomingDeferredTransitions` (line 2) and all incoming non-deferred transitions to *S* in `incomingNonDeferredTransitions` (line 3) If there are no deferred transitions then return (lines 4 and 5).

Otherwise, create a `deferVariable` valued object in the parent state of *S* (line 6).

Initialize this variable to `FALSE` (line 7) and ensure that it is reset in each tick by a during action (lines 8 and 9).

For all incoming deferred transitions *T* (line 10) do the following: Reset the deferred flag making it a typical transition (line 11). Add an effect which sets the `deferVariable` to `TRUE` when entering state *S* by this transition. For all incoming non-deferred transitions *T* (line 13) do the following: Add an effect which sets the `deferVariable` to `FALSE` when entering state *S* by this transition. Finally, for all outgoing immediate transitions of state *S*, prevent that these transitions can be taken in case the `deferVariable` is true by modifying their triggers (lines 15 to 17). Note that for the deep deferred feature expansion additionally, all internal immediate actions of *S* would be deactivated if the `deferVariable` is true, similarly to lines 15 to 17.

5.3.15 Weak Suspend

The weak suspend feature transformation is described and illustrated by examples in Section 5.2.18 on page 215. The transformation pseudocode does the following:

First put all weak suspends of state *S* in a dedicated list `weakSuspends` (line 6). If there are no weak suspends then return (lines 3 and 4).

Create a boolean flag `wsFlag` in state *S* (lines 5 and 6) and initialize it to false. For all weak suspends create a during action in *S* which as an effect evaluates the trigger of the weak suspend and saves the result in the `wsFlag` for the purpose of WTO (lines 7 and 10).

Now, go through all regions *R* of state *S* (line 11) and do the following: First save the list of existing substates of the region *R* in `subStates` for later traversal (line 12).

Create an auxiliary weak suspend state *WS* in *R* (line 13). This state will perform the re-routing to ensure to get back to the state where the “last wish”

5. Compiling SCCharts

Algorithm 16 Weak Suspend Transformation

```
1: function WEAKSUSPEND(STATE  $S_s$ )
2:   ActionList weakSuspend $_a$  := weak suspend actions of  $S_s$ 
3:   if weakSuspend $_a$  is empty then
4:     return
5:   create BoolValuedObject wsFlag $_v$  in  $S_s$ 
6:   wsFlag $_v$ .initialValue := FALSE
7:   for all weakSuspend $_a$  as  $W_a$  do
8:     create DuringAction  $D_a$  in  $S_s$ 
9:      $D_a$ .immediate :=  $W_a$ .immediate
10:     $U_a$ .effects.add(wsFlag $_v$  :=  $W_a$ .trigger)
11:   for all  $S_s$ .regions as  $R_r$  do
12:     StateList subStates $_s$  :=  $R_r$ .states
13:     create State  $WS_s$  in  $R_r$ 
14:     create IntValuedObject stateBookmark $_v$  in  $S_s$ 
15:     stateBookmark $_v$ .initialValue := 0
16:     Int counter := 0
17:     create BoolValuedObject lastWishDone $_v$  in  $S_s$ 
18:     create DuringAction  $D_a$  in  $S_s$  and set immediate
19:      $D_a$ .effects.add(lastWishDone $_v$  := FALSE)
20:     create EntryAction  $E_a$  in  $WS_s$ 
21:      $E_a$ .effects.add(lastWishDone $_v$  := lastWishDone $_v$  || TRUE)
22:     create State  $I_s$  in  $R_r$  and set initial
23:     create Transition  $IWS_t$  from  $I_s$  to  $WS_s$  and set immediate
24:      $IWS_t$ .trigger := (wsFlag $_v$  AND lastWishDone $_v$ )
25:     for all subStates $_s$  as  $suS_s$  do
26:       create Transition reEnter $_t$  from  $WS_s$  to  $suS_s$  and set immediate
27:       reEnter $_t$ .deferred := TRUE
28:       reEnter $_t$ .trigger := (stateBookmark $_v$  == counter)
29:       create EntryAction  $ES_a$  in  $suS_s$ 
30:        $ES_a$ .effects.add(stateBookmark $_v$  := counter)
31:        $ES_a$ .trigger := (NOT wsFlag $_v$ )
32:       counter := counter + 1
33:     if (NOT  $suS_s$ .final) OR ( $suS_s$ .parentState ==  $S_s$ ) then
34:       create Transition  $WST_t$  from  $suS_s$  to  $WS_s$  and set immediate
35:        $WST_t$ .trigger := wsFlag $_v$ 
36:       set  $WST_t$  to lowest priority
```

5.3. Pseudocode for High-Level Transformations

```
37:         if suSs.initial then
38:             suSs.initial := FALSE
39:             create Transition from ls to suSs and set immediate
40:     if Ss may terminate then
41:         create Region wsTr in Ss
42:         create State ls in wsTr and set initial
43:         create State Fs in wsTr and set final
44:         create Transition T1t from ls to Fs and set immediate
45:         create Transition T2t from Fs to ls
46:         T1t.trigger = (NOT wsFlagv)
47:         T2t.trigger = wsFlagv
```

started for the next tick. Create an integer stateBookmark (line 14) which will help to remember which state to re-route to. The original initial state will be represented by number 0. Hence, stateBookmark is initialized with 0 (line 15). A counter will help to give each substate of R a dedicated number as an ID (line 16). Create a boolean lastWishDone variable which denotes if the last wish has already been processed or if it is still processing (line 17). A during action will reset this variable to false in each tick (lines 18 and 19). An entry actions in state WS will set it to true (lines 20 and 21). Now, create an auxiliary initial state l (line 22). An immediate transition from state l to state WS (line 23) is triggered if the wsFlag is true and the lastWishDone is true (line 24). This is the case when returning silently to the starting state for the next tick. Now, create deferred transitions for every (original) substates of R as targets from the WS state triggered if the stateBookmark equals the counter ID (lines 26 to 28). Also create an entry action for every state. This entry action will set the stateBookmark to the counter ID in case the wsFlag is false (lines 29 to 31). Increment the counter for the next substate (line 32). Add a transition with lowest priority triggered by wsFlag to state WS (lines 33 to 36) which is taken when the “last wish” is complete. In case of a final state, a termination of the parent superstate should take care of completing the “last wish” and returning to the correct start state for the next tick. In case of the initial state, reset the initial flag and create a default transition

5. Compiling SCCharts

from the new auxiliary initial state l to this substate (lines 37 to 39). The lines after line 40 ensure that the scope of the weak suspend is not left in case the weak suspend trigger holds.

5.3.16 Normalization

Trigger/Effect

Algorithm 17 TriggerEffect Transformation

```
1: function TRIGGEREFFECT(TRANSITION  $\tau_t$ )
2:   Bool  $c := \tau_t.trigger$  exists
3:    $c := ((c \text{ OR } \tau_t \text{ is delayed OR } \tau_t \text{ is termination}) \text{ AND } \tau_t \text{ has effects})$ 
4:    $c := (c \text{ OR } \tau_t \text{ has more than one effects})$ 
5:   if  $c$  then
6:     State  $originalTarget_s := \tau_t.targetState$ 
7:     Region  $parentRegion_r := originalTarget_s.parentRegion$ 
8:     Transition  $lastTransition_t := \tau_t$ 
9:     for all  $\tau_t.effects$  as  $effect$  do
10:      create State  $effectState_s$  in  $parentRegion_r$ 
11:      create Transition  $effectTransition_t$  and set immediate
12:       $effectTransition_t.effects.add(effect)$ 
13:       $effectTransition_t.sourceState := effectState_s$ 
14:       $lastTransition_t.targetState := effectState_s$ 
15:       $lastTransition_t := effectTransition_t$ 
16:       $lastTransition_t.targetState := originalTarget_s$ 
```

The normalization feature transformations are described and illustrated by examples in Section 5.2.19 on page 221. The transformation pseudocode for trigger effect does the following:

First test if the trigger and effect transformation is necessary to run (lines 2 to 4) for a given transition τ . This is the case if a transition trigger exists or the transition is non-immediate (delayed) or the transition is a termination and additionally the transition has any effects. It is also the case if the transition has more than one effects (line 4).

5.3. Pseudocode for High-Level Transformations

Remember the original target state in `originalTarget` (line 6) and the parent region of the transition in `parentRegion` (line 7).

The `lastTransition` variable is first set to the currently transformed transition `T` (line 8). For all effects `effect` of the transition `T` (line 9) do the following: Create a new state `effectState` in the same parent region of the transition (line 10). Create a new effect transition `effectTransition` (line 11). Add the single effect to the new transition (line 12). Connect the new effect transition `effectTransition` as a source from the new state (line 13). Connect the last effect (or original) transition `lastTransition` as a target to the new state (line 14). Now, let the current `effectTransition` be the `lastTransition` for the next effect if any (line 15). Finally, for the last target, connect the `lastTransition` to the original target of `T` (line 16).

Surface/Depth

The normalization feature transformations are described and illustrated by examples in Section 5.2.19 on page 221. The transformation pseudocode for surface depth does the following:

First test if the surface and depth transformation is necessary to run (lines 2 to 6) for a given state `S`. This is the case if `S` has outgoing transitions and the first/only transition is not a termination. Additionally, the first/only transition must either have a trigger or it must be delayed. If `S` is a superstate without any possibly taken termination then the surface depth transformation should not run in order to not produce dead code (lines 4 to 6). If `S` is a superstate with a possibly taken termination then the surface depth transformation adds an auxiliary simple halt state with a termination (lines 7 to 10). If `S` is a simple state without any outgoing transition then add an explicit delay transition as an auxiliary pause construct (lines 11 and 12).

Get a list of immediate transitions and sort these according to their inverted priorities (lines 13 and 14). Create copies of all these immediate transitions and set the original transitions to be non-immediate (lines 15 and 18). Denote the surface and the depth. Initially, set both to `S` (lines 19 and 20). Denote the currently (current) and the previously processed (previous) states (lines 21 and 22). Initially set both to the surface state `S`.

5. Compiling SCCharts

Algorithm 18 SurfaceDepth Transformation

```
1: function SURFACEDEPTH(STATE  $S_s$ )
2:   if  $S_s$  is pause construct OR conditional construct OR action construct
   OR superstate construct then
3:     return
4:   if  $S_s$  is a halt-superstate then
5:     if  $S_s$  cannot terminate then
6:       return
7:     else
8:       create State  $halt_s$  in parent region of  $s_s$ 
9:       create Transition from  $s_s$  to  $halt_s$  and set termination
10:      create Transition from  $halt_s$  to  $halt_s$ 
11:     if  $S_s$  has no outgoing transition and is not final then
12:       create Transition from  $S_s$  to  $S_s$ 
13:     TransitionList immediateTransitionst := immediate outgoing transitions
of  $S_s$ 
14:     sort immediateTransitionst by priorities (lowest to highest)
15:     for all immediateTransitionst as  $T_t$  do
16:       create Transition  $TC_t$  as a copy of  $T_t$ 
17:       set  $TC_t$  to highest priority
18:        $T_t.immediate := FALSE$ 
19:     surfaces :=  $S_s$ 
20:     depths :=  $S_s$ 
21:     State currents := surfaces
22:     State previouss := surfaces
23:     Bool pauseInserted := FALSE
24:     TransitionList orderedTransitionst :=  $S_s.outgoingTransitions$ 
25:     sort orderedTransitionst by priority (highest to lowest)
26:     for all orderedTransitionst as  $T_t$  do
27:       if (NOT  $T_t.immediate$ ) AND (NOT pauseInserted) then
28:         pauseInserted := TRUE
29:         create State depths in  $S_s.parentRegion$ 
30:         create Transition from previouss to depths and set immediate
31:         create State pauses in  $S_s.parentRegion$ 
32:         create Transition from depths to pauses
33:         previouss := pauses
34:         currents := NULL
```

5.4. Priority-Based Low-Level Compilation

```
35:     if currents == NULL then
36:         create State currents in Ss.parentRegion
37:         currents.outgoingTransitions.add(Tt)
38:         create Transition connectt from previouss to currents and set
immediate
39:         Tt.immediate := TRUE
40:         Tt.priority := 1
41:         previouss := currents
42:         currents := NULL
43:     create Transition from previouss to depths and set immediate
```

The flag `pauseInserted` is initialized to false and denotes whether a pause which separates the surface and the depth part has already been inserted (line 23). Then go through all outgoing transitions T of S which are sorted by priority, higher priorities first (lines 24 and 25), and do the following (lines 27 and 42):

If T is non-immediate and a pause was not inserted (line 27) then do the following: Set the `pauseInserted` flag to true and insert a pause construct by adding an auxiliary state and a delayed transition (lines 28 to 32). Set the previous state to the added pause-state and reset the current state to null (lines 33 and 34). If the current state is null (line 35) then do the following: Create a new current state (line 36) and add the existing transition (as if branch) (line 37). Additionally, connect the previous state with a new immediate transition (else branch) (line 38). Make sure that T is an immediate transition with priority 1 (lines 39 and 40). For the next cycle, set previous to the current state and set current to null again (lines 41 and 42). Add an immediate transition back from previous to depth (line 43).

5.4 Priority-Based Low-Level Compilation

After giving details of the high-level transformations in the previous sections, the focus of the following sections is on the low-level compilation paths for SCCharts. As introduced in Section 5.0.2 on page 115, there are two

5. Compiling SCCharts

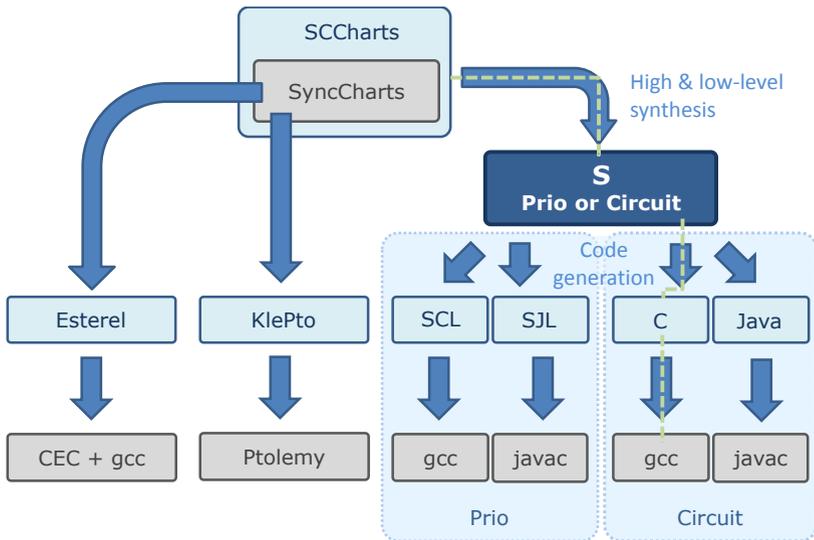


Figure 5.4.1. S intermediate language in the context of SCCharts and SyncCharts SW compilation alternatives: The green dashed line highlights the default circuit-based compilation path to C.

principle compilation paths to follow for low-level synthesis of SCCharts. Figure 5.4.1 gives an overview of SCCharts and SyncCharts compilation paths for SW simulation as implemented in KIELER². It also highlights the Prio-based path (cf. Figure 5.4.2) where SCL and SJL are used as lightweight runtime extensions for C and Java, respectively, to execute code that is based on priorities for a dynamic dispatching of concurrent threads. It also highlights the Circuit-based path where C and Java code is derived directly. Both paths utilize S as an intermediate low-level representation. This S code can contain statically computed priorities, if targeting SCL or SJL, or it can be already purely sequentialized code, if targeting circuit-based C or Java code.

This section gives some details about the S intermediate format and

²<http://rtsys.informatik.uni-kiel.de/kieler>

5.4. Priority-Based Low-Level Compilation

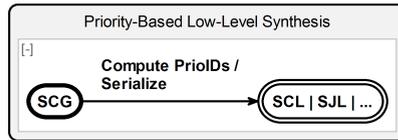


Figure 5.4.2. SCCharts compilation tree (cf. Figure 5.0.1 on page 114): Priority low-level synthesis part

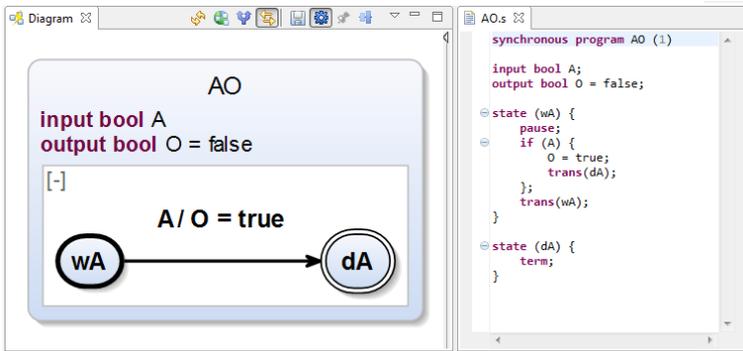


Figure 5.4.3. S intermediate language example

about the priority-based low-level compilation approach to S code with priorities. SJL, as a possible target for S programs with priorities, is discussed later in Section 8.4 on page 407.

5.4.1 S

S was introduced as an intermediate language for targeting both, the priority-based and the circuit-based low-level compilation. Having a common language such as S comes with the following benefits:

Comparison: To compare different low-level compilation paths, it is essential to minimize a possible “technical bias” resulting only from possibly different infrastructure (parts). S serves as a common language which unifies the infrastructure at some point for minimizing such a bias. This

5. Compiling SCCharts

emphasizes conceptual differences of compilation paths.

Model Transformations: S is a minimal imperative language tailored to express computations for the priority-based *and* the circuit-based approach. However, S is still based on a modeling framework such that no textual code generation to S is needed. Hence, model transformations to S can be used. This enables to use the interactive incremental compilation approach down to S code, e. g., tracing of model elements is possible down to S instructions.

Fully-featured Editor: S is based on the Xtext framework. Hence, a fully-featured modeling Editor for S programs can be generated without much effort. Generated S programs can be inspected or modified using this editor.

Simulation and Visualization: The model-based approach and the textual S editor allow to integrate a simulation for S with visualization of active instructions in the editor. This helps to inspect and validate the behavior of 1. the S programs that are generated from a higher abstraction level and 2. the executed programs that are generated from S code on the levels below S.

An example for a simple S program is shown in Figure 5.4.3. The SCChart on the left side is the already discussed AO example (cf. Section 5.2.20 on page 231). The right side shows the equivalent S program of the priority-based low-level compilation path in the textual S editor. The interface of S can declare variables or signals as the interface for SCCharts. In this example, a boolean input variable I and a boolean output variable O is declared. An S program is organized in a sequence of named *states*. Each state contains a sequence of S instructions. In this example, there are two states, state wA and state dA. Control for an S program starts with the

5.4. Priority-Based Low-Level Compilation

```

1 generate s "http://www.cau.de/cs/kieler/s"
2
3 Program:
4   (annotations += Annotation)*
5   'synchronous_program' name = ID '('
6     priority=INT ')'
7   (declarations += Declaration)*
8   (globalHostCodeInstruction = HOSTCODE)?
9   (states += State)+
10 ;
11 State:
12   (annotations += Annotation)*
13   'state' '(' name = ID ')' '{
14     (declarations += Declaration)*
15     (instructions += Instruction;)*
16   }';
17
18 HostCodeInstruction:
19   hostCode = HOSTCODE
20 ;
21 Instruction:
22   Halt | Abort | Join | Pause | Term | If | Trans | Fork
23   | LocalSignal | Emit | Await | Prio |
24   HostCodeInstruction | Assignment
25 ;
26 Assignment:
27   valuedObject=[kexpressions::ValuedObject]
28   '[' indices+=Expression ']*
29   "=" expression = Expression ;
30
31 Prio:
32   'prio' '(' priority=INT (',' continuation=[
33     State])? ')'
34 ;
35 Trans :
36   'trans' '(' continuation=[State] ')'
37 ;
38 Fork :
39   'fork' '(' continuation=[State] ',' priority=INT
40     ')'
41 ;
42 Join :
43   {Join}
44
45   'join' '(' continuation=[State] ')'
46 ;
47 Pause :
48   {Pause}
49   'pause' '(' continuation=[State] ')'
50 ;
51 Term :
52   {Term}
53   'term' '(' continuation=[State] ')'
54 ;
55 Halt :
56   {Halt}
57   'halt' '(' continuation=[State] ')'
58 ;
59 LocalSignal:
60   'signal' '(' signal = [kexpressions::ValuedObject]
61     ')'
62 ;
63 Emit:
64   'emit' '(' signal = [kexpressions::ValuedObject
65     ] '(' value=SEExpression ')' '?' (','
66     continuation=[State])? ')'
67 ;
68 Abort :
69   {Abort}
70   'abort' '(' continuation=[State] ')'
71 ;
72 If :
73   'if' '(' expression = SEExpression (','
74     continuation=[State])? ')' '{
75     (instructions += Instruction;)*
76   }';
77 ;
78 Await:
79   'await' '(' signal= [kexpressions::
80     ValuedObject] (',' continuation=[State
81     ])?)'
82 ;
83 SExpression returns kexpressions::Expression:
84   Expression
85 ;

```

Listing 5.4.1. S intermediate language Xtext grammar

5.4. Priority-Based Low-Level Compilation

first state, which is state `wA` in this example. The first instruction of `wA` is a pause which means that control rests here until the next synchronous tick. After that, there is a conditional `if` instruction which contains another instruction list for its `if` branch. In this example, this means that if the condition, i. e., variable `A` evaluates to true then the instruction list of the `if` branch is executed. This sets the output variable `O` to true and transitions to the state `dA` using the `trans` instruction. In state `dA` there is only a term instruction because the `SCChart` terminates here with a final state.

Grammar and Meta Model

Listing 5.4.1 shows the grammar of `S` which shares the declaration part with the `SCCharts` `SCT` grammar, see Section 3.3 on page 75.

The resulting meta model for `S` is visualized in Figure 5.4.4. As already discussed, an `S` Program consists of a declaration part and a list of State items. Each State has a list of Instruction items. An Instruction is a Pause, an `if`, an Assignment, and so on. Most instructions as the Pause or `Trans` have a continuation which is a State where to continue control. The `if` instruction contains another list of Instruction items for its `if` branch. `S` also supports expressing concurrency using `Fork` and `Join` instructions. Additionally, it supports setting a thread's priority using the `Prio` instruction.

5.4.2 S Code Generation

As Figure 5.4.1 suggests, there are two possible paths to go for low-level synthesis, the priority-based and the circuit-based code generation. Both approaches share `S` as an intermediate representation. The priority-based approach additionally makes use of states, `fork`, and `prio` instructions of `S`. `S` code with priorities, which is produced by transforming from Extended or Core `SCCharts`, directly reflects the structure of an `SCChart`, which eases understanding this compilation step.

Alternatively, as suggested by Figure 5.0.1 on page 114, the (non-sequentialized) `SCG` can be transformed into `S` code with priorities quite easily too by keeping the common synthesis path with the circuit-based approach from Extended `SCCharts` up to the `SCG`. A crucial step is how

5. Compiling SCCharts

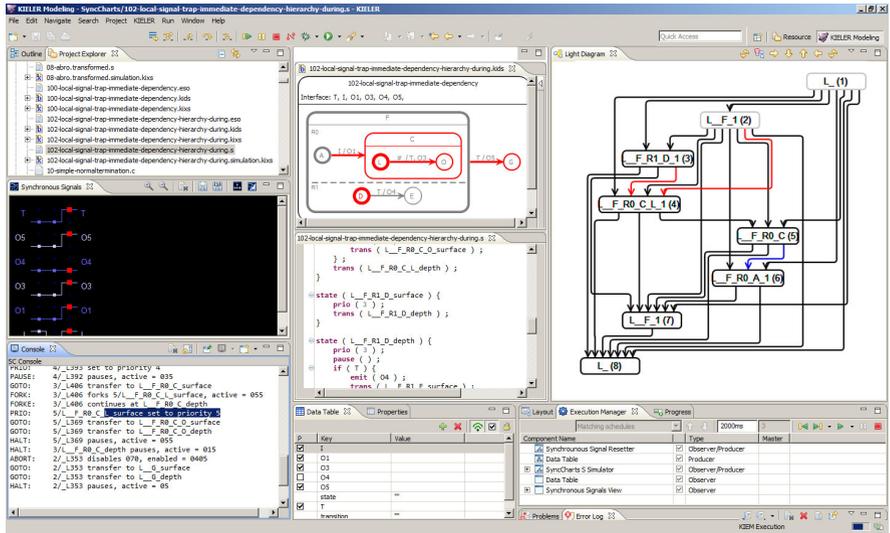


Figure 5.4.5. Simulation with the SCCharts compiler predecessor. The SyncCharts2S compiler uses a priority-based approach. Priority value computation is visualized (right) and dynamic runtime debug information is printed to command line (lower left).

to derive priority values. Therefore, dependency analysis is needed. This information can be retrieved from the SCG in the low-level synthesis.

Details on this part of the low-level synthesis and how to derive priorities were published elsewhere [vHDM⁺14].

Evolution of Priority-Based Compilation

The priority-based approach was initially inspired by *SyncCharts in C/ Synchronous C (SC)* [vH09] and by the goal to produce code from SyncCharts to SC automatically. Amende [Ame10] and Traulsen et al. [TAvH10] developed a first priority-based SyncCharts compiler (SyncCharts2C) as briefly described in Section 5.1 on page 116. This natively handled strong aborts, weak aborts, and normal terminations of SyncCharts as well as pure signals.

5.4. Priority-Based Low-Level Compilation

The successor, the SyncCharts/SCCharts to S compiler (SyncCharts2S), also described in Section 5.1 on page 116, was able to handle strong and weak aborts as well by its core compiler part, but no terminations or signals. It already relied on SLIC-like transformations to pre-process the SyncChart to eliminate, e. g., termination or signal features before. Still, both compilers were able to calculate priorities. The SyncCharts2S compiler was additionally able to produce S code from which SJL or SCL code could be generated. Figure 5.4.5 shows this compiler during simulation. It also shows a view at the right side that visualizes the dependencies that are used to calculate specific priorities. The SyncChart and the resulting S code are shown in the center of the figure. At the lower left side one can see the debug output of the SCL code that was generated and compiled. This drives the simulation in the background and is used to visualize active states (red-colored).

5.4.3 SJL/SCL Code Generation

As mentioned earlier and shown in Figure 5.4.1 on page 264, the SyncCharts/SCCharts to S compiler (SyncCharts2S) is able to generate S code with priorities. From there it is easily possible to generate both, Java-based SJL and C-based SCL code. Figure 5.4.6 shows this for the AO SCCharts example. Note that the SyncCharts2S compiler implementation originally only accepts SyncCharts but conceptually is ready to process all *Berry-constructive* SCCharts (cf. Section 2.6 on page 36) as well.

As shown in Figure 5.4.6, the code generation for SJL and SCL is quite straight forward. Basically, a `tick()` and a `reset()` method is generated. `reset()` resets the SCChart before execution begins and `tick()` is called to compute a reaction for each synchronous tick. For SJL, a switch-case structure is created inside the tick method with S states becoming case labels. For SCL, the tick function contains C labels, one for each S state. One after another, each S statement is transformed into the appropriate SJL or SCL statement.

5. Compiling SCCharts

High- & low-level synthesis (prio) SJL code generation SCL code generation

The screenshot displays the SCCharts simulation environment with three main panes:

- Diagram Pane (Left):** Shows a state transition diagram for the AO example. It features two states: 'wa' (waiting) and 'da' (done). A transition from 'wa' to 'da' is labeled 'A / O = true'. The diagram also indicates 'input bool A' and 'output bool O = false'.
- Synchronous Program AO (1) Pane (Middle):** Shows the SJL (System Java Language) code generated from the diagram. The code defines a state 'wa' and a transition function 'trans(wa)' that sets 'O = true' when 'A' is true.
- AOProgram Pane (Right):** Shows the SCL (System C Language) code generated from the SJL. This code includes a 'public AO()' constructor, a 'protected final void tick()' method that implements the state transition logic using a 'while' loop and a 'switch' statement, and a 'void printOutputs()' method for debugging.

Figure 5.4.6. Code generation (priority approach) from S intermediate language to SJL and SCL for the AO example

5.5. Circuit-Based Low-Level Compilation

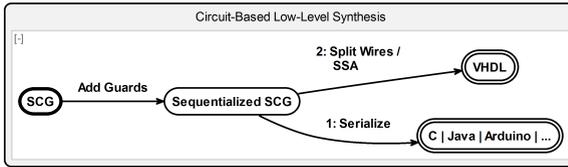


Figure 5.5.1. SCCharts compilation tree (cf. Figure 5.0.1 on page 114):
Circuit-based low-level synthesis part

5.5 Circuit-Based Low-Level Compilation

As mentioned earlier and introduced in Section 5.0.2 on page 115, there are two principle compilation paths to follow for low-level synthesis of SCCharts. Figure 5.4.1 on page 264 gives an overview of SCCharts and SyncCharts compilation paths in particular showing the Circuit-based path (cf. Figure 5.5.1) where C or Java code is generated that can be compiled and executed without any kind of additional runtime. The circuit-based approach also utilizes S as an intermediate, already low-level representation. Section 5.4.2 gave details about the S intermediate format. This section will give additional details about the circuit-based low-level compilation approach to purely sequential S code. This approach is used for our default SCCharts compile chain to C (cf. green dashed line in Figure 5.4.1 on page 264). However, more details to the priority-based low-level compilation can be found elsewhere [SMvH15, Smy13].

5.5.1 S Code Generation

The priority and the circuit-based approach both share S as an intermediate representation. The circuit-based approach makes neither use of different states, nor of fork and prio instructions of S. The sequentialization of an SCG [SMvH15] finds one of possibly many static schedules, where all *write-before-read* dependencies and in particular the absolute writes before relative writes constraints (cf. Figure 2.7.2 on page 42) have been satisfied for concurrent parts while respecting the sequential order for non-concurrent parts.

5. Compiling SCCharts

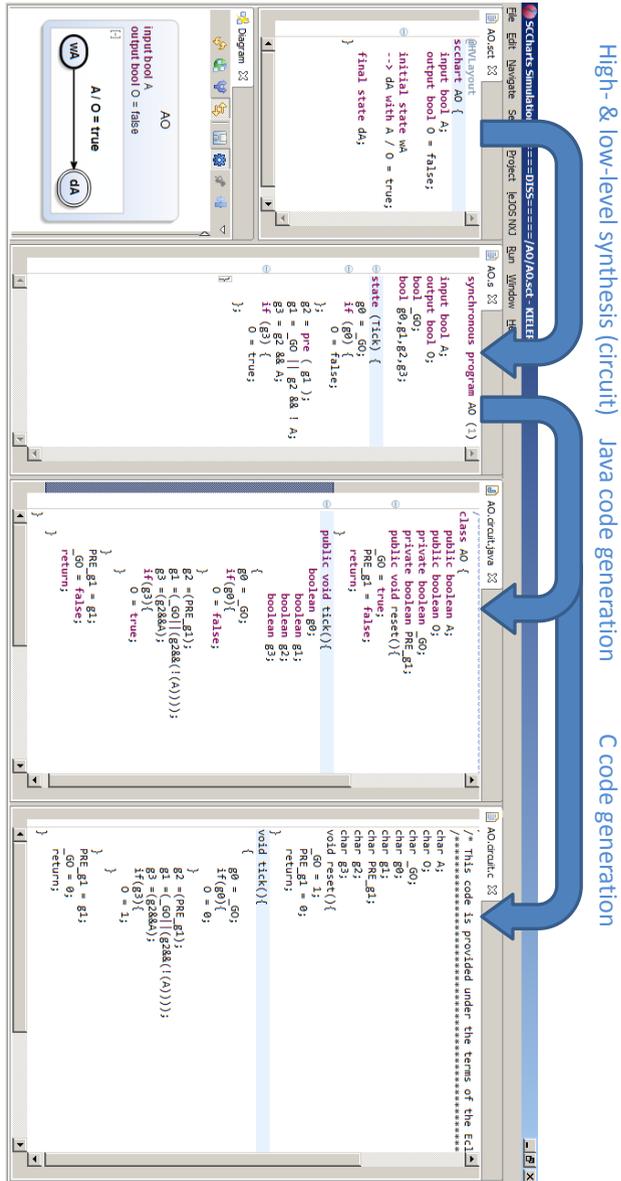


Figure 5.5.2. Code generation (circuit approach) from S intermediate language to Java and C code for the AO example

5.5. Circuit-Based Low-Level Compilation

Figure 5.5.3 shows the sequentialized SCG for ALDO (cf. Figure 5.5.3b) and for AO (cf. Figure 5.5.3a). For AO, as shown in Figure 5.5.3a, note that the first guard `g0` is true only in the initial tick because `_GO` is. Hence, `g0` guards the initialization of `O` to false which is scheduled/ordered before possibly setting `O` to true with guard `g3`. It becomes true once the transition is taken. Guard `g2` always gets the value of guard `g1` of the previous tick. Essentially, `g2` helps to implement the delayed transition (cf. guard `g3`) to never be enabled in the initial tick. Note that with the delayed transition, the initialization (`g0`) and taking the transition (`g3`) can never happen in the same tick. Consequently, there are more possible schedules than the one shown in the figure, e.g., the conditional with `g0` and its if branch could be moved/scheduled further down. The sequentialized SCG for the ALDO example shown in Figure 5.5.3b is a little bit more complex but follows the same principles as the sequentialized SCG of AO. The S code generation shown in Figure 5.5.2 results basically from encapsulating the tick logic in one initial state called Tick.

5.5.2 Java/C Code Generation

As mentioned earlier and shown in Figure 5.4.1 on page 264, the default SCCharts compiler (SCCharts) generates sequential S code. Also, for the Java/C code generation two methods are created: `A tick()` and a `reset()` method. `reset()` resets the SCChart before execution begins and `tick()` is called repeatedly to compute a reaction for each synchronous tick.

From the S code it is easily possible to derive both, Java and C code. Figure 5.5.2 shows this for the AO SCCharts example and reveals that the code generation is straight forward.

For Java and C, a reset function/method resets the guards as well as pre-guards and a tick function/method encapsulates all guarded expressions that were part of the tick S state. At the end of the tick function/method, the pre-guards are updated accordingly.

5. Compiling SCCharts

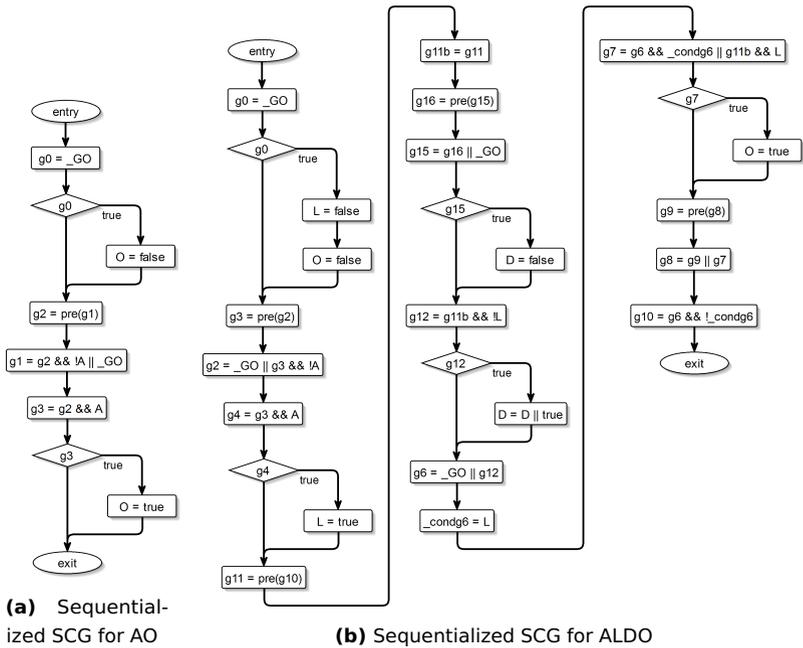


Figure 5.5.3. Sequentialized SCG examples

5.6 Compilation Design Choices

When compiling SCCharts, numerous design choices appear such as the specific high-level transformations and their dependencies (cf. Section 5.2 on page 123). Additionally, the chosen low-level path makes a difference, e. g., making use of dynamic priority scheduling or a static circuit-based low-level synthesis. Finally, more or fewer features can be eliminated by high-level transformations. If fewer features remain, e. g., only Core SCCharts features, then this conforms to a RISC approach. If more features remain, e. g., Core SCCharts features plus aborts and suspend, then this conforms to a CISC approach. These design choices were explored. Results are presented and discussed in the following paragraphs and are summarized in Table 5.6.1.

5.6. Compilation Design Choices

Table 5.6.1. Design choices for SCCharts compilation

	Priority	Circuit	CISC	RISC
Predictable timing	-	+	+/-	+/-
Predictable memory	-	+	+/-	+/-
Efficiency w. r. t. timing	+	-	+/-	+/-
Efficiency w. r. t. memory	+	-	+/-	+/-
Less compiling effort	+	-	+	-
Less runtime effort	-	+	-	+
Platform independence	-	+	-	+
Generate HW circuit	-	+	-	+
Generated code size	+	-	+	-
Model size	+	-	+	-
Executable size	+/-	+/-	+/-	+/-
Execution speed	-	+	+	-

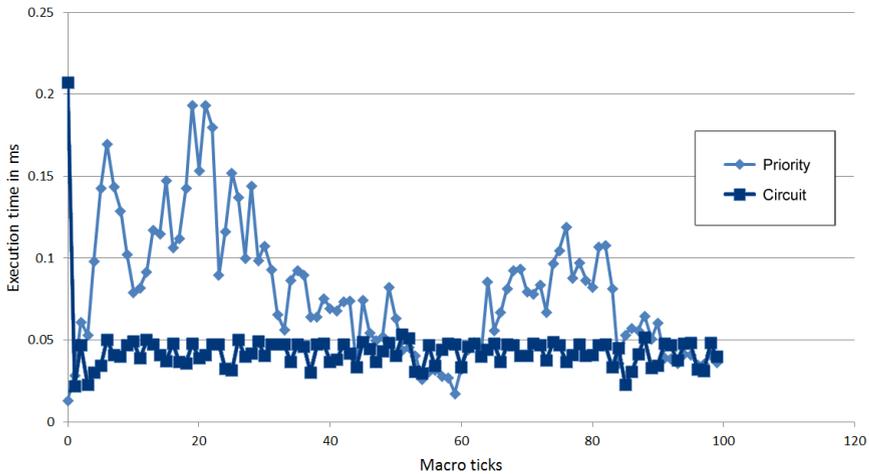


Figure 5.6.1. Circuit vs. priority SCCharts compilation: Execution time (from [Smy13])

5. Compiling SCCharts

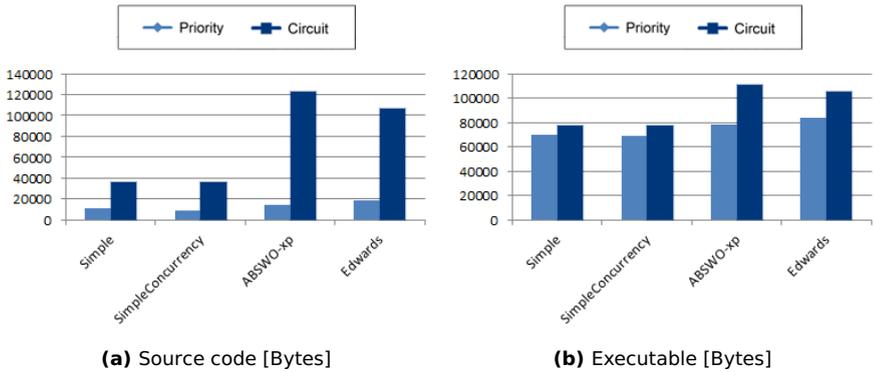


Figure 5.6.2. Circuit vs. priority SCCharts compilation: Code size (from [Smy13])

5.6.1 Circuit vs. Priority

Smyth [Smy13] compared the earlier priority-based SyncCharts/SCCharts compilation to an early version of the current circuit-based approach. Note that the early circuit-based compiler applied almost no optimizations to the code. The comparison of Smyth included measurement of execution times as shown in Figure 5.6.1. Interestingly, the circuit-based approach (light blue) proved itself to be roughly constant in its execution time while the priority-based approach (dark blue) has significantly more jitter.

Reasons for this are that in the circuit-based low-level approach, all resulting (guard) expressions are evaluated in each tick. These guard expressions correspond to wires in a digital circuit. Hence, a value for each wire is calculated for every tick, no matter if this causes internal or external reactions. The opposite is true for the priority-based approach. The code is organized such that the scheduler runtime always knows which SW threads are active and which of them still need processing for a tick. Processing only occurs for necessary parts of the code. For conditional expressions (and possibly further conditional sub-expressions) this means to execute code only in the if or in the else branch but never in both. In contrast, the

5.6. Compilation Design Choices

circuit-based approach executes both branches but only uses the results of one. The consequence for the priority-based approach is a high variety of reaction times for different ticks/inputs as shown in Figure 5.6.1.

Advantages of the priority-based approach are that for some examples and some ticks the reaction time can be far more efficient because only parts of the code need to be evaluated/executed. This not only affects execution time but also the needed memory. Nevertheless, this volatility in execution time and memory requirements implies unpredictability for the priority-based approach in both areas.

When compiling with the priority-based approach, the structure of an SCChart could be kept, which eases compilation. However, a runtime which does the low-level dispatch based on priorities is necessary. In contrast, when compiling with the circuit-based approach, the structure of the SCChart is broken up and the ordering of the code is guided by read-write-dependencies which already include a low-level static schedule. Therefore, a runtime simply needs to evaluate assignments, expressions, and conditionals in a static order and does not need any dynamic dispatching mechanism. A further consequence is that the resulting circuit-based code is more platform independent and that it can be more easily adapted because capable runtimes exist for most universal processing systems. Deriving a HW circuit is also eased if the code is already organized in terms of conditional assignments. As Figure 5.6.2 suggests, the code size in the priority-based approach is significantly smaller than in the circuit-based approach because the common priority-handling and dispatching infrastructure is not part of the code. However, it is still part of the executable such that, in the end, the binary sizes do not differ much.

5.6.2 RISC vs. CISC

As already sketched in the introduction, when designing the high-level and low-level transformations it is essential what features are supported by the runtime. We experimented with two options here:

1. A runtime that supports weak and strong preemption. We will call this option CISC as such a runtime needs more complex bookkeeping to handle executions.

5. Compiling SCCharts

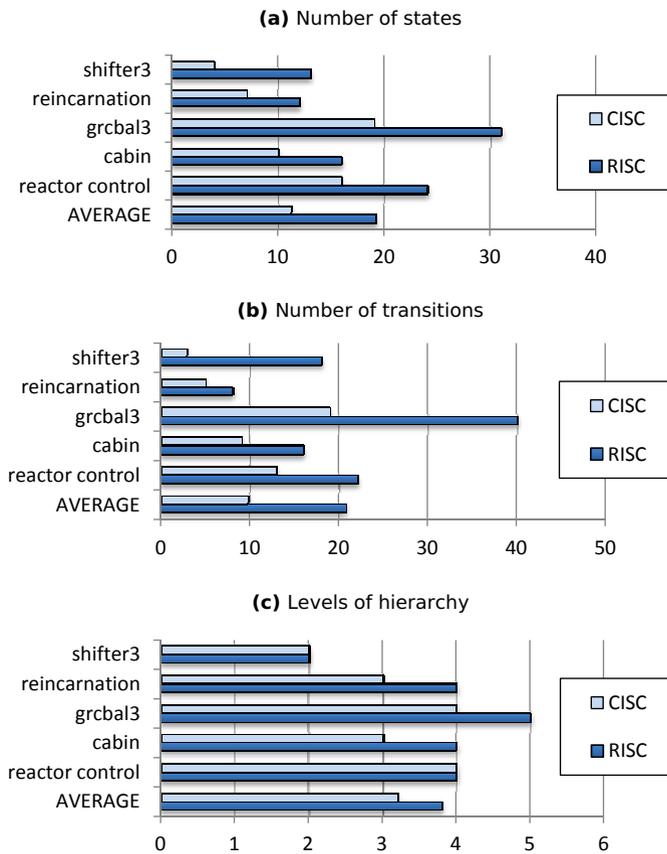


Figure 5.6.3. Comparison of Extended SCCharts (CISC) with equivalent Core SCCharts (RISC) resulting from transformations (partly from [vHDM⁺13c])

5.6. Compilation Design Choices

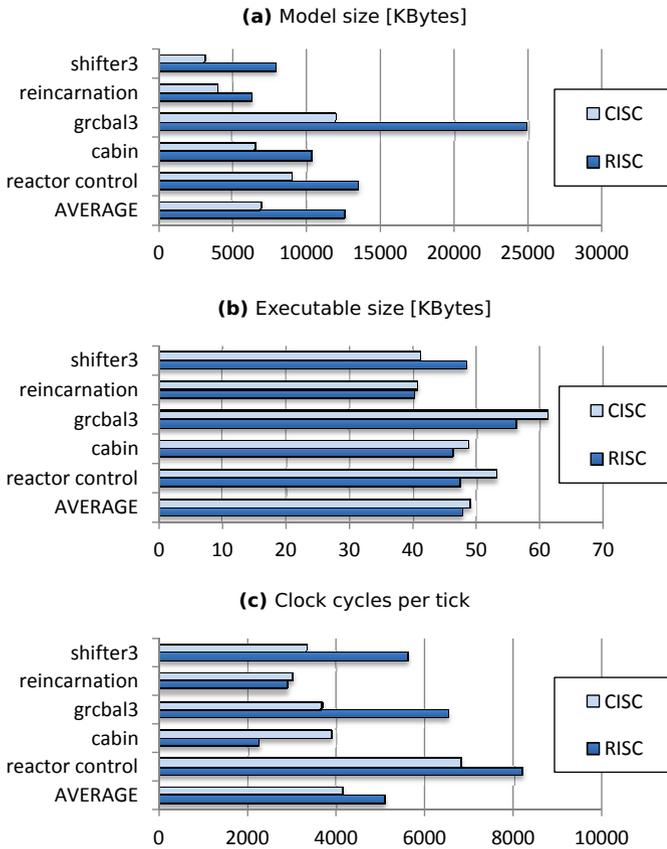


Figure 5.6.4. Comparison of code synthesis of Extended SCCharts directly to Synchronous C (CISC) with synthesis to SCL via transformations to Core SCCharts (RISC) (partly from [vHDM⁺13c])

5. Compiling SCCharts

2. A runtime that supports Core SCCharts features only. We will call this option RISC as it is a very restrictive minimal set of constructs which requires hardly any bookkeeping in the runtime.

We did several experiments as shown in Figures 5.6.4a and 5.6.4. The CISC approach in these experiments was based on the SyncCharts/SCCharts to SC compiler (SyncCharts2S) as described in Section 5.1 on page 116. The RISC approach in these experiments was not based on the current SCCharts compiler (SCCharts) to not bias the results with advantages of the circuit-based approach over the priority-based approach, e. g., that no low-level runtime would be necessary. The RISC approach in these experiments was also based on the SyncCharts2S compiler but with another high-level synthesis path that eliminates aborts (as it is currently done in the SCCharts compiler) and hence by explicitly not making use of the abort-handling capabilities of the SC runtime.

Figure 5.6.4 shows the results of counting numbers of states, transitions, and hierarchy levels of the model that can finally be executed by the runtime. While the number of hierarchy levels does not differ much, the number of transitions and states is roughly doubled in the RISC approach compared to the CISC approach. This is because the complex abort-handling is expressed explicitly in the model in terms of terminations and additional states and transitions in the RISC synthesis path. Consequently, the expanded model in the RISC approach is larger and so is the resulting code (cf. Figure 5.6.4a).

Interestingly, the execution binary is still hardly affected although the abort-handling capable runtime code is still included in the RISC synthesis path (cf. Figure 5.6.4b).

The execution speed per tick of the RISC approach using the priority-based low-level synthesis path, is significantly lower than the CISC approach as Figure 5.6.4c illustrates. However, speed drawback is still unexpectedly small if keeping in mind that the preemption processing of the runtime is still running in the background, but simply not used by the code. Hence, the RISC code is not optimal for running on a CISC runtime infrastructure. In contrast, the RISC code is optimal for running as a circuit with no additional runtime. Still, if comparing again the runtimes of the priority-based and the circuit-based low-level synthesis (cf. Figure 5.6.1) then the execution

speed is significantly more efficient also for the RISC approach. However, then there still will be evaluation of both, the if and the else branches as explained earlier.

5.6.3 Defaults for KIELER SCCharts Compiler Implementation

The current SCCharts compiler evolved as discussed in Section 5.1 on page 116. It is based on a circuit-based RISC compilation strategy. This compensates execution time drawbacks of the RISC approach as explained above. The circuit-based low-level synthesis also evolved as the default for SCCharts compilation, because time and memory predictability is key for targeting safety-critical embedded real time systems. Resolving time and memory efficiency drawbacks is an area of interest for future work.

5.7 SCCharts Targets

SCCharts is a high-level synchronous modeling language, primarily suited for specifying safety-critical reactive systems. Chapter 5 presented how SCCharts can be compiled incrementally by M2M transformations. Often, the goal is to execute SCCharts on a target hardware of a reactive system. As Figure 5.7.1 suggests, there are several scenarios.

For Development, often an IDE with an integrated software simulation is favored because development cycles are fast and debugging is eased. In the context of the KIELER SCCharts tool, the integrated C or Java simulator can be used for that purpose. C or Java code is generated as part of the low-level synthesis from SCGs.

For Prototyping, slightly longer development turn around cycles are often acceptable, but therefore prototyping better approximates productive operation. A general purpose Microcontroller, such as the Arduino platform, can be used or FPGAs that conduct the dedicated functionality. SCCharts can be compiled from their SCG representation into a Static Single Assignment (SSA) form [CFR⁺91] and from there VHDL code for FPGAs can be acquired.

5. Compiling SCCharts

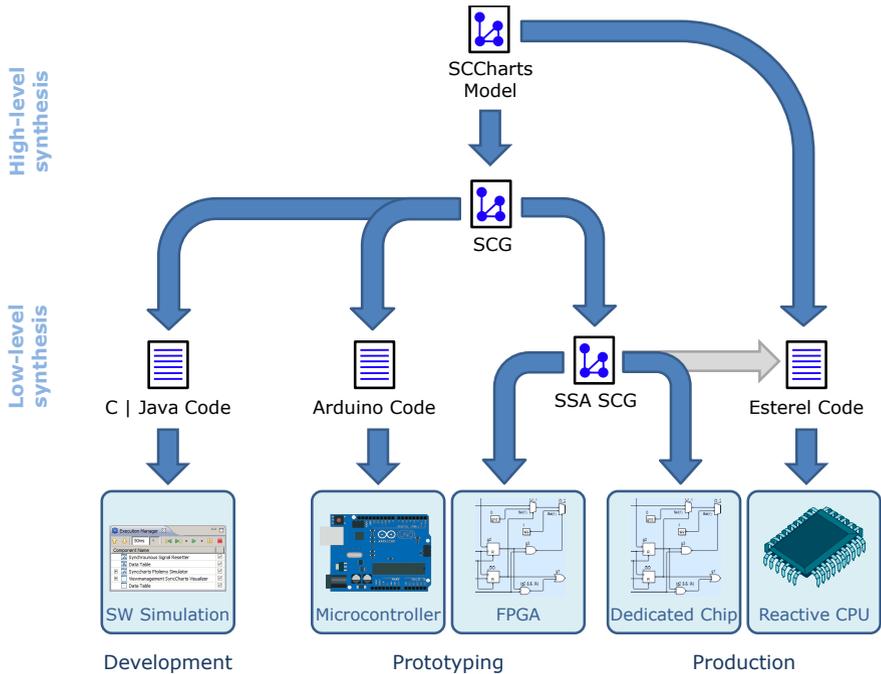


Figure 5.7.1. SCCharts SW and HW targets

The SSA form also permits to infer a dedicated HW circuit for mass production. KIELER allows to compile SCCharts into HW circuits that can be already viewed and inspected during modeling phase. An alternative to dedicated HW chips are specialized reactive processors that can run arbitrary reactive programs more or less directly. For that purpose KIELER exemplarily allows to transform SCCharts into the reactive synchronous language Esterel for which such reactive processors exist (see also Section 5.7.2).

The following sections summarize these possible target scenarios for SCCharts and the projects that have dealt with enabling the KIELER SCCharts implementation to suit all these scenarios.

5.7.1 Software

Since a long time C has established itself as a widely used programming language that is heavily utilized especially in the context of embedded and reactive systems [Bar99]. Also emerging are embedded platforms that have a Java runtime environment [LHS10] or which are able to process Java natively [Sch03].

For that reason especially C and Java are also code generation targets for higher-level synchronous modeling languages such as SCADE [Est16] or Esterel [GR83]. Hence, also SCCharts targets C and Java primarily for software simulation but also for prototyping and productive operation on general purpose embedded HW platforms like Arduino [Rus10] or Lego Mindstorms³ [LHS10].

C

As Figure 5.4.1 on page 264 suggests, the SCCharts compiler by default produces circuit-based C code which can be compiled, e. g., by using the gcc⁴ to an executable. This executable can be run and interacted with by stimulating it with input data, executing the tick function, and observing the output data. This is the way it is used within the KIELER SCCharts tool for simulating SCCharts. The tooling around SCCharts is further studied in the next chapter, Chapter 6.

The C-like Arduino code can be also used directly to program embedded HW targets such as the Arduino platform. Further details and examples are given in Section 5.7.2.

Finally, the C code can be used to interact with embedded HW that has a C interface. Chapter 7 demonstrates how to use generated C code from SCCharts for controlling a larger model railway embedded system.

Java

Besides the default circuit-based C code generation from SCCharts, Figure 5.4.1 on page 264 also illustrates other alternative synthesis paths, e. g.,

³<http://mindstorms.lego.com>

⁴<http://gcc.gnu.org>

5. Compiling SCCharts

the priority-based approach or Java as a host language instead of C.

Section 8.4 on page 407 illustrates how Java can be used for embedded targets. The SyncCharts2S compiler, predecessor of the SCCharts compiler (cf. Section 5.1 on page 116), was already tightly integrated into the KIELER SCCharts tooling (cf. Chapter 6) and was able to simulate SyncCharts purely in Java without the need of an external C compiler.

5.7.2 Hardware

SCCharts target safety-critical reactive systems which often are embedded into a controlled environment. Typically, these systems consist of multiple sensors that observe the controlled environment, a microcontroller which does the reaction computation based on the sensor information, and several actuators that affect the controlled environment as illustrated by the introducing Figure 1.0.2 on page 3.

Reconsider Figure 5.7.1. For development, the microcontroller is often replaced by a general purpose computer software simulation as integrated in the KIELER tool. For prototyping and productive operation, there are several possibilities where the reaction computation can be accomplished. To name the most common ones:

- ▶ General purpose microcontroller
- ▶ Special reactive microcontroller
- ▶ Dedicated controller chip

A very popular and successful representative of a general purpose microcontrollers is the Arduino⁵ platform. Other similar and comparable projects are the very cheap MSP430 LaunchPad⁶, the STM32 Discovery⁷ which has notably many I/O ports, and the extremely small-sized Teensy⁸ controller. We adapted our code generation for the Arduino platform and built the ABROINO demonstrator which will be discussed in this section.

⁵<http://www.arduino.cc>

⁶<http://www.ti.com/tool/msp-exp430g2>

⁷<http://www.mouser.com/ProductDetail/STMicroelectronics/STM32VLDISCOVERY>

⁸<http://www.adafruit.com/products/731>

5.7. SCCharts Targets

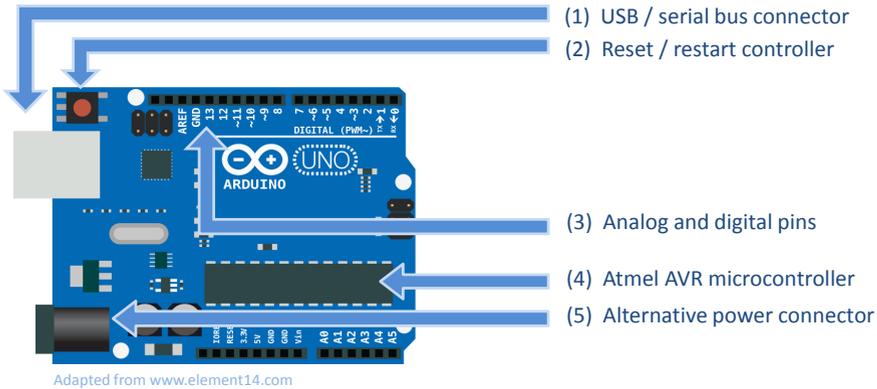


Figure 5.7.2. Arduino microcontroller platform

The Kiel Esterel Processor (KEP) [LvH12] is a special reactive processor for the Esterel language that was developed in Kiel, equipped with dedicated deterministic multi-threading. A similar processor is the STARPro [YAY⁺08] developed in Auckland. In order to use such a *reactive processor* [Tra07], a synchronous target language is necessary. Since SCCharts is already a synchronous language, this requires least effort. Exemplarily, we implemented a transformation from SCCharts to Esterel which potentially enables to use a processor like the KEP for executing numerous SCCharts on a special reactive microcontroller chip. The transformation from SCCharts to Esterel is sketched in Section 5.7.2 on page 296.

FPGAs have become very popular. These are integrated circuits that compute logical functions and which can be programmed. Hence, they serve as good prototyping platforms before a dedicated HW chip is mass produced. We show how a digital circuit can be derived from SCCharts and used together with our interactive SLIC-based compiler. This digital circuit is shown graphically as another live-updated transient view w. r. t. the currently modeled SCChart. We also show how to conceptually derive FPGA code from the circuit (see Section 5.7.2 on page 298).

5. Compiling SCCharts

Table 5.7.1. Advantages of using SCCharts for modeling Arduino software vs. programming the C-like code directly

	Code	SCCharts
Low-cost platform	+	+
Established COTS HW	+	+
Open-source	+	+
Learning curve	+/-	+
Maintenance	-	+
Concurrency	-	+
Deterministic behavior	-	+
High-level modeling	-	+
Model checking possible	-	+
Simulation	-	+
Control-flow nature	-	+
Platform independence	-	+
Manual optimizations	+	-

Arduino

Since 2005, Arduino is a low-cost, open-source project of an embedded microcontroller board and an open-source, platform independent, Java-based IDE. The typical Arduino board hosts an 8, 16, or 32 bit Atmel AVR microcontroller which comes in different versions with different speeds, different form factors and different numbers of analogous and digital I/O connectors. Arduino boards are usually programmed via a serial connection. The Arduino IDE allows to write microcontroller software in a C-like language. Any such program consists of two functions, `setup()` and `loop()`. The `setup()` function is called at startup or after a reset to initialize the program. Afterwards, the `loop()` function is called repeatedly while the board is powered.

SCCharts Arduino Modeling: Since both, the controller board HW and the programming environment SW are open-source, projects that build upon Arduino can be used commercially but also academically and do

not require any fees. When developing Arduino-based projects and programming with the C-like language to build software for the embedded microcontroller, one can profit from all these benefits. However, using SCCharts for modeling the microcontroller software may further significantly boost a project and Table 5.7.1 summarizes some of these additional benefits. Although the learning curve for programming Arduino with code may be acceptable for people that already are familiar with C or Java, modeling graphical state machines is supposed to be learned comparably fast. Furthermore, the number of core features is quite small and can be enlarged step-by-step with extended features without limitations to the expressiveness from the beginning. Maintaining larger SCCharts is supposed to be easier because SCCharts can be inspected not only in the textual SCT format but also graphically with many filtering features that help to get an overview of the project or parts of it.

The Arduino itself is not equipped with any built-in concurrency implementation such as threads. This makes it hard to express concurrency manually by implementing explicit interleaving in a *Cyclic Executive*-fashion [Loc92]. In contrast, SCCharts naturally allow to express deterministic concurrency and the compiler takes care of producing explicit interleaved code. In general, as a synchronous language, SCCharts inherently react deterministically. All needed inputs are read before the tick method computes the reaction and all outputs are written afterwards. In ordinary Arduino code it might be possible to observe non-deterministic behavior, e. g., w. r. t. different timings of sensors that are requested within the `loop()` function. Opposed to programming low-level C-like code, SCCharts as a modeling language, is designed for high-level specifying abstract graphical models. This abstraction separates functionality from actual timing. Such abstract models allow for model checking system safety and liveness properties. Additionally, such models can be easily compiled to other languages such as C or Java or transformed to other modeling languages such as Ptolemy. This enables simulation, which boosts development (and maintenance) of embedded devices where turn around cycle times and testing on the target platform typically are more time consuming. Reactive systems often involve dealing with system states and control-flow.

5. Compiling SCCharts

```
1 int inputPinA = >>ENTER_INPUT_PIN_HERE<<;
2 int outputPinO = >>ENTER_OUTPUT_PIN_HERE<<;
3 int A;
4 int O;
5 // ----- VOLATILE GENERATED CODE BELOW -- DO NOT EDIT -----
6 int _GO;
7 int g0; int g1; int g2; int g3; int g4; int g5;
8 int PRE_g4; int PRE_g1;
9 // ----- VOLATILE GENERATED CODE ABOVE -- DO NOT EDIT -----
10 void setup(){
11     pinMode(inputPinA,INPUT);
12     digitalWrite(inputPinA,HIGH);
13     pinMode(outputPinO,OUTPUT);
14     _GO = 1;
15 // ----- VOLATILE GENERATED CODE BELOW -- DO NOT EDIT -----
16     PRE_g1 = 0;
17     PRE_g4 = 0;
18 // ----- VOLATILE GENERATED CODE ABOVE -- DO NOT EDIT -----
19     return;
20 }
21 void loop(){
22     // Read inputs
23     A = digitalRead(inputPinA);
24 // ----- VOLATILE GENERATED CODE BELOW -- DO NOT EDIT -----
25     {
26         g0 = _GO;
27         if (g0) {
28             O = 0;
29         }
30         g2 = PRE_g1;
31         g1 = (g2&&(!A))||_GO;
32         g3 = g2&&A;
33         if (g3) {
34             O = 1;
35         }
36         g5 = PRE_g4;
37         g4 = g5||g3;
38     }
39     PRE_g1 = g1;
40     PRE_g4 = g4;
41 // ----- VOLATILE GENERATED CODE ABOVE -- DO NOT EDIT -----
42     _GO = 0;
43     // Write outputs
44     digitalWrite(outputPinO,O);
45     return;
46 }
```

Listing 5.7.1. Arduino code generated from AO SCCharts example

These system states often need to be represented somehow in the software. As SCCharts is a Statechart dialect, it naturally allows for expressing systems states directly. The C-like Arduino code specifically works for Arduino HW platforms. If one decides to use another microcontroller platform, rewriting of the code is required. SCCharts benefit from its higher abstraction level and generally also other microcontroller platforms could be integrated with the SCCharts tooling such that the concrete SCCharts model hardly needs any modification. However, as SCCharts is a modeling language with a higher abstraction level than the C-like code, special and individual low-level optimizations may not be realizable as straight forward as they may be when programming low-level code directly.

Adaptive Code Generation from SCCharts: In order to generate code for the Arduino, the existing C code generation for SCCharts was re-used for most parts. The `setup()` function essentially corresponds to the `reset()` function of the C code generated from SCCharts, and the `loop()` function corresponds to the repeatedly called `tick()` method.

Listing 5.7.1 shows the generated Arduino code for the AO SCChart (cf. Figure 5.2.78 on page 231). Special comment boundaries mark the volatile parts of the generated code. The non-volatile parts are meant to be preserved if the code is re-generated and an existing INO file is found in the same directory. The non-volatile sections also contain the configuration part, e. g., the mapping of SCCharts inputs and outputs to physical Arduino pins, but also the setup configuration where the pins can be further configured. The volatile sections contain only the relevant parts for resetting the SCChart and performing synchronous ticks. If the *adaptive code generator* for the Arduino does not find an INO file in the same directory, all non-volatile parts are also freshly generated but may need further configuration as in the code shown in Listing 5.7.1. In lines 1 and 2, the actual pins of the Arduino need to be mapped to the input A and the output O of the AO SCChart. Note that the volatile parts are updated according to any SCChart changes. Also, note that changes to the SCChart's interface may require manual adjustments of non-volatile parts in the generated Arduino code.

5. Compiling SCCharts

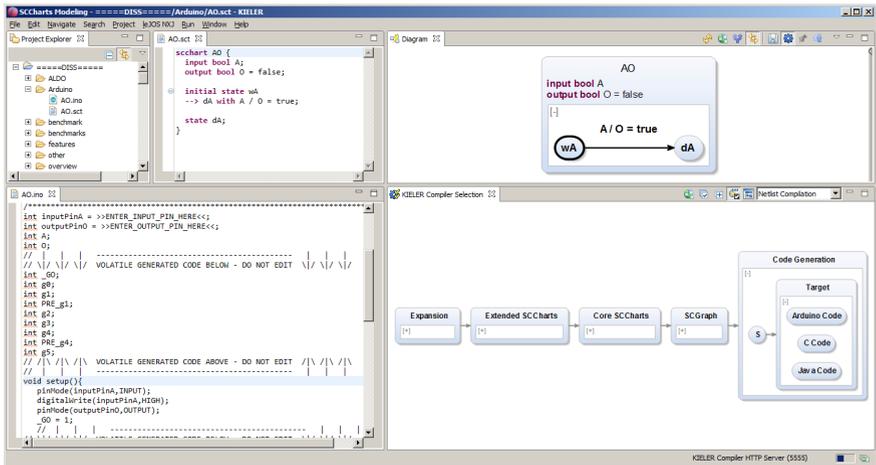


Figure 5.7.3. Adaptive code generation with KIELER for the Arduino

Arduino Development in KIELER: Figure 5.7.3 shows the AO SCCharts example (upper part) and the Arduino code (lower left part) generated by the KIELER compiler. The interactive compiler selection (lower right part) allows to select the Arduino target for code generation from S intermediate code. As seen there, the Arduino code generator is implemented as a compiler back end alternative to C or Java code. Note that the code generator is *adaptive* w. r. t. a possibly existing INO file in the same directory of the model file and with the same name (upper left part). All non-volatile parts of the generated code are taken from this file upon its existence, i. e., pin to input or output variable mappings.

The user story for modeling Arduino code with KIELER SCCharts and bringing it onto the embedded target is sketched in Figure 5.7.4. The KIELER SCCharts tool allows for simulating the SCCharts by using the integrated SCCharts-C code simulator as discussed later in Chapter 6. After validating that the SCChart functions as intended, the Arduino Code generation target is selected (cf. Figure 5.7.4a) in order to produce Arduino code that can be saved as an INO file, typically in the same directory as the SCCharts model. Non-volatile code can be modified within that INO file,

5. Compiling SCCharts

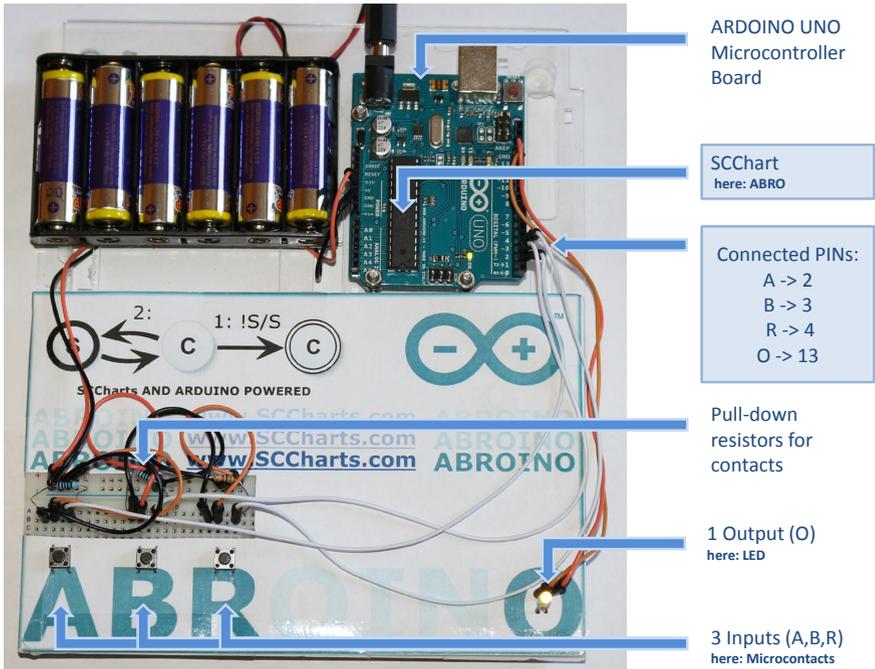


Figure 5.7.5. The SCCharts-Arduino demonstrator: ABROINO

and these changes are maintained in subsequent Arduino code generation calls while the volatile code parts are updated according to possible SCChart changes. The INO file can be opened directly in the Arduino IDE as shown in Figure 5.7.4b. This IDE also assists in downloading the generated code to the actual embedded Arduino board.

The ABROINO Demonstrator: The ABROINO is an Arduino demonstrator as shown in Figure 5.7.5. It consists of an externally battery-powered Arduino that is connected to three microcontacts marked as A, B, and R as digital inputs and an LED marked as O as a digital output.

5.7. SCCharts Targets

The screenshot displays the KIENER Model View (AO.act) interface. The main window shows the AO.strl code, which defines a module AO with inputs A and O, and outputs signal, s176889379, and regional16348257. The code includes a synchronous signal register, a state machine with states 'initial state WA' and 'state dA', and a trap mechanism for the 'dA' signal.

```

module AO;
  input A;
  output O;
  signal s1568150151;
  s176889379;
  regional16348257 : boolean in
  trap trapBone in
  {
    exit s1568150151;
    loop
      trap trap116348257 in
      present s1568150151
      then
        signal surface1568150151 in
        exit surface1568150151;
        trap trap1568150151 in
        nothing
        ||
        loop
          pause ;
          present A
          then
            exit O;
            exit trap1568150151;
          end
        end;
        nothing
        }
      end;
      present surface1568150151
      else
        exit trap116348257
      end
    end
  }
end

```

The interface also shows a component list on the left, a synchronous signals diagram, and a state machine diagram on the right. The state machine diagram shows a transition from state 'WA' to state 'dA' when the condition 'A/O = true' is met.

Figure 5.7.6. SCCharts to Esterel: Simulating generated AO.strl example using the KIENER CEC-based Esterel simulator

5. Compiling SCCharts

The Arduino is able to run an arbitrary SCChart that has an interface with these boolean variables *A*, *B*, *R*, and *O* and where the correct mapping to the pins 2, 3, 4, and 13 is configured in the non-volatile code section of the INO file. By default, the demonstrator is running the ABRO SCChart (cf. Section 2.3.1 on page 26), i. e., the *hello world* of synchronous programming. The purpose of the ABROINO demonstrator is to show how easily SCCharts can be brought on a widely used embedded target with all benefits as discussed in Table 5.7.1.

KEP: SCCharts to Esterel

Using a special reactive processor like the KEP [LvH12] that is able to process arbitrary Esterel programs, combines the advantages of a dedicated

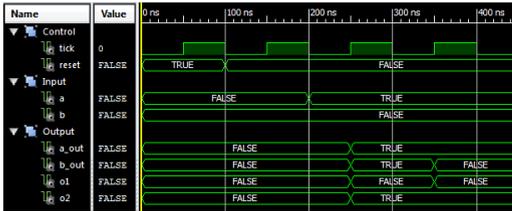
HW with fast development turn around cycle times. However, a drawback usually is that such processors are not widely used. As a consequence, the technology that they base on rarely is state-of-the-art.

To show that conceptually SCCharts are suited to be executed on such platforms, we implemented a prototype of an SCCharts to Esterel compilation. It is fully described in the work of Nasin [Nas15]. Figure 5.7.6 presents a screenshot of the KIELER tool running the resulting Esterel code for the AO example with an Esterel simulator. Part of the Esterel code is shown in the center of the figure. This approach is based on ideas of André [And95] who compiled SyncCharts, the predecessor of SCCharts, to Esterel code.

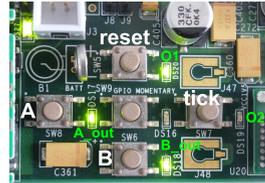
It can be observed that this approach leads to large Esterel programs even for small models as the AO example.

Hence, it is practically hardly usable, especially for embedded systems that typically have limited resources. It is future work to come up with a more compact and efficient transformation. Nevertheless, the approach and implementation still show that, conceptually, SCCharts can be compiled into other synchronous languages. Of course even a special reactive processor for executing SCCharts could be envisioned. It is noteworthy that the current prototype only accepts Berry-constructive SCCharts for compiling them to Esterel code. Using techniques as transforming sequentially constructive SCCharts to SSA-style-Esterel code, as presented elsewhere [RSM⁺15], should make it possible to loosen this limitation.

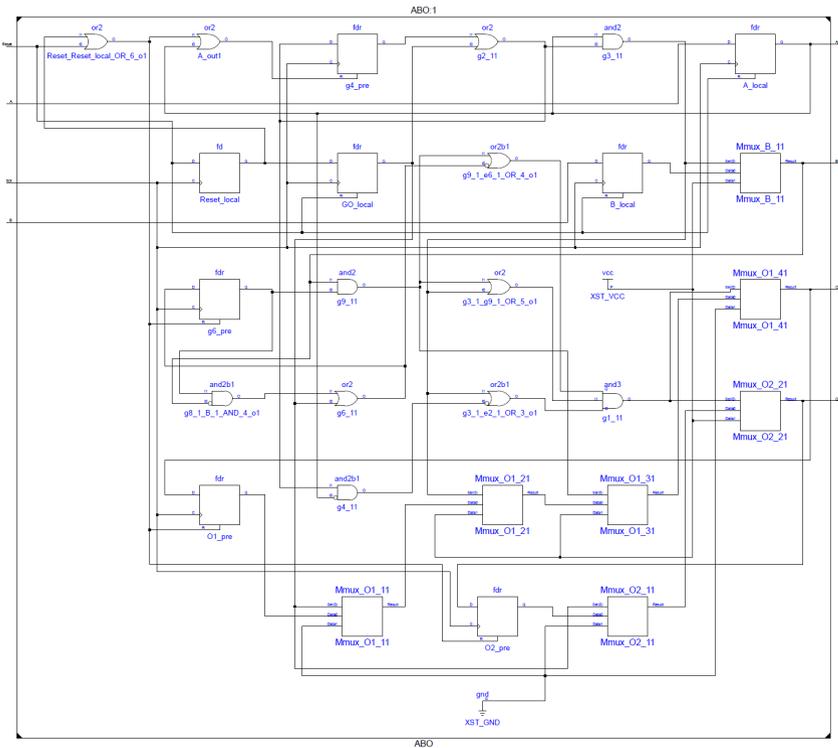
5.7. SCCharts Targets



(a) ABO simulated in the ISE tool



(b) FPGA running the compiled ABO example



(c) Synthesized circuit of the ISE tool for the ABO example

Figure 5.7.7. Running SCCharts on FPGAs: The ISE tool (from [Joh13])

5. Compiling SCCharts

VHDL and Circuits

The Integrated Synthesis Environment (ISE) from Xilinx⁹ is an IDE for synthesis and analysis of the Hardware Description Language (HDL) that also comes with a simulator. It can be used to compile VHDL code, download it, and run it on an FPGA.

SCL2VHDL Project: Johannsen [Joh13] targeted the ISE tool and integrated a first prototype of a VHDL code generator as a back end for the KIELER SCCharts compiler. As Figure 5.7.1 on page 284 indicates, VHDL code generation is possible from the SCG intermediate format.

Figure 5.7.7 presents the FPGA circuit created by and visible within the external ISE tool after feeding it with generated VHDL code. This VHDL code was compiled with KIELER from an SCCharts example termed ABO (cf. [vHDM⁺14]). Figure 5.7.7b demonstrates a running version of the compiled ABO on an Xilinx ML605¹⁰ developer board equipped with a Virtex-6 FPGA.

SCG2Circuit Project: The implementation of Johannsen [Joh13] was based on an earlier version of the SCG meta model or more specifically for a textual description of SCGs. Rybicki [Ryb16] continued the work of Johannsen including an implementation which fully integrates into the interactive incremental SLIC compilation approach. It is also based on the current SCG meta model.

Table 5.7.2 compares the two projects. Both projects deal with generating VHDL code or circuits from SCCharts and its intermediate SCG representation. Also, both projects offer a possible visualization of the resulting HW circuit. The earlier project generated textual VHDL code in KIELER but uses the external ISE tool for generating and visualizing the actual circuit, based on the generated VHDL code. The ISE tool is also able to directly connect to an FPGA which eases programming of FPGAs if this tool is already part of the tool-chain. The new SCG2Circuit project conceptually may also generate the same VHDL code that could be downloaded to an FPGA with ISE. A

⁹<http://www.xilinx.com/products/design-tools/ise-design-suite.html>

¹⁰http://www.xilinx.com/publications/prod_mktg/ml605_product_brief.pdf

5.7. SCCharts Targets

Table 5.7.2. Comparing the predecessor SCL2VHDL [Joh13] with the current SCG2Circuit [Ryb16] SCCharts circuit project

	SCL2VHDL project	SCG2Circuit project
VHDL/circuits from SCCharts	+	+
VHDL/circuits from SCGs	+	+
Visualize circuits	+	+
Directly program FPGAs	+	+/-
Open-source / no license needed	-	+
Seamless tool-chain	-	+
Circuits visible while modeling	-	+
Simulation visualization	-	+
Side-by-side co-simulation	-	+
Understanding circuits	-	+
Element tracing: SCChart ↔ circuit	-	+
Online and cmd line compiler	-	+
Re-use SSA representation	-	+
KIELER layout for circuits	-	+

drawback of having ISE be a required part of the tool-chain is that it is not open-source software and a license must be purchased. This means the tool-chain of the old project is not as seamless compared to the tool-chain of the new project, which generates and visualizes circuits entirely inside the KIELER tool. These circuits are also visible and continuously updated as transient lightweight diagrams while modeling, i. e., while editing and modifying the underlying SCChart. The circuit visualizations can also be used together with an SCCharts/SCG simulation component to help understanding the dynamics of the resulting HW circuit. A side-by-side simulation visualization of the underlying SCChart/SCG and the circuit not only eases understanding of modeled SCCharts and resulting circuits but also helps validating the correctness of the circuit transformation itself.

5. Compiling SCCharts

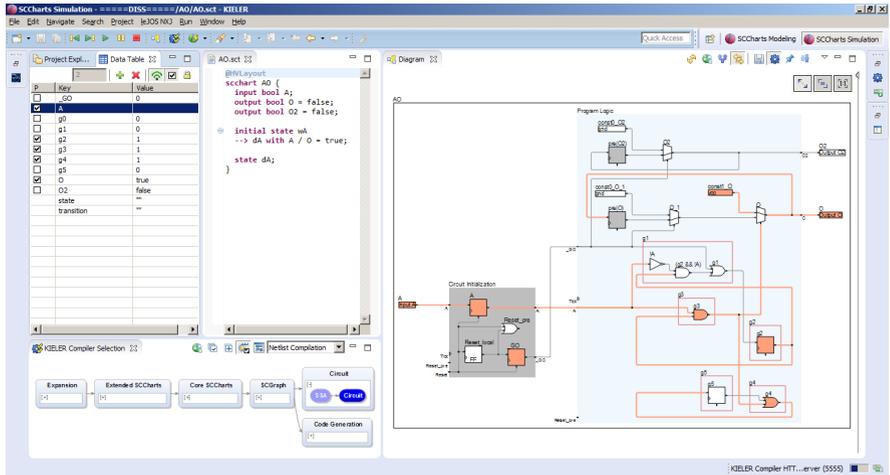


Figure 5.7.8. AO SCChart visualized as HW circuit inside the KIENER SCCharts tool during simulation

The new project focuses even more on gaining benefits from the white-box compiler approach w. r. t. understandability of the circuit and how parts of the SCCharts influence parts of the circuit. These enhancements are based on two facts: 1. The SSA transformation for the SCG generates an intermediate SSA SCG where each variable is only written to once. Such a variable will become a dedicated wire in the circuit which in each tick can have exactly one value. 2. Due to the seamless KiCo SLIC compiler integration, tracing of model elements is available which allows to actually see which elements of the circuit originate from which elements of the modeled SCChart. Both was not possible with the old approach. Because the new project is part of the SCCharts KIENER compiler, it can also be used together with an online compiler, with a command line compiler, and with other possible KIENER compiler integrations. The developed SSA representation for SCGs of the new project is conceptually based on results of its predecessor. However, as it seamlessly integrates into the current compile chain, it can easily be re-used, e. g., for generating SSA-style Esterel

5.7. SCCharts Targets

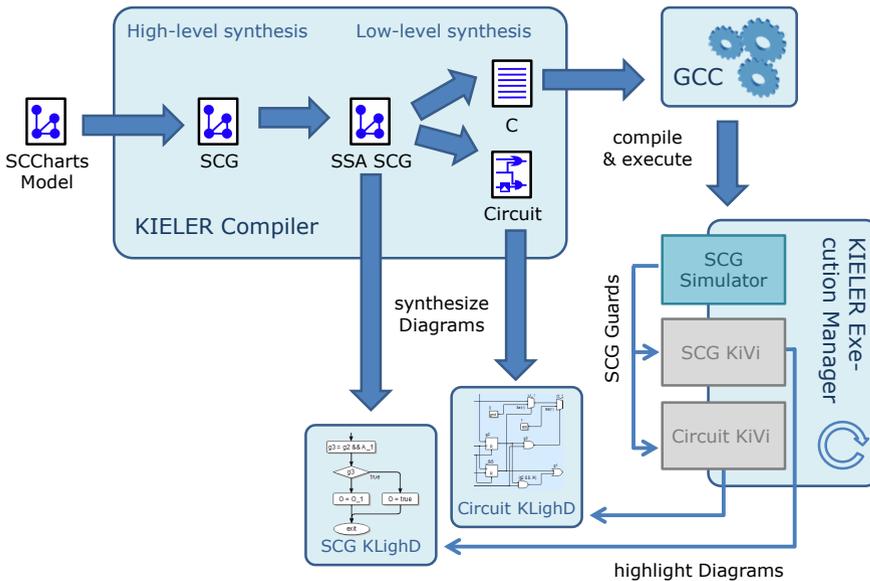


Figure 5.7.9. Simulation and visualization of SCCharts compiled to electrical HW circuits

or Lustre code from SCGs for alternative SCCharts code synthesis paths (see Figure 5.7.1 on page 284, light gray arrow from SSA SCG to Esterel Code). Finally, because the circuit visualization is part of KIELER, specially developed data-flow layout algorithms [SSvH14] can be utilized and also be validated/optimized for HW circuit design.

Circuit Simulation Visualization and Validation: Figure 5.7.9 shows the current integration of the HW circuit synthesis for simulation visualization and transformation validation purposes. SCCharts are compiled as described earlier in a high-level and low-level synthesis down to SCGs and further on to SSA SCGs, i. e., SCGs in SSA form. These can be easily transformed on into a Circuit and visualized with in a transient lightweight view using the KIELER Lightweight Diagrams (KLighD) [RSS⁺13] technology. The

5. Compiling SCCharts

SSA SCGs can also be compiled down to C code as usual. The SCCharts simulation component is also capable of simulating SCGs. It is labeled as SCG Simulator in Figure 5.7.9. Simulating an SCG means using the usual SCCharts compile chain but starting with an intermediate SCG model (cf. Section 8.2.3 on page 397). When simulating SCGs, the simulator additionally outputs Guards that can be mapped back to the SCG but also to elements of the circuit in order to visualize active/inactive parts or propagation of zeros/ones in the circuit. This highlighting task is fulfilled by the SCG KiVi visualization component for the SCG KLighD synthesized diagram and the Circuit KiVi visualization component for the Circuit KLighD synthesized diagram.

Further details on deriving and visualizing HW circuits from SCCharts and SCGs can be found elsewhere [Ryb16, RSM⁺16].

This concludes the SCCharts compiling chapter. The next chapter will give details on the KIELER SCCharts tooling, its implementation, and design decisions.

SCCharts Tooling

This chapter introduces the SCCharts tooling and its KIELER implementation together with notes on its evolution. The SCCharts tooling consists of the current SCCharts editor implementation, the KIELER Compiler (KiCo) implementation, and its automatic validation. Beforehand, other similar tools are sketched that had influences on the development of the SCCharts tooling.

6.1 Related Tools

This part will give an overview of other tools, compilers, and simulation integrations in the context of synchronous languages that influenced the development of the current SCCharts tooling.

6.1.1 Ptolemy

Ptolemy II [EJL⁺03, Lee03] is a Java-based framework that supports heterogeneous modeling, simulation, and design of concurrent systems. The Ptolemy II framework contains a graphical modeling editor called *Vergil*. Ptolemy II itself is discussed in more detail in Section 8.1.1 on page 381. This section gives an overview of the GUI of Ptolemy II and its model simulation capabilities. As *Ptolemy II* is the most current development, it will just be called *Ptolemy* throughout the rest of this thesis. Implicitly, the Java-based Ptolemy II is meant and *not* its C-based *Ptolemy Classic* predecessor.

6. SCCharts Tooling

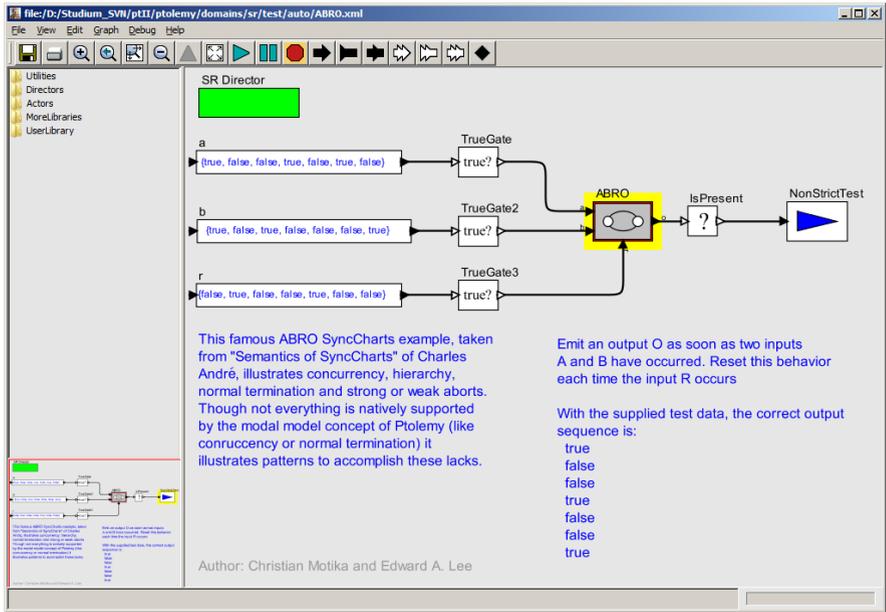


Figure 6.1.1. The Ptolemy graphical model editor *Vergil* with an opened model showing the top-level part of the Ptolemy version of ABRO

Ptolemy Vergil

Figure 6.1.1 shows the Ptolemy GUI *Vergil*, which can be used for developing, maintaining, and inspecting Ptolemy models.

Ptolemy models itself are persisted as XML. At runtime, they consist of instantiated Java classes. Hence, Ptolemy models can also be created and manipulated programmatically in various ways.

Vergil comes with an automatic layout assistant that uses layout algorithms provided by KIELER¹/Eclipse Layout Kernel (ELK)² [SFvH09, Fuh11].

¹<http://rtsys.informatik.uni-kiel.de/kieler>

²<http://www.eclipse.org/elk>

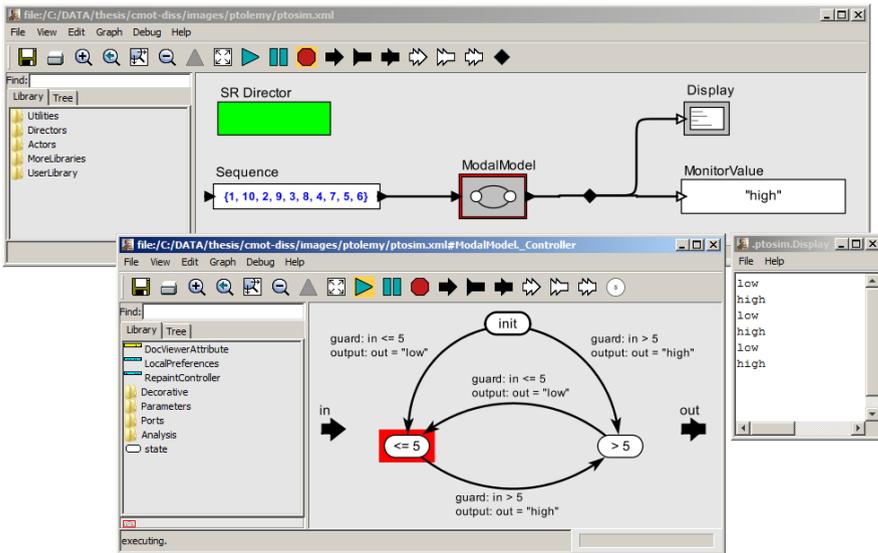


Figure 6.1.2. Simulating a Ptolemy model

Ptolemy Simulation

In order to also inspect dynamics of a model, Vergil provides integrated simulation features that are based on the fact that all Ptolemy models have a well defined dynamic behavior and are runnable. Semantical details will be sketched in Section 8.1.1 on page 381.

Figure 6.1.2 shows a Ptolemy model during simulation in Vergil. There are several generic input actors that act as data *sources*. In this example, a Sequence of integer tokens is used to generate inputs that stimulate a state machine ModalModel. This state machine can be opened and displayed in a separate window which is shown in the lower part of Figure 6.1.2. In addition to data sources, there are also data *sinks* which are actors that store data or present them to the user. In this example, a MonitorValue and a Display actor is used as a data sink.

During simulation, the MonitorValue can only show the current/last

6. SCCharts Tooling

(token) value of the link it is connected to. In contrast, the Display opens another window and displays each different token value in a new line. Such a window is shown in the right part of Figure 6.1.2.

Typically, the simulation is started by using the *play* button from the toolbar (of any window). Depending on the settings of the green SR Director actor, which defines the execution semantics, the simulation speed is often very fast and only final or traced behavior can be inspected, but no interaction is possible. The director can be used to generally slow down execution and the model can even be changed during execution, which becomes necessary if input actor values need to be changed dynamically. Another possibility to slow down simulation is to use visual debugging for the state machine as done in this example. A number of milliseconds can be defined to rest after a state change has occurred and the new state is visualized by a red highlighting.

In general, the computation with the SR Director, for instance, is stepwise. However, the GUI does not allow to execute just a single execution step. The user needs to press *play* and then quickly enough has to press *pause* to execute only one step.

Furthermore, if a model becomes larger and has many hierarchy levels and/or display actors, it might become hard to place all windows such that the dynamics of the model can be well understood and interesting changes in behavior can be noticed.

In the context of this work, our KIELER simulation and modeling GUI is largely inspired by Ptolemy (cf. Figure 6.3.1 on page 316). E. g., we also integrated a play, pause, and stop button in the KIELER simulation GUI. Also, highlighting of active states was added. Furthermore, we added a step button and additionally we enable the user to perform a defined number of steps until the simulation pauses automatically and can be resumed. We also allow for making steps backwards into history of already computed behavior/ticks. We further allow to input values dynamically (or automatically from other data sources) and do not need to modify the model during runtime. Typically, we try to display the model in one or just a few different views that incorporate hierarchy layers to allow an easier navigation and not overwhelm the user with numerous model windows.

Table 6.1.1 summarizes simulation and simulation GUI features of

Table 6.1.1. Ptolemy Vergil vs. KIELER Execution Manager (KIEM) simulation features

	Ptolemy Vergil	KIELER Execution Manager
Play/pause/stop buttons	+	+
Stepwise execution	+	+
Highlight active parts	+	+
Automatic layout	+	+
Dynamic inputs	+/-	+
Step button	-	+
Step back button (history)	-	+
Reduce number of windows	-	+
Modify model during simulation	+	-

Ptolemy Vergil that we adopted and extended in our KIELER simulation infrastructure and GUI. It is introduced in Section 6.3 on page 316. As Ptolemy models are plain Java programs that can be smoothly integrated into an Eclipse-based project, we have also integrated a generic model Simulator that is able to stepwise execute Ptolemy models directly from the KIELER GUI. Section 8.1.1 on page 381 will give insights about this simulator that we used mainly for simulating SyncCharts as a case study.

6.1.2 SCADE

The commercial tool SCADE has been introduced earlier (cf. Section 2.4.1 on page 30).

Integrated Simulator

In SCADE, an integrated simulator (cf. Figure 6.1.3) uses compiled C code to simulate the behavior of the SCADE model within the tool. Inputs can be set by the user and outputs can be inspected in a tabular view. SCADE also offers a TCP interface for communicating and controlling the simulation from outside the tool.

6. SCCharts Tooling

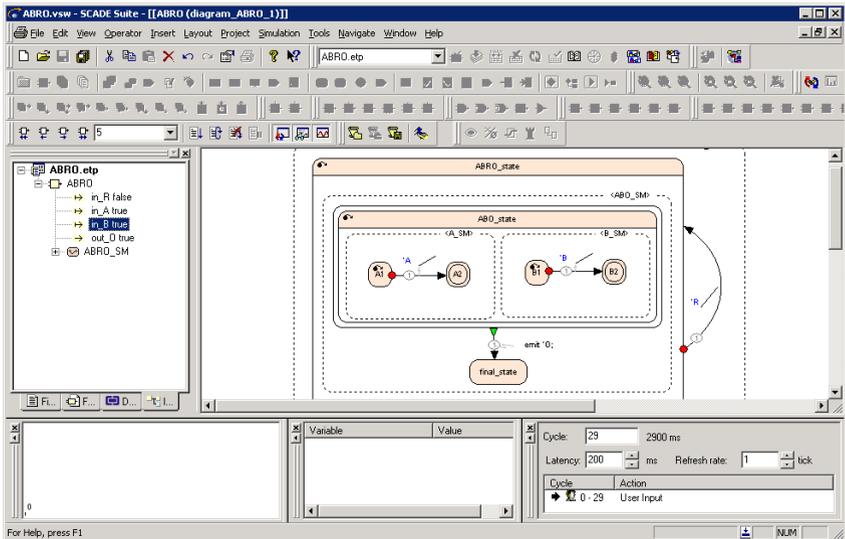


Figure 6.1.3. ABRO synchronous state machine within the SCADE suite and its model simulation GUI

In the context of this work, the KIELER Execution Manager (KIEM) execution framework offers a generic simulation infrastructure together with other GUI components for letting the user interact with simulated generic models. SCADA's stepwise simulation inspired much of the user interface part of KIEM. Where SCADA's simulation is not able to make steps backwards, this is made possible in KIEM. In addition, our work aims at integrating generic black box compilers as well as model transformation-based interactive white box compilers for model simulation purposes, while most of the GUI components can be re-used for arbitrary simulators.

6.1.3 Esterel Studio

Esterel Studio [Est04] (cf. Figure 6.1.4) was a tool to design control-flow models. It was mainly used to synthesize hardware, e.g., by exporting

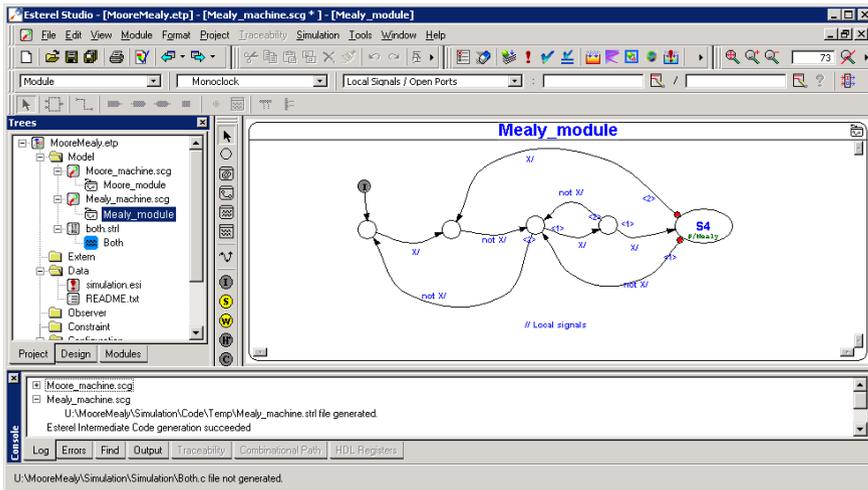


Figure 6.1.4. Esterel Studio GUI

models to VHDL code. Formal verification is well integrated into this tool.

Simulation is also a built-in feature. In early compiler versions (Esterel v4), a VHDL-based soft-prototype was generated out of a graphical control-flow model termed Safe State Machines (SSM), a Statechart dialect, and then simulated with a generic hardware simulation engine. SSMs can be seen as the commercial version of SyncCharts (see Section 2.3 on page 24), the predecessor of SCCharts. Simulation visualization of active states or taken transitions in Esterel Studio inspired the simulation visualization component of SyncCharts and SCCharts as part of this work. Furthermore, Esterel Studio was able to generate execution trace (ESO) files with a coverage of all reachable states and possibly taken transitions. These trace files are still used to validate the SCCharts compiler and can be manually recorded by the SCCharts tooling. Esterel Studio and its compiler built upon the Esterel language. In particular, all graphical SSMs are first compiled into Esterel code and then further compiled to SW or HW.

6. SCCharts Tooling

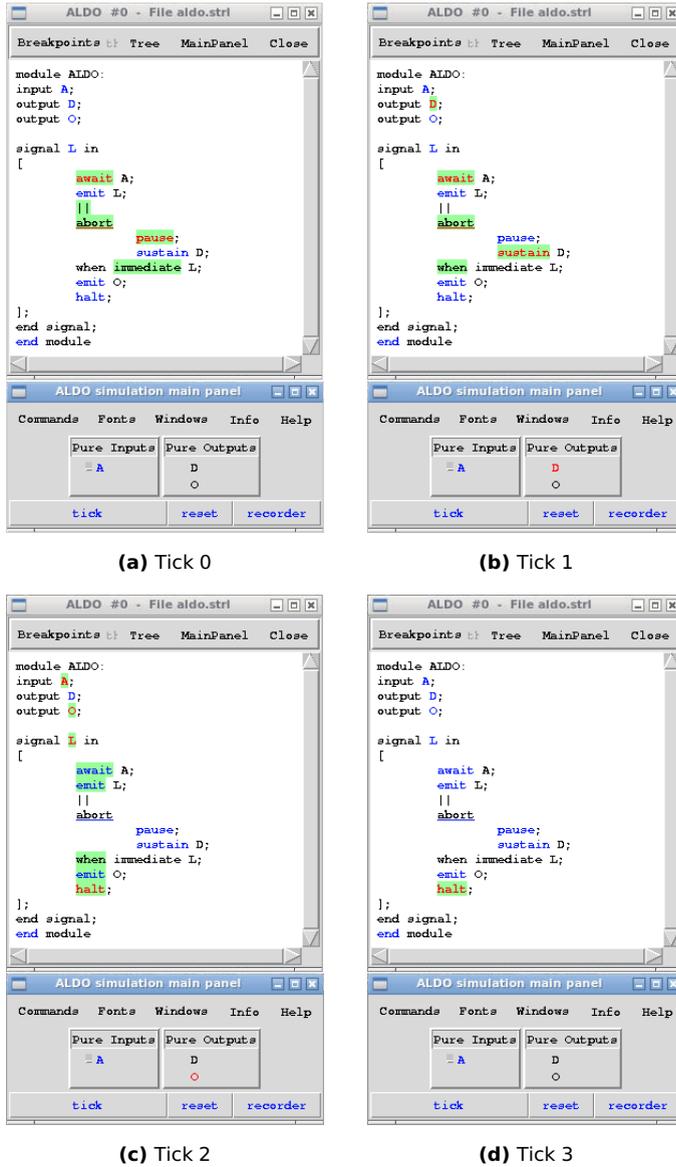


Figure 6.1.5. Esterel v5.92 simulator running ALDO

6.1.4 Berry Esterel v5.92 Compiler and Simulator

The Berry Esterel v5.92³ compiler was developed at the Inria institute in France. It supports the Esterel v5 standard [Ber00a].

Figure 6.1.5 shows the ALDO example running in the GUI of the Esterel v5.92 simulator for an example trace of 4 ticks. The GUI allows to set the presence status of input signals (lower part) and inspect the presence status of output signals. The graphical editor is directly used to visualize active statements during the simulation. Statements that were executed in the current tick have a green background and statements, where control rests in the current tick and control will start in the next tick, are red-colored.

Tick 0 of Figure 6.1.5 is the initial tick where no input was set. The concurrent statement is entered. For Tick 1, control will therefore start with the depth of `await A` of the first thread and the depth of pause in the second thread. In Tick 1, also the input signal `A` was not set to be present. Hence, the `sustain` action in the second thread is executed emitting `D`, and the `await` statement in the first thread is still awaiting `A` to become present. In Tick 2, the input signal `A` is set to be present. Hence, the `await(A)` statement terminates and the first thread emits the local signal `L`. The second thread instantaneously reacts to `L` by strongly aborting the `sustain` and emitting output signal `O`.

The combination of the I/O view at the bottom and the colored highlighting in the editor gives a complete understanding of the I/O and also the internal status of the Esterel program. We adapted major parts of the simulation GUI from the Esterel v5 simulator for the KIELER Esterel simulator. This includes highlighting of active statements, the tick and reset button, and a visualization of present/absent signals.

A drawback is the tight integration of the simulation visualization and the Esterel compiler. We came up with more generic visualization that can be used for any black box Esterel compiler. Note that the approach is rather generic and can be used for other compilers as well. As an example, we also integrated two Esterel compilers with the same simulation visualization: The Columbia Esterel Compiler (CEC) and our own SCEst compiler. The CEC and the generic simulation visualization are discussed

³http://www-sop.inria.fr/esterel-org/files/v5_92

6. SCCharts Tooling

in the following paragraphs. The SCEst compiler is sketched in Section 8.2.4 on page 403. Another drawback is that only a snapshot view of the current tick's signal statuses is accessible. This motivated the development of the KIELER *Synchronous Signals* view (cf. Section 6.3.3 on page 320). Yet another drawback is that one cannot make steps backwards to replay past behavior.

6.1.5 The Columbia Esterel Compiler (CEC)

The open-source Columbia Esterel Compiler⁴ [CEC, EZ07] is developed at the Columbia University of New York. The CEC is capable to produce SW and HW from Esterel code and it supports a subset of the Esterel v5 standard [Ber00a]. E. g., the CEC does not support the Esterel `pre()` statement, which allows to access the instance of a signal from the previous tick.

The CEC is written in the C++ language and can be compiled for different operating systems. This has been done in order to integrate version 0.4 of the CEC with the Java-based and Eclipse-based KIELER platform. The CEC is currently used as the default compiler for Esterel programs in KIELER. The CEC is also used as a reference compiler to generate ESO files as described in Section 6.6 on page 353.

Figure 8.2.5 on page 399 shows a simulation of an Esterel program in the KIELER tool using the CEC. This simulation is connected to a visualization component that uses an Esterel editor to highlight active Esterel statements, similar to the Esterel v5.92 simulator (cf. Figure 6.1.5). The used editor is generated from an Esterel grammar and uses an Esterel meta model as discussed in Section 8.2 on page 393.

6.2 Eclipse

The implementation of the KIELER simulation infrastructure as well as the prototype for interactive incremental model-based SLIC compilation and the modeling of SCCharts is part of the KIELER⁵ project. This builds upon

⁴<http://www.cs.columbia.edu/~sedwards/cec>

⁵<http://rtsys.informatik.uni-kiel.de/kieler>

the Eclipse⁶ framework. Eclipse is a platform, well known as *the* Java IDE. It itself is implemented in the Java language but by now has evolved to be a development tool for various other languages, e. g., C++, PHP, XML, or Python. It can also be seen as a framework for building IDEs. This is summarized by the common principle that Eclipse is “an IDE for anything, and yet nothing in particular” [dRB06]. Figure 6.2.1 shows an Eclipse IDE with two opened Java Eclipse editors on the right side and three Eclipse views.

Eclipse Workbench: This is the minimal set of basic plug-in components. It can be seen roughly as the main window of an Eclipse application. It integrates Eclipse editors and Eclipse views as workbench parts.

Eclipse Editors: These are integrated into the workbench, meaning that all contributions, such as toolbar icons, are embedded into the overall workbench when a specific editor is active, i. e., has the focus.

Eclipse Views: These typically present additional or different information about the contents of the currently active editor or even a selected object within the editor. Views are not as integrated into the workbench as Editors are. Typically, there exists at most one instance of a view. Views usually have their own toolbar but can also contribute to the workbench’s toolbar and main menu.

6.2.1 Eclipse Plugins

The building blocks of Eclipse are components called *plugins*. The basic Eclipse platform consists of a small number of such plugins and is extended by other plugins. This architecture is flexible, modular, and extendable at the same time: New plugins can be easily built upon existing plugins or new plugins can replace existing ones.

Figure 6.2.2 shows the main idea of the fundamental extension point mechanism. The mechanism provides extendability by defining extension

⁶<http://www.eclipse.org>

6. SCCharts Tooling

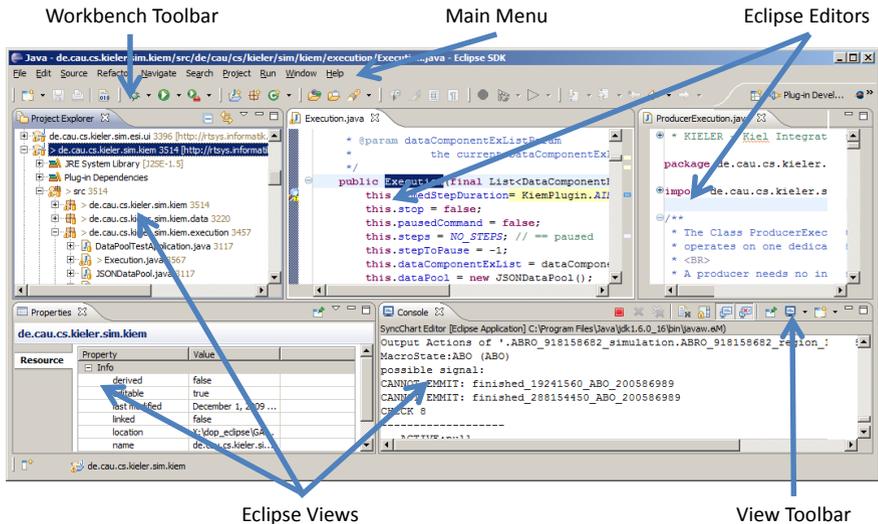


Figure 6.2.1. Eclipse IDE with editors and views [Mot09]

points in a Base Plugin that can be used by some — possibly unknown — other Extending Plugin that may contribute functionality using this extension point. It is noteworthy that the Base Plugin, at runtime, uses code from the Extending Plugin without knowing any details or having any dependency to it. However, the Extending Plugin must have a so called *plugin dependency* to the Base Plugin in order to extend the extension point. An extension point is defined by the Base Plugin often providing Interfaces and/or Abstract Classes. This information is later used to access and execute code from extending plugins in the Base Plugin. The Extending Plugin must have a dependency to be able to implement or extend the plugin extension point interfaces and/or abstract classes. The Eclipse infrastructure subsequently allows the Basic Plugin to instantiate classes of the Extending Plugin that implement/extend extension point interfaces/abstract classes. Without knowing any details about the Extending Plugins or even about the existence of concrete Extending Plugins. The Base Plugin can iterate over all extending plugins and instantiate the classes. The access is possible because the interfaces/abstract classes

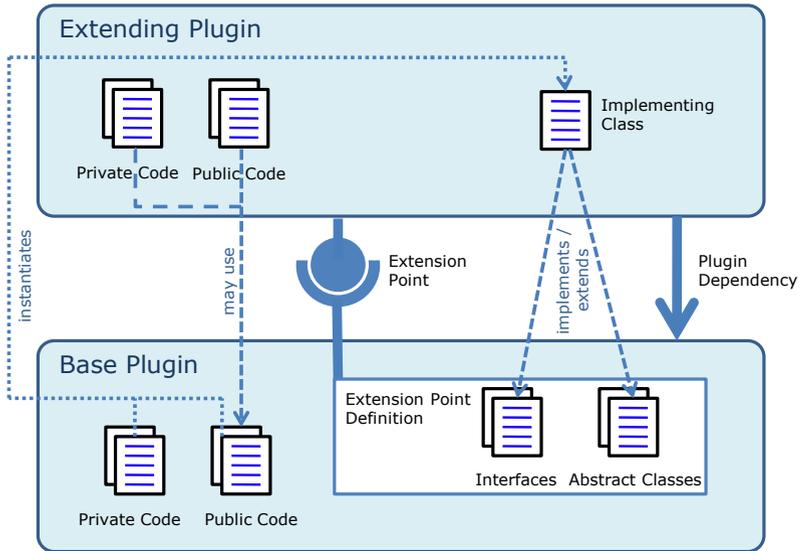


Figure 6.2.2. Eclipse extension point mechanism

are defined in the Base Plugin. Additional methods, other than defined in the interfaces/abstract classes, cannot be accessed by the Base Plugin.

However, since there is a plugin dependency from the Extending Plugin to the Base Plugin, it is possible for the Extending Plugin to use the public API of the Base Plugin. The extension point definition and the extension definition is part of the `plugin.xml`. This is a special configuration file for a plugin where such information is stored. The plugin dependencies are configured in `MANIFEST.MF`, another configuration file. The `plugin.xml` and the `MANIFEST.MF` file will be analyzed by the Eclipse runtime kernel and also at compile-time, e. g., to make public APIs of other (base) plugins visible to a depending/extending plugin.

6. SCCharts Tooling

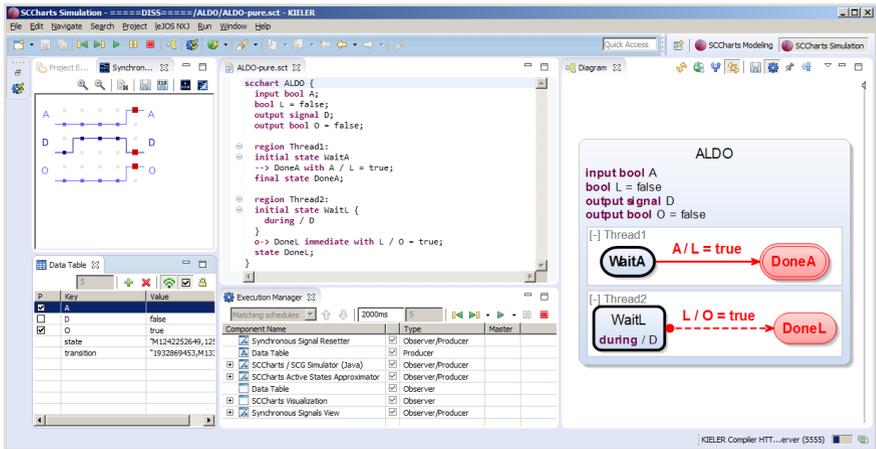


Figure 6.3.1. GUI of the KIELER simulation infrastructure while simulating the ALDO SCCharts model

6.3 The KIELER Simulation Infrastructure

This section gives an introductory overview of the KIELER simulation infrastructure that is part of the KIELER SCCharts tooling. It mainly consists of the *KIELER Execution Manager* (KIEM) which provides a generic infrastructure for integrating model simulators/compiler, a UI for KIEM that enables user control over simulations, and data input/output components such as a *Data Table* and a *Synchronous Signals* view. Figure 6.3.1 shows this Eclipse-based infrastructure and its components while simulating the ALDO example introduced earlier. These individual components are discussed in the following sections.

6.3.1 KIELER Execution Manager (KIEM)

The *KIELER Execution Manager* (KIEM) [MFvHL12, MFvH10] is the central and unifying component for any simulation in KIELER.

KIEM consists of a GUI part and a non-GUI part. The GUI part is

6.3. The KIELER Simulation Infrastructure

shown in the central lower part of Figure 6.3.1. It offers predefined lists of *execution schedules* that can be defined per model editor. These schedules consist of instances of *DataComponents* that can interact with each other or with the user. For example, the SCCharts/SCG Simulator is a *DataComponent* instance that can be stimulated by user input. It outputs a reaction for the currently simulated model. Each *DataComponent* instance may have additional properties (see Key/Value properties in the Execution Manager view of Figure 6.3.4) that can be predefined but also be modified by the user. A common property is the currently active model which often serves as an input in the initialization phase. For each (synchronous) execution step, the *DataComponents* are executed in the sequential order of their schedule. A step can be triggered by the appropriate GUI button or it can be executed automatically after the user-defined step-time (here 2000ms) when the user has pressed on the play button. A running execution can be paused/resumed and stopped. Additionally, the user may want to make steps backwards into the history of already computed values.

KIEM's non-GUI part is the management, control, and scheduling of

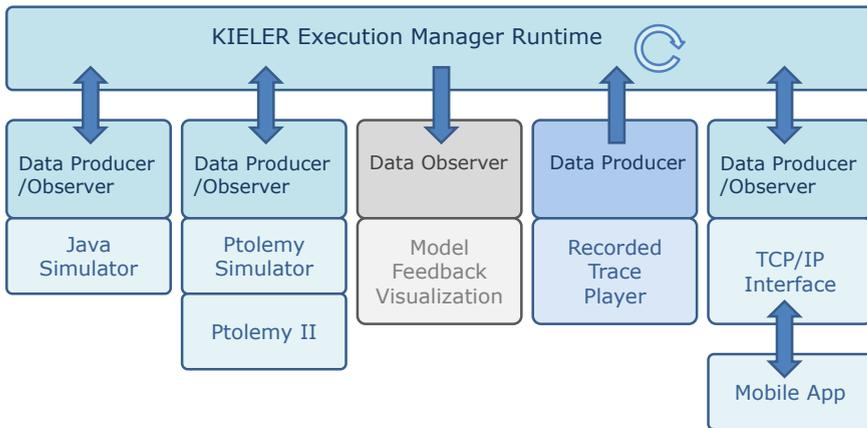


Figure 6.3.2. Schematic overview of the KIELER Execution Manager (KIEM) infrastructure with various kinds of interacting *DataComponents* (adapted from [MFvHL12])

6. SCCharts Tooling

the DataComponents, their instances, but also execution schedules and an organized dynamic data pool under the hood. The conceptual schema is shown in Figure 6.3.2. All DataComponents have different roles which are central for the scheduling. A Producer DataComponent possibly produces data but not necessarily according to the current step/tick. An example is a Recorded Trace Player which re-produces signals which may have been previously recorded or have been imported from other tools. Producer components do not depend on any inputs. An Observer DataComponent observes data coming from other components but does not produce any data/reaction. An example is a Model Feedback Visualization which could take the reaction data of a simulator component to highlight active model elements in a model editor. There are also components that have both roles, i. e., they observe and also produce data. A typical example are simulation components such as a Java Simulator or a generic Ptolemy Simulator but also proxy DataComponents like a TCP/IP Interface for remotely connected components. KIEM puts all produced data into a special dynamic data pool. Data are JavaScript Object Notation (JSON) key-value pairs. Producing data means introducing or updating data for a key. Observing data means seeing all possibly updated values of all keys for which data that has been produced in an execution. The data pool also allows for easily making steps into the history. More details about KIEM are published elsewhere [MFvHL12, MFvH10, Mot09].

The following sections will cover some generic DataComponents that are used for simulating SCCharts but also other languages.

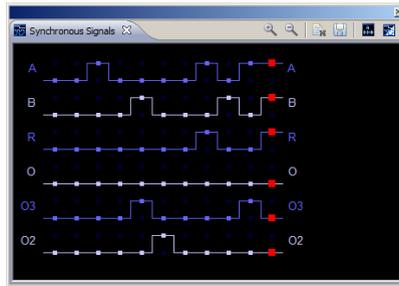
6.3.2 Data Table

The *Data Table* is shown in the lower left part of Figure 6.3.1. According to the signature of the current model, it lists all inputs and outputs and allows the user to set inputs for the next synchronous tick/step and/or to inspect the outputs of the current synchronous tick/step. Besides the name, a value can be set or inspected and a presence status of a signal can be modified/inspected by the appropriate checkbox. For signals, a checked checkbox represents the presence and an unchecked checkbox represents the absence of a signal. For boolean variables, a checked checkbox represents a

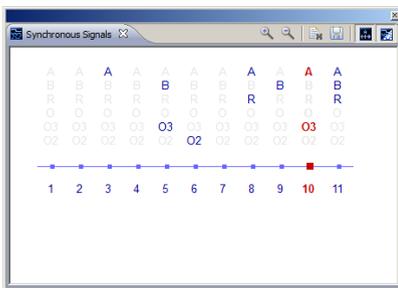
6.3. The KIELER Simulation Infrastructure

P	Key	Value
<input checked="" type="checkbox"/>	A	
<input type="checkbox"/>	B	
<input type="checkbox"/>	O	
<input type="checkbox"/>	O2	
<input checked="" type="checkbox"/>	O3	
<input checked="" type="checkbox"/>	statement	M114636917,M114...

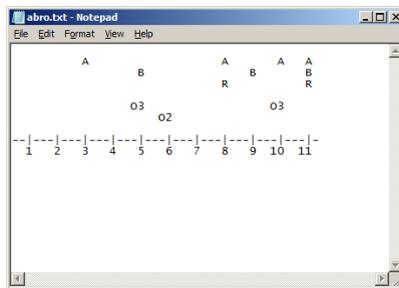
(a) Data Table: One dimensional view of current tick signal status



(b) Signals view: Two dimensional view of signal status over time (ticks)



(c) Signals view's additional viewing for modes and contrast options



(d) Export traces to ASCII or ESO file for documentation and validation

Figure 6.3.3. GUI for input/output and debugging: KIELER Data Table and Synchronous Signals view

true and an unchecked checkbox represents a false value of a variable.

The Data Table DataComponent is a very generic component that allows the user to access and manipulate data exchanged by other DataComponents. It is used in various settings and for various simulation components. The Data Table acts very similarly to the input/output facility of the Esterel v5.92 simulator that is shown in Figure 6.1.5 on page 310. Nevertheless, it is designed to be re-usable for all kind of other simulators that are integrated into KIEM.

6. SCCharts Tooling

6.3.3 Synchronous Signals View

One drawback of the Data Table is that it only displays the signal presence statuses and variable values of the current tick. The user is unable to visually see the actual trace of such signal statuses or variable values. However, this may strongly be desired for debugging purposes, e. g., to compare signal traces to each other and see whether two simulators or different versions of one and the same simulator behave the same for a specific model and a specific input trace. This is also a drawback in the GUI of the Esterel v5.92 simulator shown in Figure 6.1.5 on page 310.

To overcome this situation, the *Synchronous Signals* view (cf. Figure 6.3.3) was developed. It enables the user to see the signal presence status change or the variable value change over time in a two dimensional diagram, not a one dimensional presentation as in the Data Table.

This view comes with alternative view modes such as white or black background and view modes that show the names or values of currently present signals. Boolean variables are represented as pure signals by this view. An export mechanism allows to create ASCII text for documentation purposes or ESO trace files for (later) validation (see Section 6.6.3 on page 358).

6.3.4 Benchmark Component

It may be desirable to test and compare a compiler for certain properties such as the compile-time, the size of the fully expanded model, the size of the compiled executable, or even the reaction tick time of the running executed model. This may help the compiler developer to optimize transformations or the tool user/the modeler to find a more optimal way to express desired functionality in a model. It may also help comparing one compilation to another one for the same model in order to identify general drawbacks or advantages of one or the other. Benchmarking combined with automated nightly regression tests help maintaining stability for certain compiler properties such as the ones mentioned above.

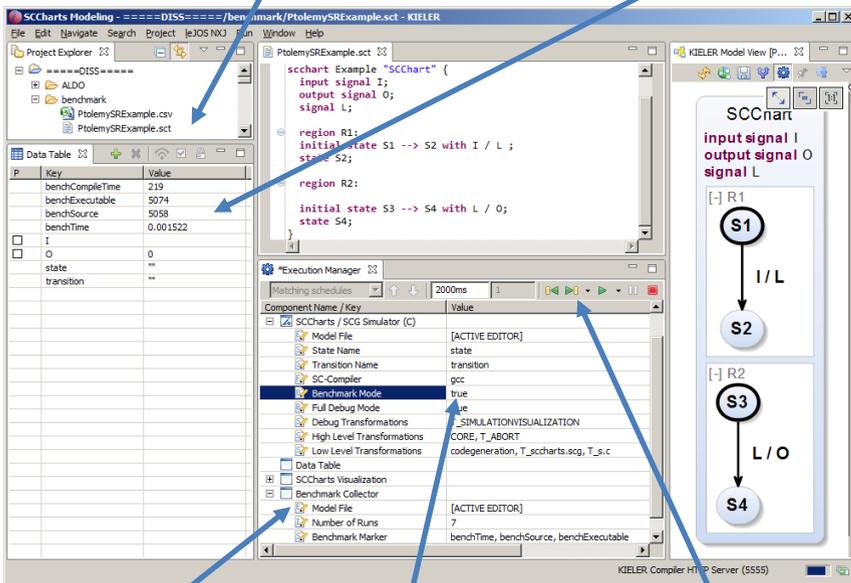
KIEM provides a benchmark interface that simulation DataComponents can implement to express their support for a number of measurable com-

6.3. The KIELER Simulation Infrastructure

piler properties. For instance, the SCCharts simulator implements this interface. The KIEM properties of the SCCharts simulator DataComponent are expanded and visible in Figure 6.3.4. 1. The benchmarking must be turned on explicitly by using these properties or a predefined KIEM schedule can be chosen where the benchmarking is already turned on. 2. The simulation can then be done interactively by using, e. g., the Data Table for inputs or by using an ESO trace file player DataComponent to automatically

5. CSV file automatically created by benchmark collector when simulation finishes

3. Inspect live benchmark values



4. Use benchmark collector to gather values

1. Turn on benchmarking (KIEM Property)

2. Execute simulation manually or with pre defined ESO trace

Figure 6.3.4. Benchmarking user story exemplified by SCCharts simulation

6. SCCharts Tooling

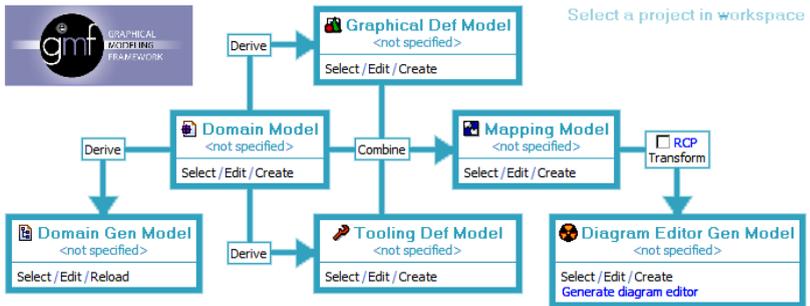


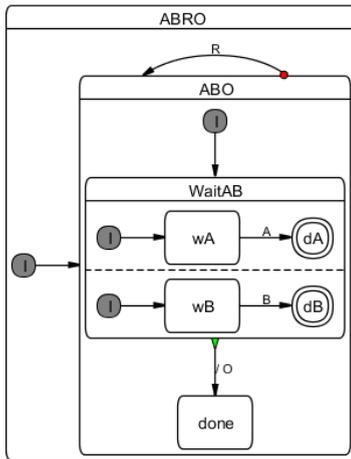
Figure 6.4.1. GMF dashboard for model-based graphical editor generation in Eclipse (adapted from GMF website, see below)

feed the simulator with inputs. 3. The benchmark values are additional outputs of the simulator and hence may be inspected in the Data Table. 4. They can be gathered by an additional Benchmark Collector DataComponent. 5. When the simulation finishes because the ESO file ends or the user clicks on the KIEM stop button then the Benchmark Collector will output its collected values in a CSV file which can be used, e.g., with other tools to analyze and evaluate the numbers.

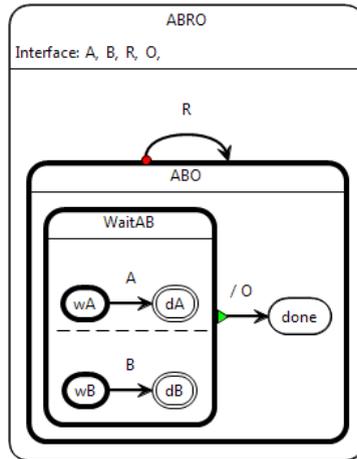
6.4 SCCharts Editor Implementation

The KIELER SCCharts modeling environment is shown in Figure 3.0.1 on page 47. It heavily uses Eclipse infrastructure. The basis for the KIELER SCCharts implementation is the Eclipse Modeling Framework (EMF) [SBPM09] for the abstract syntax of SCCharts (cf. (3) in Figure 3.0.1 on page 47).

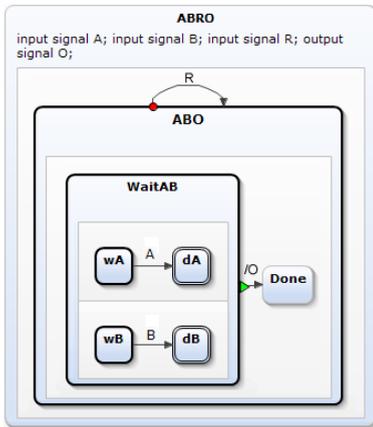
6.4. SCCharts Editor Implementation



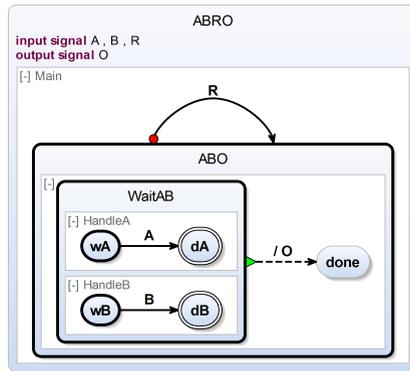
(a) KIEL tool (SyncCharts)



(b) ThinkKCharts editor (SyncCharts)



(c) Yakindu SCCharts editor



(d) Xtext and KLightD-based SCCharts editor (current)

Figure 6.4.2. The KIELER SyncCharts/SCCharts editor evolved over time. Here, the ABRO example is shown in the different editor implementations.

6. SCCharts Tooling

6.4.1 KIELER ThinkCharts Editor

The current KIELER editor for SCCharts has its roots in an early SyncCharts Eclipse implementation [Sch09a] called ThinkCharts (see Figure 6.4.2b). This implementation was based on GMF⁷. It followed the model-based approach in specifying higher-level models with information about the graphical editor, its graphical entities appearance, its palette, and editing options. The fully-featured graphical “What You See Is What You Get” (WYSIWYG) editor was generated using the GMF tooling Eclipse project (see Figure 6.4.1).

The Eclipse-based ThinkCharts editor itself evolved from an originally monolithic, pure Java tool called KIEL [PvH07] (see Figure 6.4.2a).

Although an early version of the ThinkCharts editor implementation was already usable after a short development time, the fully-featured editor needed much tweaking which turned out to consume an unexpected high amount of development time and effort. To be able to later change specification models, the generated artifacts were not touched and only GMF templates were adjusted to achieve the desired modeling editor.

As it turned out, this created rather unexpected problems elsewhere. Whenever the version of GMF was updated, this resulted in unpredictable changes to the (old) GMF templates or in even completely new GMF templates. However, the old templates had been modified for the specific SyncCharts editor generation. Re-using the old templates from an earlier GMF version was not possible and it took much effort to reproduce the modifications of the old templates and to apply them to the new templates.

6.4.2 The Yakindu SCCharts Editor

After the moderate GMF experiences, a new WYSIWYG editor prototype for SCCharts [Har13] was implemented based on the Yakindu statecharts tools⁸ (see Figure 6.4.2c). The hope was to reduce the effort and struggling with editor development. The Yakindu statecharts tools are designed to start with a common statechart editor that is part of the Yakindu framework.

⁷<http://www.eclipse.org/modeling/gmp>

⁸<http://statecharts.org/documentation.html>

6.4. SCCharts Editor Implementation

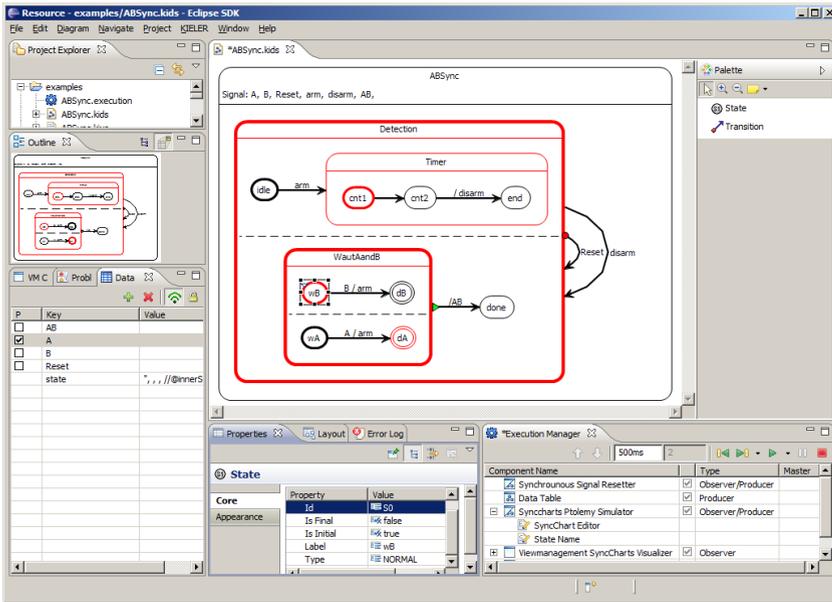


Figure 6.4.3. KIELER ThinkCharts editor for modeling and simulating SyncCharts in Eclipse (from [MFvH10])

Using an API, a concrete statechart editor can be derived for the specific needs, inferred from the concrete syntax and semantics of a statechart language. More information about Yakindu can be found in the Yakindu documentation.

The Yakindu SCCharts editor project was realized in close cooperation with the itemis company that develops the Yakindu statechart tools. During this project much improvement could be realized for the Yakindu statechart tools w. r. t. extensibility. Although most of the SCCharts functionality could be implemented in this prototype with great support of itemis, the project still revealed some limitations when it came to heavier meta model design requirements. It was further designed to build upon an existing expression language.

With a new version of the Yakindu statechart tools, the prototype imple-

6. SCCharts Tooling

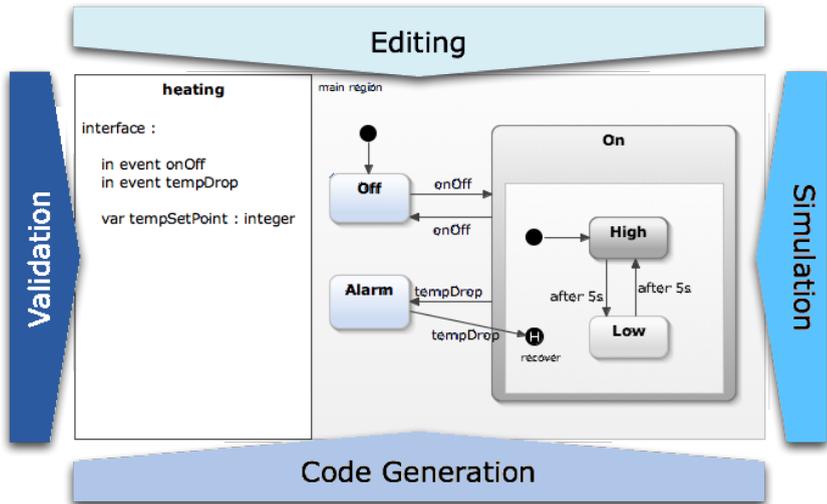


Figure 6.4.4. Yakindu statecharts tools overview showing the common statechart editor that can be derived to be specialized (adapted from the Yakindu website, see footnote)

mentation revealed similar problems as the GMF implementation. Due to a larger number of API changes, it seemed disproportionately difficult to update the prototype to the newer version of Yakindu without appropriate man-power. Thus, the prototype was used for a while but never released together with KIELER.

6.4.3 The SCCharts Textual Editor

Besides the Yakindu-based SCCharts prototype, a textual editor for SCCharts was developed that was based on the KIELER Textual SyncCharts (KITS) language [Sch11] and implementation (see Figure 6.4.2d). Technically, this textual SCCharts editor was based on the Xtext⁹ framework. Eclipse Xtext editors are also backed by the Eclipse modeling framework EMF.

⁹<http://www.xtext.org>

6.4. SCCharts Editor Implementation

The development of another project called *KIELER Lightweight Diagrams (KLighD)* [SSvH13], which aimed to automatically visualize arbitrary EMF models, lead to the decision to use the Xtext-based textual model editor together with KLighD for modeling and simulation purposes and to move away from WYSIWYG editors in general. The KLighD project is using a *diagram synthesis* that preferably is specified in the Xtend¹⁰ language to visualize arbitrary EMF models.

Figure 6.4.5 visualizes the current KIELER SCCharts modeling implementation and w. r. t. Figure 3.0.1 on page 47, it shows further details and used technologies. The SCCharts Xtext editor allows to view and edit an abstract SCChart EMF model using the textual SCT language. The Xtext framework automatically synchronizes the changes with an abstract EMF SCCharts model that conforms to the EMF meta model shown in Figure 3.4.1 on page 82.

From this EMF representation of a concrete modeled SCChart, the KLighD framework uses model transformations to render a graphical diagram automatically. These model transformations are usually described using the Xtend language in a so called SCCharts *diagram synthesis*. The same SCCharts EMF representation is used when (interactively) compiling an SCChart to intermediate models or executable code using the KIELER Compiler.

6.4.4 Graphical vs. Textual vs. *Textical* Modeling

Table 6.4.1 summarizes the advantages and disadvantages of pure graphical and textual modeling. Additionally, the combined textual modeling with a graphical view, called *textical modeling* here, is compared. Textical modeling aims at combining the best of both worlds. The advantages of textical modeling were the main motivation for the current SCCharts modeling tool implementation. Details were presented elsewhere [RSS⁺13].

Learning curve: Graphical modeling typically has a short learning curve as developers often already are familiar with the concepts of a palette, a canvas, or context menus for editing diagram entities. A graphical

¹⁰<http://www.xtend-lang.org>

6. SCCharts Tooling

Table 6.4.1. Graphical vs. Textual vs. Combined *Textical* Modeling. SCCharts uses a combined textual and graphical modeling with an automated graphical view. (adapted from [RSS⁺13])

	Graphical Modeling <small>(WYSIWYG)</small>	Textual Modeling <small>(Text Editor)</small>	<i>Textical</i> Modeling <small>(Text Editor + Graphical View)</small>
Learning curve	+	-	+/-
Readability	+/-	+/-	++
Visualizations	+	-	++
Validation	+	-	++
Maintainability	+/-	+/-	+
Focus and context	-	-	+
Rapid development	-	+	+
Model handling	-	+	+

editor is often self-explaining to use. For textual modeling this is often not the case. Although often only low-level editing operations such as inserting lines, copy and paste, or context assists are common for most textual IDEs, every modeling language typically has a very different syntax and meaning of symbols such as brackets or spacing. Building valid and desired graphical models is often easier because the modeler is restricted to editing operations that more or less guide her or him towards a valid model.

For textical modeling, the main drawback of a new textual syntax cannot be allayed but an automated graphical view is very helpful for learning even a new textual modeling syntax. It can reveal much undesired issues and hence also help in guiding the modeler towards valid models.

Readability: On the one hand graphical models are often easier to understand when inspecting them. Operations like zooming and panning or overview panels aid the modeler and facilitate readability. On the other hand graphical models that get larger and have a manual layout often experience readability problems if the modeler does not invest much effort for clean-up, reducing white space, and moving around or align-

6.4. SCCharts Editor Implementation

ing entities. Textual models are often equipped with a formatter that makes smaller parts of code readable immediately. However, especially larger textual models often are harder to read due to a lag of overview abstraction.

As textual modeling combines the textual editor with a graphical view it also benefits from a textual formatter that makes smaller chunks of code readable. The automatically synthesized graphical diagram does not need manual clean-up, reducing white space, or aligning. It still conserves the graphical benefits such as zooming and panning to give

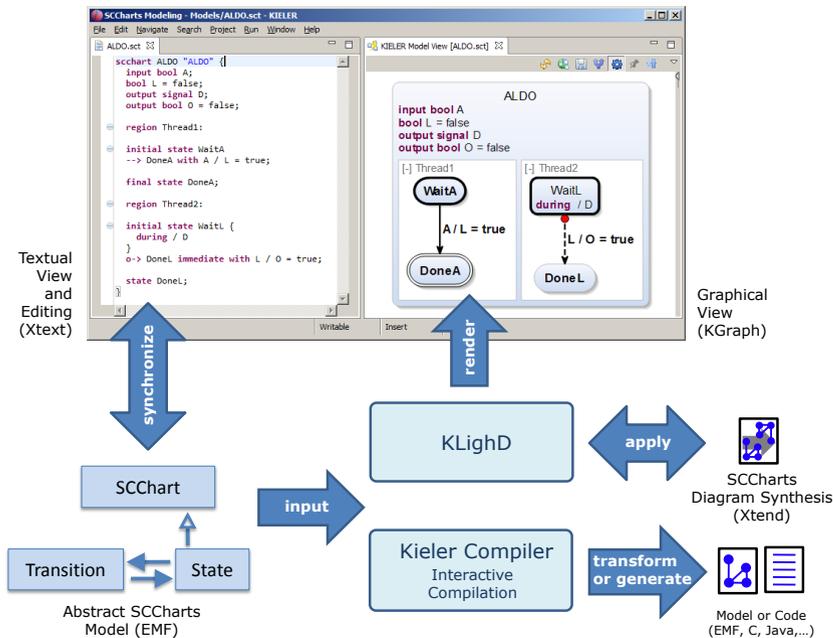


Figure 6.4.5. KIELER SCCharts modeling tool implementation overview: SCCharts textual modeling with Xtext editor and automatic synthesized KLightD diagram based on model transformations of the abstract EMF model. The interactive compilation is also based on the abstract model.

6. SCCharts Tooling

the modeler a good overview or use the graphical diagram for navigation in the textual model representation.

Visualizations: Graphical models are well-suited for visualizing dynamics, e. g., of a running simulation. For SCCharts, this could be marking active states with a certain color. However, also other information like scheduling numbers or timing information can be easily brought to the users attention in a graphical model. Textual editors always struggle when it comes to well-suited visualizations because typically only a small fraction of the whole model is visible. Additionally, often one is limited to a certain coloring and to put side markers to the text editor.

Textual modeling can use the appropriate textual or graphical view to a model for visualizing any kind of feedback for the modeler. It is even possible to have a separate view, which filters other information to focus on the subject in question.

Validation: Validating models is often a split up task: On the one hand automatic validation rules are applied to a model and in case of failure, error, or warning markers are applied to the model. This is a special use of visualizations as discussed before. On the other hand the modeler often performs a manual validation by visual inspection of their model. For graphical models it is far easier to detect, e. g., unconnected nodes or asymmetries that should not exist. This is significantly harder or even impossible to realize in a textual editor.

Textual modeling allows both, a good automatic and a good manual validation of models as the graphical view can be utilized for this purpose or even a different and specialized view can be generated.

Maintainability: The maintainability of graphical models is often a harder task as readability aspects mainly bias maintainability. Although graphical models are often easier to understand, editing an existing graphical model and extending it, quickly leads to struggling with insufficient space for new parts, moving around existing parts, and/or re-aligning model entities. As mentioned before, textual editors often come with integrated formatters that support efficient editing and extending a textual model.

Textual modeling comes with the comfort of a textual model editor and

6.4. SCCharts Editor Implementation

an integrated formatter, but at the same time it facilitates understanding the model or parts of a model that need modifications or extensions.

Focus and context: Most graphical modeling tools let the user edit and inspect the model only for one hierarchy level at a time. Often, each hierarchy level is edited in its own window. Textual editors, at best, allow to collapse certain logically related blocks of text.

Automated graphical views that show exactly the necessary context are not limited by hierarchy levels. They focus on the parts of interest depending on the subject to visualize and are more flexible and adequate than pure graphical or pure textual editors.

Rapid development: The development time is often a key aspect of a modeling tool. Where graphical models with a manual layout often need much effort, as explained earlier, textual models can typically be developed much faster. For example, copy and paste in a textual editor or multi-editing are very helpful features where graphical model editors regularly get into consistency problems of, e. g., unconnected transitions or labels when using copy and paste there.

Textual modeling allows a rapid editing like textual editors and still visualize all elements that are possible to visualize. In case of, e. g., unconnected transitions, warnings or colors in automatically synthesized views can bring inconsistencies to the user's attention.

Model handling: Graphical models typically are persisted in a binary or XML format. When it comes to tool interchangeability, graphical models often expose significant problems. Additionally, using version control systems is typically much more tedious if binary or XML data are used. Incremental changes are harder to detect, automated merges often are impossible, and manual merges often lead to frustration. Textual models are often much easier to interchange between different modeling tools and version control is simplified.

Textual modeling uses a textual concrete model syntax for persisting models. Hence, it benefits from the same advantages as textual modeling.

6.5 Interactive SLIC Compiler Implementation

The KIELER Compiler (KiCo) is a prototype for interactive incremental model-based SLIC compilation and modeling. The KIELER SCCharts tool consists of a textual SCT editor that is based on the Xtext framework. This framework provides a fully-featured textual editor, a parser and serializer, and it is based itself on EMF. EMF is internally used to represent models in Eclipse. Model transformations for KiCo are usually implemented using the Xtend language. This is a statically typed programming language with functional additions based on Java. Xtend classes fully compile to Java classes. Hence, Xtend fully integrates into any Java project in Eclipse. Model transformations can also be written in Java directly. KiCo itself heavily uses the Eclipse concept of plugins as explained in the following.

6.5.1 KiCo Implementation

The KIELER Compiler was implemented to validate the SLIC concept. The implementation utilizes the Eclipse plugin concept (see Section 6.2.1 on page 313). The runtime of KiCo serves as a base plugin, where other plugins may contribute compilation M2M transformations using an extension point defined by KiCo (cf. Figure 6.2.2 on page 315). This way, other plugins may register a transformation and make it known to KiCo. The extension point is part of KiCo and defines an interface that specifies a transform method signature. That is, a transformation takes a model and returns a possibly modified or completely new model. Clearly, all plugins contributing transformations to KiCo need a dependency on KiCo while KiCo does not need a dependency on these contributing plugins thanks to the Eclipse extension point mechanism.

After several plugins contributed transformations to KiCo, some other plugin may *use* KiCo in order to compile a model. Clearly, a transformation using KiCo does need a dependency on KiCo. However, a plugin that uses KiCo does not need to have any dependencies on other plugins contributing transformations, although during a compilation, code from these plugins will be run by KiCo.

During this work, there were two prototypes developed that both elabo-

6.5. Interactive SLIC Compiler Implementation

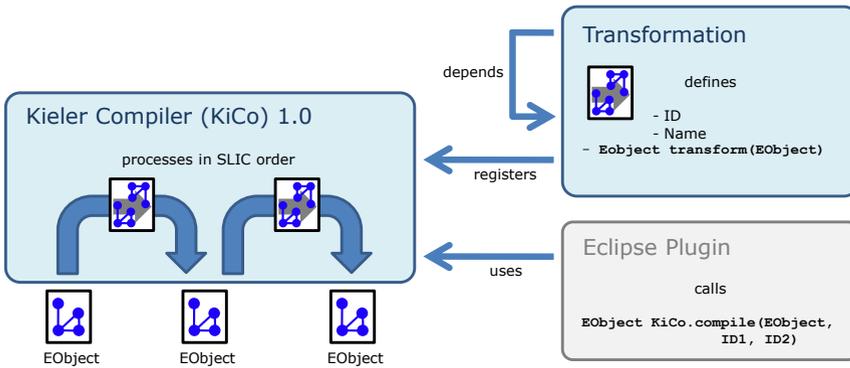


Figure 6.5.1. KiCo 1.0 prototype implementation overview

rate the SLIC concept using the Eclipse plugin mechanism. These prototypes are presented in the following.

6.5.2 KiCo 1.0 Prototype

The first prototype basically had only one extension point for registering transformations. It further did not distinguish between *produces* or *not-handled-by* dependencies as discussed in Section 4.1.1 on page 91. Figure 6.5.1 gives an overview of the design of the KiCo 1.0 prototype.

Transformation Plugin: The Transformation plugin is used to implement and configure concrete SLIC transformations (cf. Section 4.1 on page 89). The plugin defines a unique ID, a human readable Name, and a `transform()` method. The `transform()` method is the central part of a transformation plugin. It takes a model of type `EObject` as an input. Then it modifies the model, e. g., it expands a certain feature that this transformation is dedicated to. Finally, it returns the modified model of type `EObject`. The ID, the name, and the transformation method are part of the interface provided by KiCo.

Each transformation τ provides a list of *depending* transformation IDs. The ID of transformation τ' may be in the depending transformation ID list

6. SCCharts Tooling

of τ . This means that for the SLIC order, τ must be executed before τ' , i. e., $\tau \rightarrow \tau'$.

A transformation τ may produce features that are expanded by other transformations τ' . Since of $\tau \rightarrow_p \tau'$, it must be ensured that τ is executed before τ' . The \rightarrow_p order is implemented by putting the ID of τ' in the depending transformation ID list of τ .

A transformation τ' may not be able to handle a certain feature expanded by τ . Since of $\tau \rightarrow_{nhb} \tau'$, it must be ensured that τ is executed before τ' . The \rightarrow_{nhb} order is implemented by putting the ID of τ' in the depending transformation ID list of τ . The depending ID list is also part of the KiCo interface.

KiCo Plugin: The KiCo plugin Kieler Compiler (KiCo) 1.0 provides the extension point for registering transformations. At startup, it collects all transformations from plugins contributing transformations by utilizing the extension point mechanism. More concretely, for every transformation it collects the ID, the name, the transformation method, and the depending ID list (see above). Using this information and the SLIC order, a static SLIC schedule is derived that can be used later for a concrete compilation.

Other Plugins: Some other Eclipse Plugin which has a plugin dependency on KiCo may use the central `compile()` method provided by KiCo. This method takes the model to compile and a list of transformation IDs as an input. These transformations are the user selected transformations that should be used for compiling the model. It basically is the input for the interactive compilation. KiCo will take care of the correct SLIC order of these transformations.

Advanced Selection Algorithm

Typically, when calling `compile()` of KiCo version 1.0, one may pass a list of transformation IDs. The respective transformations will be processed in SLIC order.

When selecting a certain transformation, let's say τ' , to be processed, there possibly are other features f in the model that needed to be ex-

6.5. Interactive SLIC Compiler Implementation

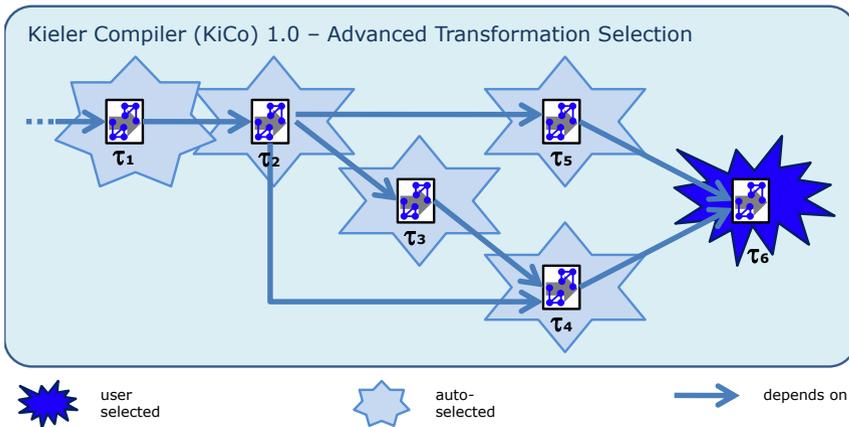


Figure 6.5.2. KiCo 1.0 advanced selection algorithm example

panded before because $f \in \text{NotHandles}_{\tau'}$. However, the transformation τ expanding f may require still other transformations before. An advanced auto-selection algorithm implemented in KiCo helps selecting all (possibly) *required* transformations in addition to a user selected transformation.

An example is illustrated in Figure 6.5.2. In this example, τ_6 is the only user selected transformation. The depending transformation ID lists are illustrated as depends on arrows between the transformations. The reverse of a depends on order is the *required* order: (τ depends on τ') \iff (τ' requires τ). The meaning for both is that τ must be executed before τ' .

The algorithm for auto-selecting τ_1, \dots, τ_5 starts with the selected transformation τ_6 and then follows the *required* transformations, here τ_4 and τ_5 . From there, the algorithm continues recursively until it hits a transformation with no “outgoing” required transformation.

6.5.3 Disadvantage of KiCo 1.0

After using the first prototype of KiCo for a while, some disadvantages became clear.

6. SCCharts Tooling

Dependencies: KiCo 1.0 does not distinguish between produces and not-handled-by dependencies, neither in the extension point interface for transformations declaring plugins nor in the internal graph representation for calculating the advanced selection. When specifying that a transformation τ' cannot handle features expanded by some other transformation τ , there must be an entry for τ' in the dependency transformation ID list of τ . However, this is counter-intuitive and hence error prone. It would be better if KiCo would allow to specify directly for τ' that it cannot handle features expanded by τ .

Furthermore, if the two types of dependencies are not distinguished in the advanced auto-selection algorithm, then more transformations than necessary may be auto-selected to be processed in a compilation run.

Features: KiCo 1.0 only knows about transformations and dependencies between transformations. It makes sense to define a produces and not-handled-by order regarding two transformations, although for specifying properties of a transformation it would be much cleaner to specify a list of *features* that this transformation produces or cannot handle.

Furthermore, KiCo 1.0 only allows to select transformations to be applied. However, if compiling a model M with some features F_M , one may want to select a certain feature f to be expanded by its transformation τ_f or only additionally a certain transformation τ_{f_i} , if alternative transformations exist to expand f .

Model analysis: As KiCo 1.0 does not know anything about features, it is impossible to optimize the compilation progress w. r. t. the features that really exist in a model and the interdependencies between the transformations and the feature selected for compilation. The information is present in theory only but not represented in the implementation.

It would be preferable to have a mechanism that 1. checks for every feature $f \in F$ whether it is present in M or not and 2. auto-selects and processes only those transformations, that are necessary to be processed, having in mind the user selection of features to expand.

Code clones: When implementing transformations, certain functionality such as a special optimization or a consistency check often needs to be done by more than one transformation. If these transformations

6.5. Interactive SLIC Compiler Implementation

do not reside in the same plugin, re-using code gets complicated and leads to additional plugin dependencies. Often, *code clones* were created which heavily harm the maintainability of a compiler or any other piece of software [Kri07]. It would be desirable to have the possibility to compose transformations from even more basic building blocks that also could be easily re-used in other transformations to avoid code clones.

6.5.4 The KiCo 2.0 Advanced Prototype

The drawbacks of KiCo 1.0 finally lead to a partial re-design. The second prototype had several extension points for registering transformations, processors, and features. Processors are atomic building blocks of transformations. Features are expanded by one or more transformations and allow KiCo to check for their existence in a model. KiCo 2.0 further distinguishes between produces, and not-handled-by dependencies. Figure 6.5.3 gives an overview of the design of the KiCo 2.0 prototype¹¹.

Feature Plugin: KiCo lets a Feature plugin contribute a language feature. A feature has a unique ID, a human readable Name, and an `isContained()` method. The ID is used by transformation plugins to declare either to expand, produce or not-handle a feature. The `isContained()` method gets the complete model of type `EObject` as an input and returns true if the feature is contained in the model. This method is internally used by KiCo to enhance the advanced auto-selection and the processing of transformations. The ID, the name, and the is-contained method are part of the interface provided by KiCo.

Processor Plugin: A Processor plugin contributes a processor, i. e., the smallest atomic basic building block. A processor has a unique ID, a human readable Name, and a `process()` method. The `process()` method similarly to the `transform()` method of transformations, takes the model of type `EObject` and (possibly) modifies it. Then it returns the modified

¹¹Many ideas for KiCo 2.0 are based on a design review of KiCo 1.0, see <http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/2015-03-06+KiCo+Design+II>.

6. SCCharts Tooling

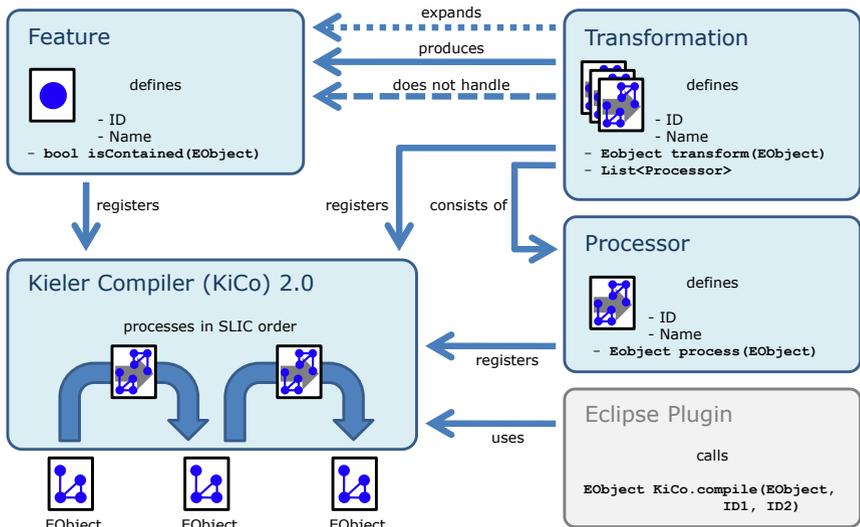


Figure 6.5.3. KiCo 2.0 advanced prototype implementation overview

model or even a completely new model of type `EObject`. A processor has no dependencies by itself. It cannot run by itself but is meant to be part of one or more transformations. Processors enhance the maintainability of the whole compiler as they can simply be re-used without introducing new plugin dependencies. The ID, the name, and the process method are part of the interface provided by KiCo.

Transformation Plugin: As in the KiCo 1.0 implementation, a plugin Transformation defines a unique ID, a human readable Name, and a `transform()` method. The `transform()` method is the central part of a transformation plugin. It takes a model of type `EObject` as an input. It then modifies the model and expands a certain feature that this transformation is dedicated to. It then returns the modified model of type `EObject`. Additionally, a transformation in the KiCo 2.0 implementation may provide a list of processor IDs. These processors constitute the transformation in the

6.5. Interactive SLIC Compiler Implementation

order defined by the list. There is a special placeholder for the `transform()` method. However, the `transform()` method is not required to be used at all, a transformation could purely consist of a list of processors. Using the processor concept in addition to the `transform()` method, it is easy to implement pre or post processors for a certain transformation. These pre and post processors can also be re-used in other transformations without the need of duplicating their code. Note that a transformation is not a processor by itself: It does not have a `process()` method and cannot be part of another transformation. The ID, the name, the transformation method, and the processor list are part of the interface/abstract class provided by KiCo.

KiCo 2.0 distinguishes between produced-by and not-handled-by dependencies regarding transformations and features. It does so for transformations and features. A transformation may produce features that are expanded by other transformations. The \rightarrow_p order is implemented by providing a list of feature IDs that are (possibly) produced. A transformation may not be able to handle certain features. The \rightarrow_{nhb} order is implemented by providing a list of feature IDs that the transformation is not able to handle. Both lists are also part of the KiCo interface.

KiCo Plugin: The KiCo plugin Kieler Compiler (KiCo) 2.0 provides the extension points for registering features, processors, and transformations. At startup, it collects all features, processors, and transformations from contributing plugins using the extension point mechanism. It instantiates all features, processors, and transformations and resolves their dependencies so that they are ready to be used for compilation. The resolved information and the SLIC order are used to derive a SLIC schedule necessary for a concrete compilation run.

Other Plugins: An Eclipse Plugin with a plugin dependency on KiCo may use the central `compile()` method provided. This method takes the model to compile and a list of feature IDs as an input. The appropriate transformations for these selected features are then used for compiling the model. The selected features are basically the input for the interactive compilation (see Section 4.2.2 on page 106). In addition to feature IDs, also

6. SCCharts Tooling

transformation IDs can be passed. This can be used to give preferences for alternative transformations. Again, KiCo will take care of a correct SLIC order of the processed transformations.

Feature Groups

KiCo 2.0 also comes with a notion of *feature groups*. These facilitate 1. the organization of features and 2. the selection of features. A feature group $G \subseteq F$ is able to contain “real features” or other feature groups to build hierarchical structures. Feature groups are also handled as features and hence come with a defined unified ID and a human readable name. A feature group can be asked for its inner features or even for its (fully recursively) resolved inner features.

Advanced Selection Algorithm

One drawback of KiCo 1.0 was that the advanced auto-selection algorithm tended to select more transformations to be processed than actually necessary (cf. Figure 6.5.2). The reasons are that there was no distinction between produces and not-handled-by dependencies and that there was no notion of model features in KiCo 1.0.

This has changed for KiCo 2.0. Figure 6.5.4 shows an example of the enhanced version of the advanced auto-selection algorithm that takes the additional information into account. As shown in Figure 6.5.2, also in this example τ_6 is the only user selected transformation. However, this time, the according **feature** f_6 is selected by the user. Hence, the transformation τ_6 is only implicitly selected.

The produces, not-handled-by, and expands order relations are represented as different types of arrows. Furthermore, features are represented as either being present in the model (f_2) or not being present in the model (f_1, f_3, f_4 , and f_5). Two aspects are worth noting:

1. Because feature f_1 is not present in the model (as f_2) and also not possibly produced, the transformation τ_1 is neither auto-selected nor processed later when compiling the model.
2. Feature f_5 is not present in the model and also not possibly produced.

6.5. Interactive SLIC Compiler Implementation

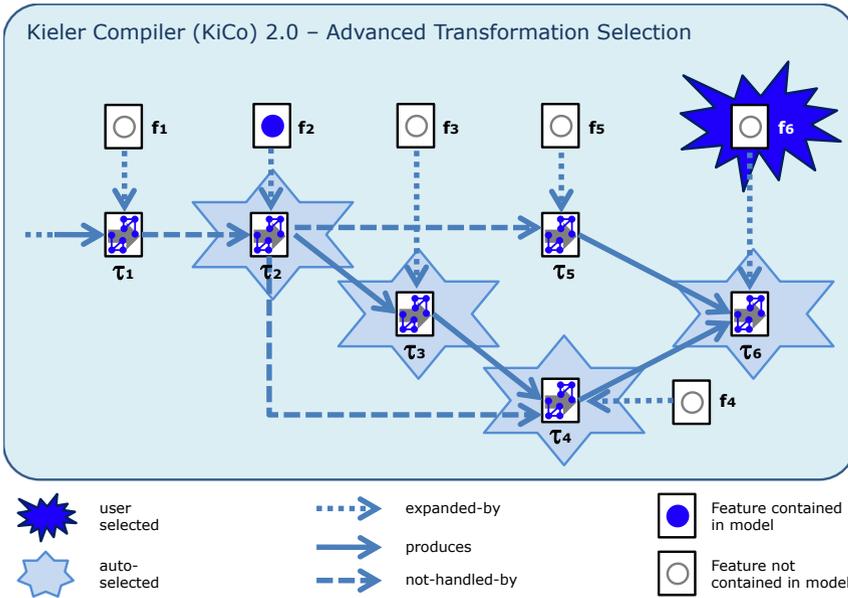


Figure 6.5.4. KiCo 2.0 advanced selection algorithm example

The new algorithm for auto-selecting τ_2 , τ_3 , τ_4 , and τ_6 does not start with the (implicitly) selected transformation τ_6 any more. In contrast, the algorithm starts with model features, i. e., features that are evaluated to be contained in the source model that should be compiled. The algorithm in short does the following to calculate the set T_A of advanced auto-selected transformations:

1. Collect all model's features $F_M = \{f \in F | f.isContained(M) == true\}$.
2. Collect all selected features F_S , these are features f selected
 - (a) because the transformation τ_f was selected, where $f \in Expand_{\tau_f}$, or
 - (b) because the selected feature group F_G contains feature f , or
 - (c) because the feature f is selected directly (as f_6 in the example shown in Figure 6.5.4).

6. SCCharts Tooling

3. Setup a new set F_A of auto-selected features, initially containing all features from F_S : $F_A := F_S$.
4. The algorithm now iterates over all model features $f_m \in F_M$. For every f_m , the algorithm iterates over all selected features $f_s \in F_S$ and tries to retrieve a path $p := f_m, f_1, f_2, \dots, f_n, f_s$ from model feature f_m to selected feature f_s by considering only *produces* edges.
 - ▶ If such a path p exists then add f_m and f_1, \dots, f_n to F_A .
5. After iterating over all pairs (f_m, f_s) , F_A contains all auto-selected features.
6. Now, the algorithm adds for every $f_a \in F_A$ the appropriate τ_{f_a} to a new set T_A of auto-selected transformations by respecting any explicitly preferred/selected transformations (in case of alternative transformations).

Note that the auto-selected features/transformations also contain the user-selected features or transformations. For the UI, one may want to subtract the user selected features or transformations from this set as it is done in our implementation and also in Figure 6.5.4. In the end this is just a design decision and an implementation detail. For the example, as shown in Figure 6.5.4, the features of F_A are f_2, f_3, f_4 , and f_6 and hence the auto-selected transformations of T_A are τ_2, τ_3, τ_4 , and τ_6 .

Enhanced Compilation Performance: KiCo will process a transformation during compilation only if the according feature is really contained (`isContained()`) in the (intermediate) model. The reason is that produces dependencies only indicate that a feature is *possibly* produced by a transformation. However, a transformation might not produce such a feature and if this feature was also not contained in the model before then there is no need to process a transformation (to eliminate a non-existing feature). Compared to the KiCo 1.0 implementation, this enhances the overall compilation performance in addition to not processing transformations which are not on any path p , such as τ_1 of the example. Figure 6.5.5 shows (one possible) final SLIC order that could be used for processing auto-selected transformations according to the model features and the user selection in a compilation run.

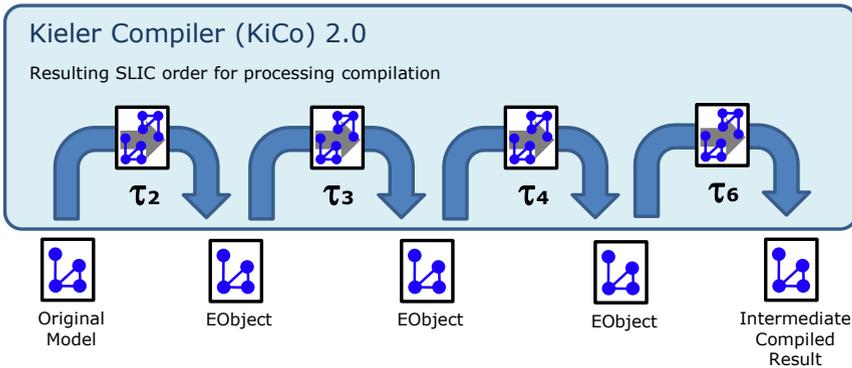


Figure 6.5.5. Example of final SLIC order for processing transformations in a compilation run

6.5.5 SLIC Implementation Notes

All features, processors, and transformations are contributed by Eclipse plugins as explained in Section 6.5.4. Transformations are defined to specify the feature they expand, the features they produce, and the features they cannot handle. These SLIC dependency information is internally represented as a graph structure similar to Figure 4.1.3 on page 97.

Each element in this graph structure has a reference on the transformation it represents, and a set of dependencies and reverse dependencies derived from the produced and not handled features. Note that when the graph is created, it is already known which transformations (or their corresponding features) were selected by the user. Hence, the precedences that arise from the user selection can be considered. Additionally, each element carries a boolean marker flag and an integer order-variable that allows to apply a topological sort algorithm. Finally, note that produces and not-handled-by dependencies are currently represented in the same way. For this reason, the actual implementation of the auto-selection-algorithm slightly differs from a technical point of view.

6. SCCharts Tooling

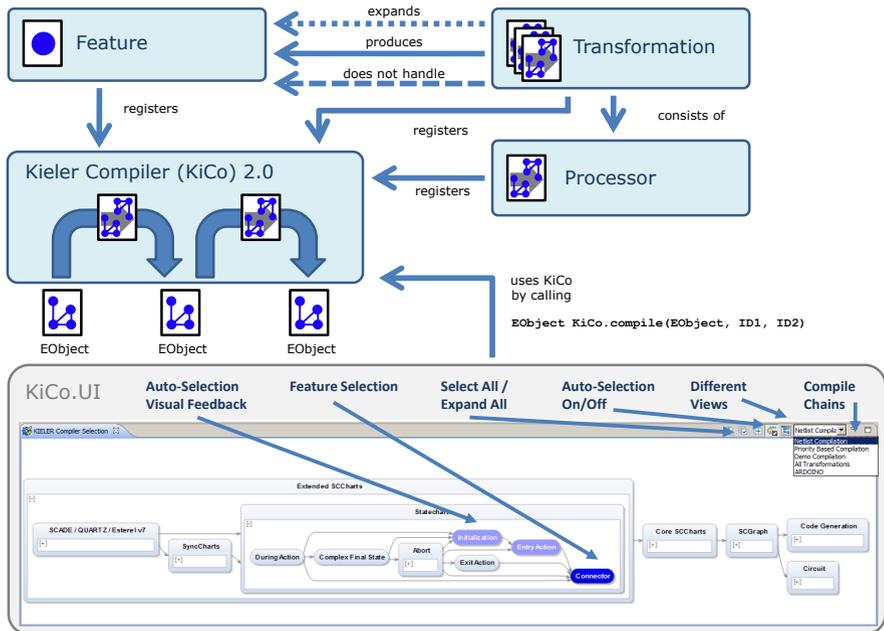


Figure 6.5.6. The graphical user interface plugin infrastructure of KiCo

6.5.6 Interactive Compilation GUI: KiCo.UI

Figure 6.5.1 on page 333 and Figure 6.5.3 exemplified Eclipse Plugins that were calling the central KiCo `compile()` method (see the gray box). Such a plugin is key to an interactive incremental model-based SLIC compilation user story as shown in Figure 4.2.3 on page 112. It serves as a GUI to allow the user to select transformations or features.

Figure 6.5.6 shows the plugin infrastructure for the graphical user interface of KiCo. It basically displays all features and transformations that are registered in KiCo. Note that the UI of KiCo re-uses the concrete statechart notation of SCCharts, but semantically this is not an SCChart. The SCCharts notation was chosen as a design decision because it allows for naturally representing transformations and dependencies: A state represents a fea-

6.5. Interactive SLIC Compiler Implementation

ture or an alternative transformation. Transitions represent dependencies of produced or not-handled-by orders. Hierarchy is used for 1. grouping features and 2. grouping more than one alternative transformation that may exist to expand one and the same feature.

Feature selection: Often, there exists exactly one transformation for a feature. Therefore, if the user selects a feature with a single mouse click to be expanded, the specific transformation follows. For that reason the user interface does not distinguish the transformation and the feature in this case. For example, in Figure 6.5.6 the user selected the feature Connector. Because there is only one transformation expanding the connector feature this is implicitly the selected transformation. In the other case, if there exist more than one transformation for a feature, as it is the case for Abort, then the feature is shown as a hierarchical node where all possible alternative transformations are child nodes (see *Transformation selection*).

Compile chain: The Compile Chain is defined by an extension point of the `KiCo.UI` plugin. For every compile chain, 1. a name must be specified which will be the label for the drop down menu items, 2. a set of features that constitute the compile chain and determine which features are presented to the user, 3. an editor ID of an editor this compile chain is bound to, 4. a priority that determines the order of the drop down menu, and 5. a set of preferred transformations for alternative transformations that are selected if the user does only select a feature for which more than one expanding transformation exists.

Transformation selection: There may be different possibilities for a feature to be expanded, i. e., there are alternative transformations expanding one and the same feature. This is also reflected by the user interface. For example, this is the case for the Abort feature. Hierarchy is used to express that the Abort feature can be expanded by two different transformations, namely Abort and Abort WTO. If only the feature Abort is selected by the user, exactly one of the alternative transformations is selected. If there is a preference given by the Compile Chain then this is used. If there is no such preference, then a transformation is randomly chosen. Hence, it is desired to *always* have a concrete preference for

6. SCCharts Tooling

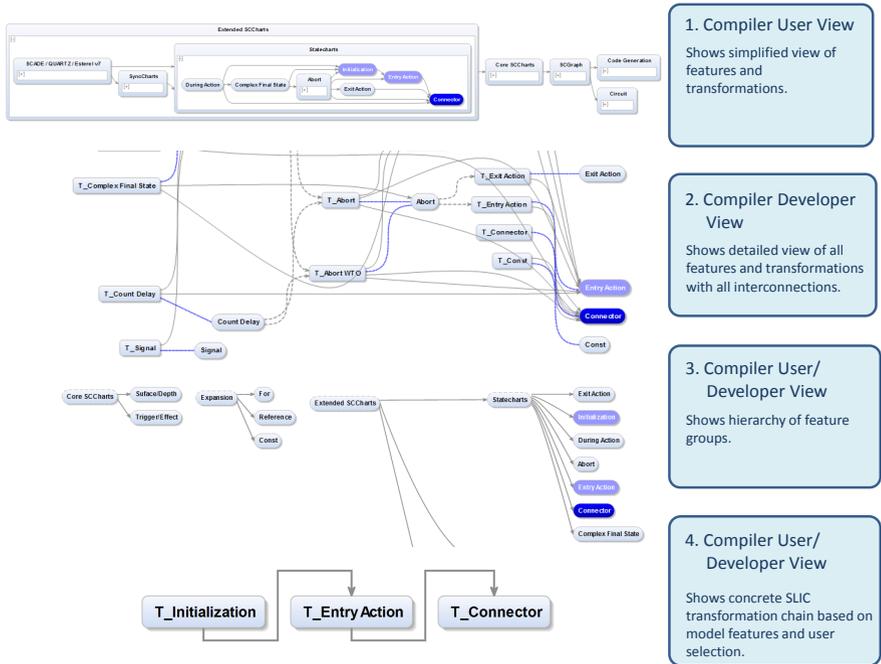


Figure 6.5.7. Different compiler selection views for the user and the developer of KiCo

each group of alternative transformations in all Compile Chains. Other configurations should be considered incomplete. A user may also select a specific transformation in which case he or she overrides a possible preference given by the Compile Chain.

Disabling features/transformations: A user can disable features or transformations with a double mouse click. For example, Complex Final State has been disabled by the user in Figure 6.5.6. If a feature or transformation is disabled, the according transformation is skipped when all SLIC transformations are processed. Disabling an alternative transformation may have the effect of implicitly selecting another alternative transformation.

6.5. Interactive SLIC Compiler Implementation

Select all / expand all: The Compiler Selection view allows to automatically select or deselect all features/transformations. It also allows to expand or collapse all hierarchy elements.

Auto selection: The advanced auto-selection algorithm is used to support the user to select all necessary feature transformations, based on the current model (features) and the selected features to expand. For example, the features Initialization and Entry Action have been auto-selected (cf. Figure 6.5.6) because the model contains an Initialization feature and there is a dependency path to Connector where the Initialization transformation produces Entry actions that are on this path. The auto-selection mechanism can be switched off using the appropriate toolbar button.

Different views: There are additional views that target different use-cases for the compiler user and for the compiler developer. A switch view button can be used to toggle between these different views. Example views are shown in Figure 6.5.7. The default view is a simplified view that hides the details of feature and transformation interdependencies. This is sufficient for the ordinary user of KiCo. However, for the developer, who also needs to take care of correct registration of features and transformations, this view often is not sufficient. A developer view that shows all details helps to validate and correct registration and dependency configurations. Other views allow to verify the hierarchy of feature groups for the developer or help to understand the group hierarchy for the user. Finally, the concrete SLIC transformation chain can be validated or inspected in another view.

6.5.7 Command Line and Online Compilation

The previous section presented the KiCo user interface plugin as one example of how to utilize KiCo for an interactive compilation user story. However, as Figure 6.5.1 on page 333 and Figure 6.5.3 on page 338 suggest, there could be numerous and various plugins using the KiCo infrastructure.

In this section, a server implementation is sketched that was used to validate the generality of the KiCo infrastructure. This server is leveraged to show two use-cases: 1. A command line support and 2. a WWW support for the interactive KIELER compiler.

6. SCCharts Tooling

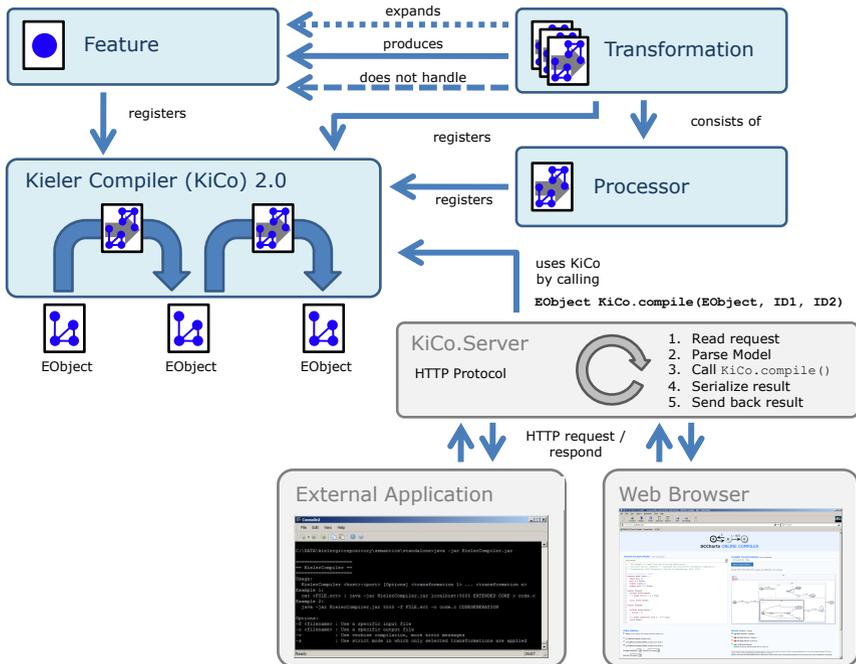


Figure 6.5.8. KiCo server infrastructure. The detailed content of External Application and Web Browser is shown separately in Figure 6.5.10 and Figure 6.5.11, respectively.

KiCo Server

Figure 6.5.8 shows another example of a more generic plugin that uses KiCo. This plugin is the `KiCo.Server`. It uses KiCo for compiling models and follows the ideas of *Compilation as a Service* [PLDM02], but it widens the view for the interactive compilation concept. Hence, it is fair to say that the KiCo server offers *Interactive Compilation as a Service*.

The `KiCo.Server` is a lightweight HTTP server that is able to handle simple GET or POST compilation requests. The model, i. e., all files that constitute the model, are attached in a BASE64 encoded, serialized form. Also, addi-

6.5. Interactive SLIC Compiler Implementation

tional compilation options, such as the selected features/transformations, are passed as GET or POST parameters. Reading a request is done by a generic lightweight HTTP server. The model is parsed using a special parsing algorithm that tries to find the best suitable XML or Xtext parser registered. Afterwards, the KiCo central `compile()` method is called. The result is either text or code that is directly sent back as the result or it is a model of type `EObject` in which case it needs to be serialized before.

Figure 6.5.8 also shows two example applications that make use of this generic server. One is an external command line Java application that allows to use the local or even remote KIELER Compiler, e. g., from shell scripts or other programs. It is noteworthy that it does not have to be a Java application, but in general could be any program that is capable of speaking the HTTP protocol which is available in libraries for almost any programming language of today. The other example application is a simple JavaScript-based website that uses the HTTP protocol directly and provides a simple interactive compilation testbed without the need of installing any program. Both examples are sketched in the next two sections.

Figure 6.5.9 shows the GUI of the KiCo server plugin. The user first needs to enable the server by opening the control window (1.). They may then configure the listening port for connections (2.). By default this is 5555. Also, the checkbox must be checked in order to start the server. The server then runs as an Eclipse task (3.) in the background.

Command Line Compilation

Section 6.5.7 introduced the KiCo server plugin as an example of an Eclipse Plugin that uses the KiCo infrastructure as illustrated in Figure 6.5.1 on page 333 and Figure 6.5.3 on page 338 (gray box).

This KiCo server plugin can be connected by any HTTP speaking program. This is utilized in a Java command line application that aims in compiling SCCharts from the command line.

This approach has the following benefits:

- ▶ The compiler can be called directly from the command line when no GUI is present or preferred.

6. SCCharts Tooling

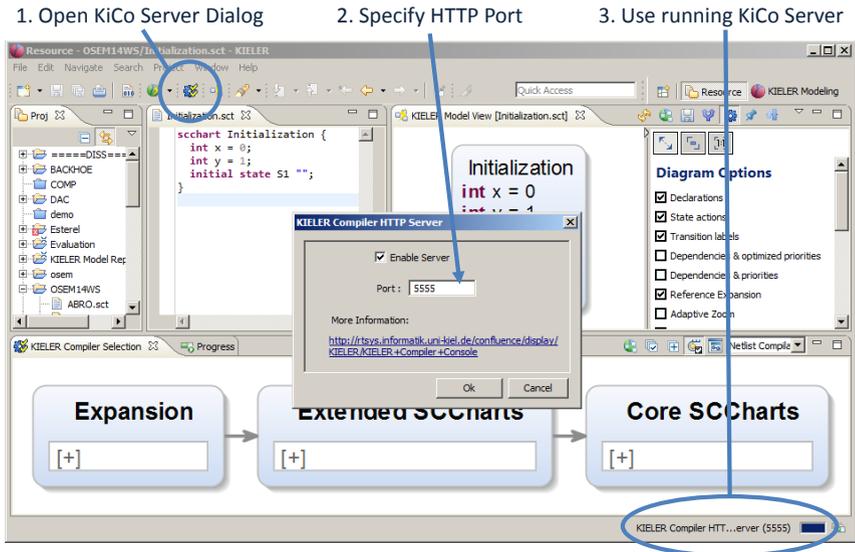
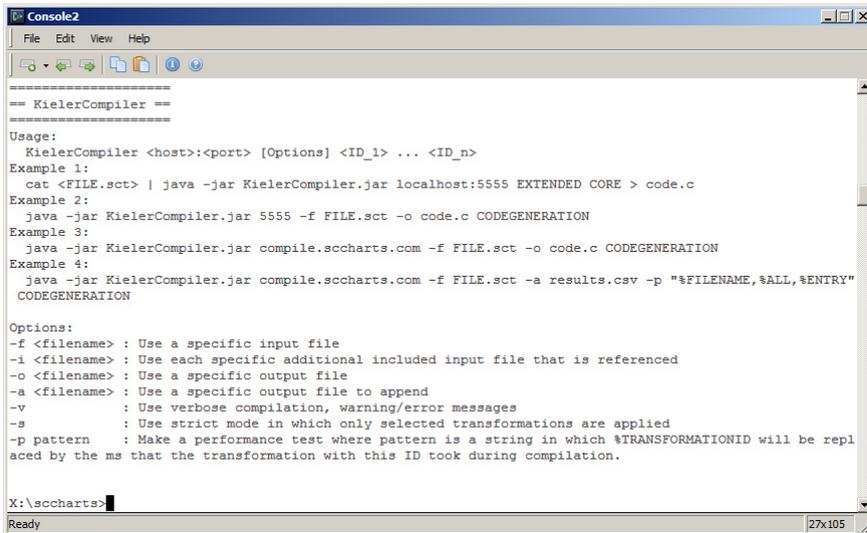


Figure 6.5.9. KiCo server GUI

- ▶ The compiler can be called from build-management scripts like Make-files [Cal10].
- ▶ The compiler can be called from shell scripts in batch processing for mass file compilation or regression tests.
- ▶ The compiler can be run and accessed on a remote (powerful) server. It can also be run locally.
- ▶ The compiler used for all connecting Eclipse plugins is exactly the same. This reduces maintenance efforts because there is only a single source.

Figure 6.5.10 shows the options that were implemented for this very lean Java command line front end of the KiCo compiler. The port and optionally a host are given depending on where the server part of KiCo runs. Transformation or feature IDs to compile are added at the end. The input can either be standard input which allows to use pipes or an input file. Also, the output can be standard output or a given output file. Warning

6.5. Interactive SLIC Compiler Implementation



```
==== KielerCompiler ====
Usage:
  KielerCompiler <host>:<port> [Options] <ID_1> ... <ID_n>
Example 1:
  cat <FILE.sct> | java -jar KielerCompiler.jar localhost:5555 EXTENDED CORE > code.c
Example 2:
  java -jar KielerCompiler.jar 5555 -f FILE.sct -o code.c CODEGENERATION
Example 3:
  java -jar KielerCompiler.jar compile.sccharts.com -f FILE.sct -o code.c CODEGENERATION
Example 4:
  java -jar KielerCompiler.jar compile.sccharts.com -f FILE.sct -a results.csv -p "%FILENAME,%ALL,%ENTRY"
  CODEGENERATION

Options:
-f <filename> : Use a specific input file
-i <filename> : Use each specific additional included input file that is referenced
-o <filename> : Use a specific output file
-a <filename> : Use a specific output file to append
-v           : Use verbose compilation, warning/error messages
-s           : Use strict mode in which only selected transformations are applied
-p pattern   : Make a performance test where pattern is a string in which %TRANSFORMATIONID will be repl
              aced by the ms that the transformation with this ID took during compilation.

X:\sccharts>
Ready
```

Figure 6.5.10. KiCo command line usage

and error messages can be collected and the command line front end can also be used to run script-based performance (regression) tests.

The implementation of the Java command line application is a very lean but effective. It simply collects all parameters and encodes them as a HTTP message. This is sent to the KiCo server part. The HTTP response is decoded and written to the specified destination.

Online Compilation

Section 6.5.7 introduced the KiCo server plugin as an example of an Eclipse Plugin that uses the KiCo infrastructure as illustrated in Figure 6.5.1 on page 333 and Figure 6.5.3 on page 338 (gray box).

This KiCo server plugin can be connected by any HTTP speaking program. The previous section showed how this was utilized by a Java command line application in order to make the compiler accessible from the command line. This section demonstrates how the server can be easily

6. SCCharts Tooling

accessed by a lean JavaScript-based website as a front end. Such a website has the following benefits:

- ▶ The server can be used without installing any additional software as a web browser is sufficient.
- ▶ The website provides an integral view for compiling and rendering graphical (output) models.
- ▶ The server can be easily accessed from handheld devices like tablets or smartphones.
- ▶ There are less barriers for beginners to get used to the compiler and the language.
- ▶ Small examples can be tried out online.
- ▶ A website can often be easily embedded in many online submission classroom systems for teaching purposes.

The *SCCharts website*¹² (see Figure 6.5.11) has a textual input edit field for the source model in SCT format. It has an input field for specifying feature and transformation IDs. The output can be viewed either as the serialized textual SCT or SCG format or an automatically synthesized diagram. A server can be chosen that can either run locally (default) or remotely. Error and warning messages are also parsed and presented to the user.

The implementation is really lean JavaScript that starts a timer to start compilation and rendering with a certain delay after the last modification to the input model. The SCT from the input field is encoded together with the other parameters and sent to the remote KiCo server asynchronously using a standard `XMLHttpRequest` component. Once the compiled result is returned, the textual output may be presented to the user or, if they selected the visual diagram output, the output is forwarded to another (similar) *rendering server* based on `KLighD`, which returns a diagram image in the desired format.

¹²<http://www.sccharts.com>

6.6. Automatic Validation

1. Enter/edit textual SCChart (SCT)
2. Enter or select compile feature/transformation IDs
3. Modify options (e.g., diagram or text)
4. Select KiCo server
5. View compilation result

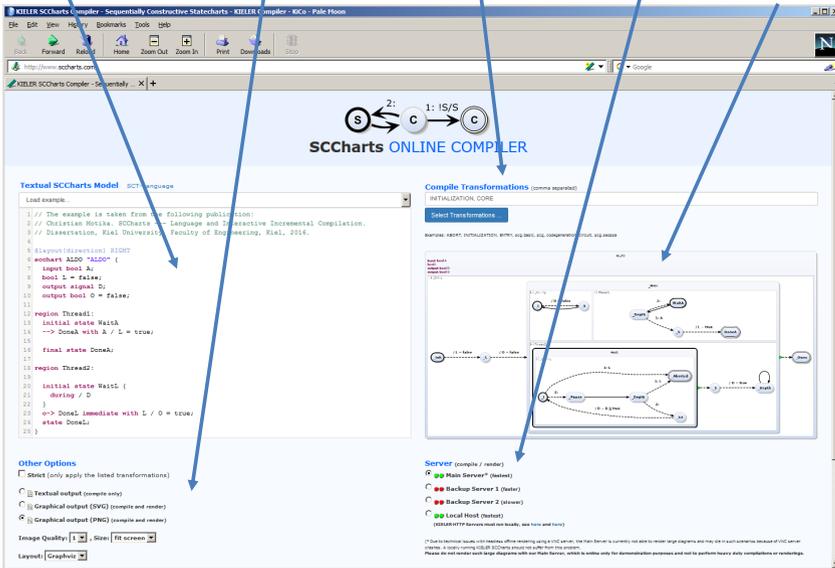


Figure 6.5.11. KiCo online compiler: <http://www.sccharts.com>

6.6 Automatic Validation

The following sections will cover decisions and details regarding the implementation of regression tests for the SCCharts compiler and details on the KIELER regression testing infrastructure.

6.6.1 Regression Tests

Regression tests [RNHL99] are essential to validate that a fix or enhancement of the compiler does not break other parts of it. It is essential to have a relatively simple workflow for including new tests. Ideally, for every essential feature or for every fixed error, one or more models (and scenarios) are created. These test and ensure that these features or fixes are not broken

6. SCCharts Tooling

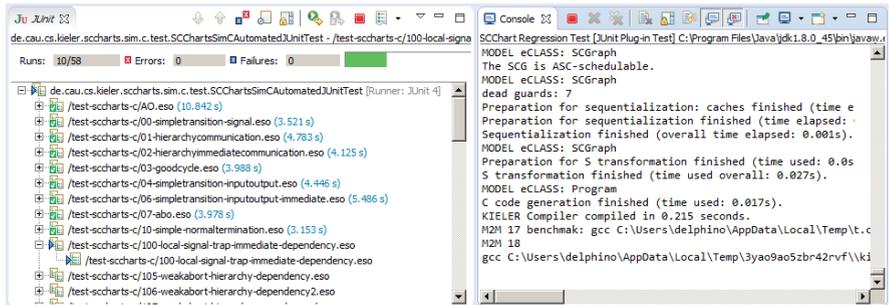


Figure 6.6.1. KIEM JUnit integration running regression tests

in the future. Such regression tests should be run for every modification that is done to the compiler, e. g., in a continuous build setup that is triggered regularly or by new commits to the compiler repository.

For the SCCharts case, these models are SCCharts and the scenarios are execution traces, i. e., ESO trace files with several ticks of output reactions for given inputs.

For the purpose of automatically running such regression tests, a JUnit integration is appropriate. Figure 6.6.1 shows a screenshot of a running instance of the JUnit integration of KIEM that is discussed in Section 6.6.2.

Generic Semantic Validation

Semantic reaction validation seems natural for reactive systems that constantly react to the inputs with computed output reactions. This means to consider the reactive system and also its compiled system model as a black box which has the following advantages:

- ▶ Any compiler changes that do not effect the semantics of the language will not break any tests.
- ▶ The concrete compiler does not matter as long as it produces code that behaves the same.
- ▶ The behavior of compiled models is captured in ESO files and can be shared by very different compilers.

- ▶ This makes it easy to validate the compiler w. r. t. a reference implementation.
- ▶ Meta model changes, changes in the internal structure, or renamings will not break tests.
- ▶ Runtime errors, e. g., when performing M2M compilation transformations are also captured.

Syntactic Validation

In contrast to semantic reaction validation, it would be possible to syntactically validate concrete M2M transformation outputs. However, this way, even very small changes to the compiler may invalidate many test cases which then need to be updated. It may be hard to determine which test cases break because of intended changes (false alarms) and which one break because of unintended, wrong changes to the compiler. Furthermore, it would not be easily possible to automatically retrieve test cases from a separate reference compiler implementation. Also, the benefit of syntactic over semantic validation is questionable because if there are enough very basic and simple tests for semantic validation then it may even be clearer which part of the compiler is affected in case of a test failure since *false alarms* are minimized w. r. t. syntactic validation. Table 6.6.1 summarizes these rationales that led to our decision to use semantic validation only.

SCCharts Use-Case

For the KIELER SCCharts compiler we use semantic reaction validation for the following purposes:

- ▶ Comparing different compilation strategies (transformation options).
- ▶ Comparing different compilers (priority-based vs. circuit-based).
- ▶ Comparing different languages + compilers (SCCharts vs. Esterel).

The following sections will explain how this is done and implemented for the KIELER SCCharts compiler use-case.

6. SCCharts Tooling

Table 6.6.1. Semantic vs. syntactic validation

	Syntactic validation	Semantic validation
Easy to implement	+	-
Very specific tests	+	+/-
Fast to execute	+	+/-
Few false alarms	-	+
Maintainability	-	+
Re-use tests	-	+
Reference compiler	-	+
Easy track down errors	-	+

6.6.2 KIEM JUnit Test Runner

The purpose of this integration is to automatically compile and execute models. These models are run with predefined inputs from ESO files. The reaction of the running models, i. e., the computed outputs of the models per tick, are then compared to expected outputs as loaded also from the ESO file. This is done for a possibly large collection of pairs of models and corresponding ESO files. The integration is sketched in Figure 6.6.2.

The KIEM JUnit Test Runner is configured with a list of directories and a file extension for recognizing model files. Based on the file names of the model files in this directory, it tries to find corresponding ESO files for each model. It also searches for a KIEM configuration file (A). The configuration file for example contains information which model simulator is used, which transformations are selected, or how to validate outputs and expected outputs. For every model and ESO file pair (B), the test runner then instantiates a new KIEM execution, initializes this (which typically means the model simulator compiles the model), and (C) runs the model for all ticks defined in the ESO file.

For every tick, KIEM will 1. load the inputs from the ESO file, 2. pass these inputs to the model simulator, 3. load the expected outputs from the ESO file, and 4. compare the expected outputs with the output reactions of the model simulator. If these differ, 5. a failure is raised. If a failure is

6.6. Automatic Validation

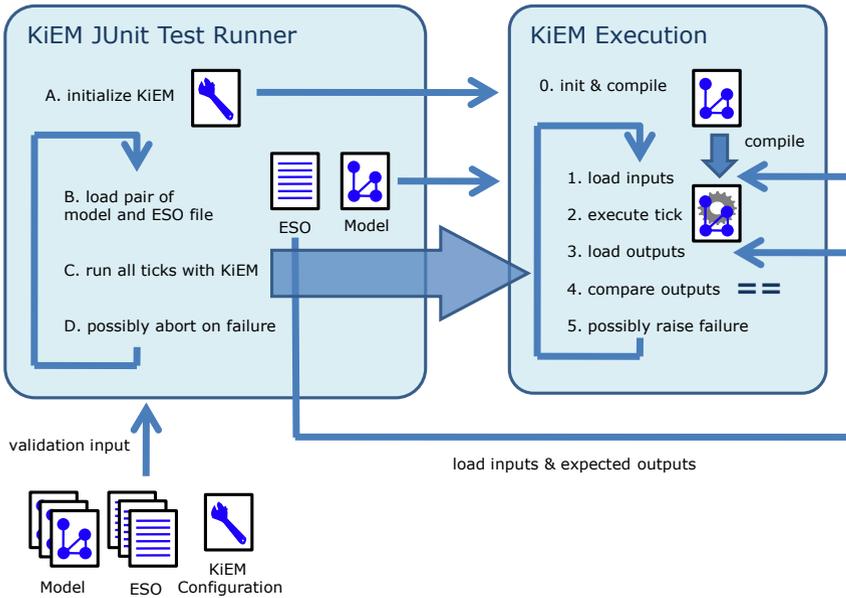


Figure 6.6.2. KiEM JUnit integration schematics

raised on KiEM level, the test runner (D) may possibly abort any further testing based on its configuration.

To be exact, these steps are not fulfilled by KiEM directly, but by KiEM DataComponents that are configured in an execution schedule, the KiEM Configuration file. If a failure is detected, a concrete KiEM test runner may abort the validation or decide to proceed with testing the rest of the model files. Note that an ESO file can hold one or more traces. For the sake of simplicity, this is omitted in Figure 6.6.2. The test runner is responsible to configure the KiEM DataComponent which loads the ESO file to run all traces in ascending order, one trace after the other.

6. SCCharts Tooling

Workflow

It is essential to have a simple workflow in order to integrate new regression tests because this is intended to be done frequently when extending or maintaining the compiler. The workflow should not hinder the modeler to make progress.

The KIEMJUnit Test Runner will work for different languages and compilers. It is abstract and must be subclassed. The concrete implementation needs to declare a list of directories. The tests are then run in the order in which their directory appears in this list. Within a directory, the alphanumeric order of the test model names determines the order in which the tests are processed. If a test runner for a model language and a compiler is implemented then it is simple to integrate a new regression test with the following advantages:

- ▶ Integrating a new test means to create a model and a corresponding ESO file and to add both of them to the desired test directory. No code or configuration must be touched.
- ▶ Changing existing tests means updating a model and/or a corresponding ESO file. No code or configuration must be touched.
- ▶ Using several directories, e. g., for *basic simple*, *basic advanced*, and *complex* regression test models has the advantage that, e. g., if already simple tests fail then the advanced tests may not even run. Hence, several directories allow a coarse grain subsetting and ordering of all regression tests.

6.6.3 ESO File Creation

Having regression test models and corresponding ESO files is essential to validate and maintain the compiler. Ideally, there exists an appropriate ESO file for every model. In practice this is often not the case.

Figure 6.6.3 shows two common ways how to obtain an ESO file for a concrete model either manually or automatically from a reference compiler.

Manual creation: The behavior of the models may be obvious when these models are simple enough, e. g., each model only tests a certain feature. In such cases it is common to specify the behavior by manually creating

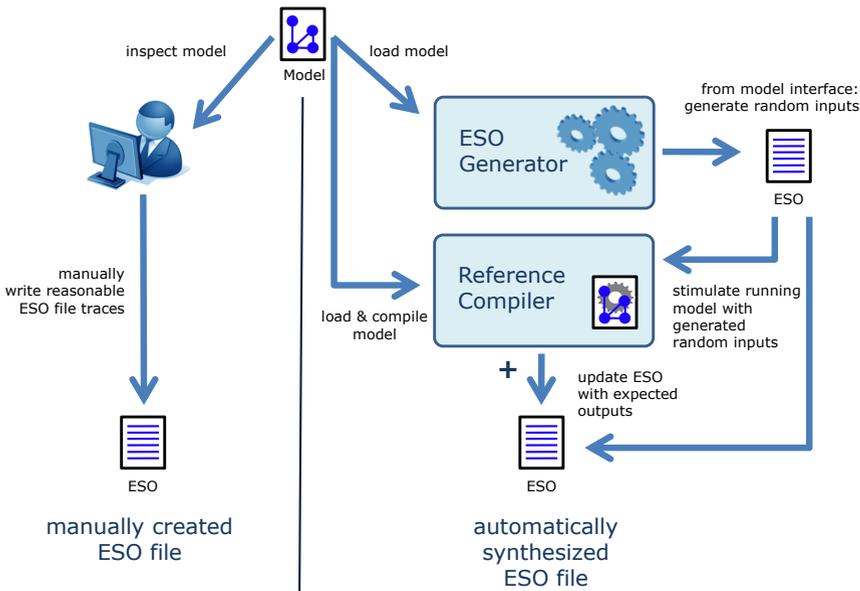


Figure 6.6.3. Creating ESO files manually (left) or automatically (right)

an ESO file or to use a manually stimulated simulation run trace. This is shown in Figure 6.6.3 on the left side.

Automatic creation: For larger and more complex models, e. g., taken from projects, books, benchmark suites, other tools, or other sources, it is often not appropriate to build ESO files manually. Then it is beneficial to have a reference compiler in order to generate ESO files that capture the behavior.

1. If the current compiler version is known to be stable and the purpose of the regression tests is just to not break the current status of the compiler then the current compiler may serve as a reference compiler.
2. If the current compiler version is not known to be stable or the purpose of the regression test is to validate new functionality of the compiler then a separate reference compiler is helpful.

6. SCCharts Tooling

Automatic ESO file creation is shown in Figure 6.6.3 on the right side. The ESO Generator reads the interface of the corresponding model and generates random inputs for a given number of ticks and traces. The Reference Compiler loads and compiles the Model. Then the compiled model is stimulated with the previously generated inputs from the ESO file. The output reaction is used to update the ESO files, i. e., to add the outputs for each trace and tick. These outputs later become the *expected outputs* when performing regression tests.

ESO files in KIELER usually are created by recording a trace using a reference compiler (see above) and then saving the trace as an ESO file, which can be done by the Synchronous Signals view, see Section 6.3.3 on page 320.

6.6.4 Benchmark Regression Tests

Regression tests effectively support the development process to maintain stability w. r. t. correctness of the compiler and its single compiler transformations by semantically validating transformation results. However, correctness may not be the only criterion to be preserved for a compiler during its development process, i. e., while extending and expanding it or while fixing buggy parts. Compilation speed and even more the size and reaction time of the resulting code may be equally important. For this reason it seems appropriate to include an automatic benchmarking also into regular regression tests and to raise warnings if benchmark properties get worse significantly, i. e., exceed a certain level of tolerance.

As explained earlier in Section 6.3.4 on page 320, the SCCharts simulation KIEM DataComponent integrates benchmarking for the following certain compiler properties 1. compile-time, 2. model size, 3. executable size and 4. reaction tick time.

This benchmarking is re-used for the regression tests by integrating it with the KIEM JUnit regression test runner (cf. Section 6.6.2).

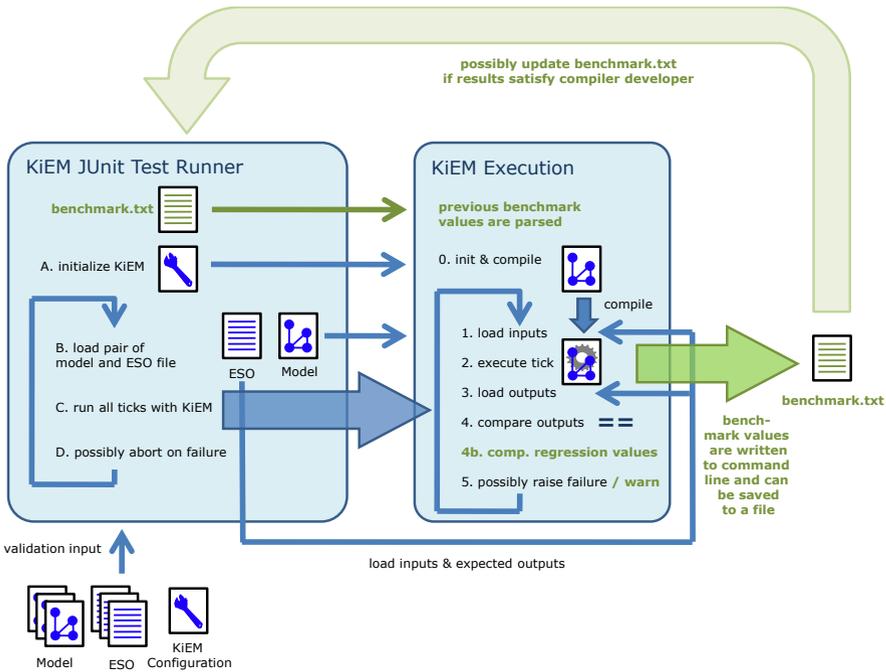


Figure 6.6.4. KiEM JUnit integration schematics enhanced with benchmarking

KIEM JUnit Benchmark Regression

Figure 6.6.4 shows the KiEM JUnit integration scheme from Figure 6.6.2 but enriched by the benchmarking (green parts). In the right part, the large green arrow indicates that the benchmark output is printed to the standard output in a structured way. The textual output of the regression test can be saved as-is to a text file named `benchmark.txt`. This text file serves as an input for any future runs of the regression test when parts of the compiler may have been changed. The persisted benchmark values are parsed and compared to the current ones. In case certain limits are exceeded, a performance warning is raised. These tests should be inspected carefully

6. SCCharts Tooling

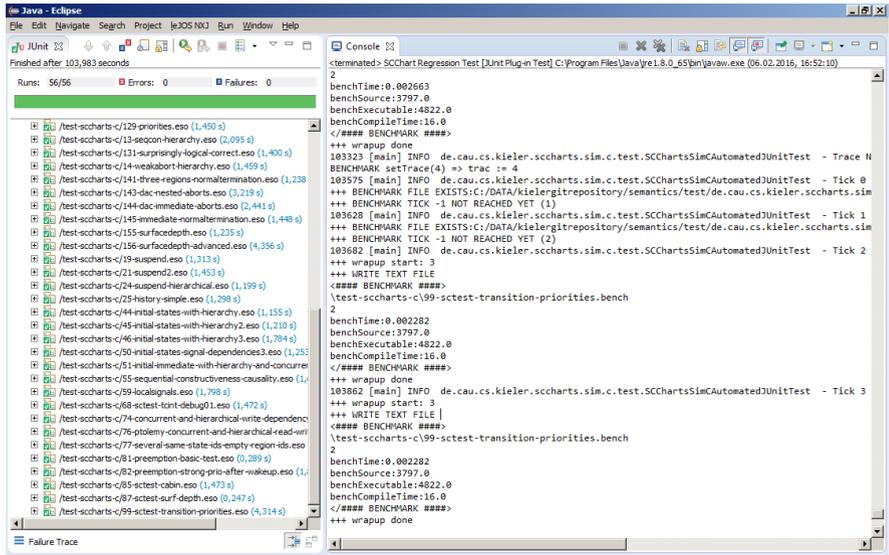


Figure 6.6.5. Benchmark output during SCCharts regression tests

by the compiler developer. If the new benchmark test results satisfy the compiler developer, benchmark.txt can either be replaced completely by the current log output or it can be updated selectively. This is relatively easy because any textual editor can be used to edit/update values that are part of benchmark.txt.

Figure 6.6.5 shows the structured benchmark outputs in a regression test run that can be executed on a server but also locally. Note that different machines may lead to significantly different benchmark values. Hence, the benchmark.txt file is machine-specific.

Regression Test and Limits

Figure 6.6.6 shows a KIEM schedule for a headless server regression test run that includes benchmarking. A KART - Replay/Record Input DataComponent is used to replay an ESO trace file which comes with each SCChart test

6.6. Automatic Validation

Component Name / Key	Value	Type	Master
Synchronous Signal Resetter		<input checked="" type="checkbox"/> Observer/Producer	
KART - Replay/Record Input		<input checked="" type="checkbox"/> Observer/Producer	
SCCharts / SCG Simulator (C)		<input checked="" type="checkbox"/> Observer/Producer	
KART - Validate/Record Output		<input checked="" type="checkbox"/> Observer/Producer	
Benchmark Test		<input checked="" type="checkbox"/> Observer/Producer	
Model File	[ACTIVE EDITOR]		
Benchmark Marker	benchTime, benchSource, benchExecutable, benchCompileTime		
Absolute Tolerance	0,0,0,1000		
Relative Tolerance	100,10,10,10		
Consolidate Ticks	1,0,0,0		
Cmd Line Output/Input	true		

Figure 6.6.6. KIEM schedule for SCCharts regression tests

case. This component is responsible for stimulating the SCCharts simulator with predefined inputs according to a specific trace in the ESO file. A KART - Validate/Record Output DataComponent is used to validate the SCCharts simulator outputs by comparing them to the desired outputs as defined in the same trace.

The generic Benchmark Test DataComponent is responsible for both: 1. Gathering the benchmark outputs of the simulator and printing them in a parsable way to the command line and 2. supervising the performance limits that can be specified in its properties, as shown in Figure 6.6.6. The properties allow to configure the following:

Benchmark Marker: Specifies the different benchmark values to supervise by comma-separated marker IDs.

Absolute Tolerance: For each specified benchmark value, give an absolute tolerance (offset) that the resulting new benchmark result is allowed to be worse.

Relative Tolerance: For each specified benchmark value, give a relative tolerance (percent) that the resulting new benchmark result is allowed to be worse.

Consolidate Ticks: For each specified benchmark value, specify whether the new benchmark result is checked individually in each tick (0) or at the end of all ticks for the average of all consolidated tick results (1).

Cmd Line Output: Specifies whether the structured log output to the command line should occur.

6. SCCharts Tooling

Note that the absolute tolerance and the relative tolerance add up to constitute the concrete *performance limit* that is individual for each benchmark value. This performance limit decides whether to tolerate a worse performance or to raise a performance warning or error.

Practicality & Validation: The Model Railway Demonstrator



Figure 7.0.1. Installation of the model railway demonstrator (from [Mot09])

7.1 Introduction

The model railway demonstrator¹ shown in Figure 7.0.1 is a practical lab at Kiel University since 1995. Originally, it was built and run by the group of Prof. Kluge. In 2006, it was transferred to the real-time and embedded systems group² of Prof. von Hanxleden. Hörmann developed the third generation [Höh06] of the demonstrator in 2006 that was effective until April 2015. The 4th generation of the model railway was developed by Wechselberg [Wec15] based on a conceptual proposal [Mot14a]. It is effective since April 2015. Figure 7.1.1 compares both concepts for the communication hardware of the 3rd generation and the 4th generation. The proposal [Mot14a] suggests to leave the current periphery and its connections untouched as seen in the upper part of both concepts. It further suggests to replace the special power electronic boards by Arduino micro controller boards. Also, it suggests to replace the PC104 computers by Raspberry Pi mini computers. Where the Raspberry Pis connect to each other over Ethernet, each Raspberry Pi is attached to several Arduinos by a serial USB connection. The Arduinos are then attached to the periphery directly or indirectly by means of other HW components.

In summer term 2014, our group hosted and supervised a railway project³ in order to evaluate SCCharts as a language and our KIELER SCCharts tooling. During the railway project, seven participants worked about six months with SCCharts to build a controller that runs up to eleven concurrent trains on the installation. Detailed information about the project and its results can be discovered in a technical report [SMSR⁺15].

Sections 7.2 and 7.3 give insights and some results of the model railway project that took place employing the 3rd generation of the model railway to evaluate the SCCharts language and tooling for developing a medium-size and complex controller model.

¹<http://www.informatik.uni-kiel.de/~railway>

²<http://www.rtsys.informatik.uni-kiel.de>

³<http://rtsys.informatik.uni-kiel.de/confluence/display/SS14Railway>

7.2. SCCharts Model Railway Project

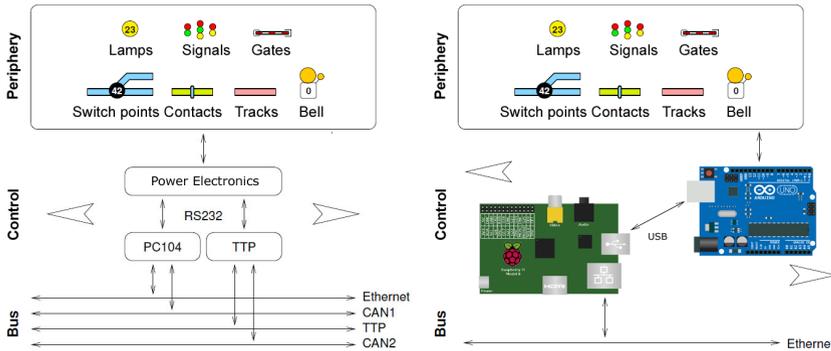


Figure 7.1.1. Conceptual view to the communication hardware of the 3rd generation (left) and the 4th generation (right) (adapted from [Höh06, Mot14a])

7.2 SCCharts Model Railway Project

In summer term 2014, a students project [SMSR⁺15] was launched where the participants were given the task to build a controller for the model railway system demonstrator. This controller should be modeled using the SCCharts language and the KIELER implementation of SCCharts tooling including the SLIC-based SCCharts KiCo compiler approach presented in Chapter 4.

7.2.1 SCCharts Topology Scenario

At the time of this project, the 4th generation of the model railway was not yet available. Though, the 3rd generation was used. However, because the 3rd and the 4th generation both use the same API, the SCCharts controller can run together with both model railway generations without any changes. Together with the current 4th generation, the SCCharts controller would run in the scenario sketched in Figure 7.1.2. The generated C tick function is called in a reactive fashion (cf. Figure 1.0.2 on page 3). Inputs, i. e., reed contact events, are generated by the concurrently running model railway

7. Practicality & Validation: The Model Railway Demonstrator

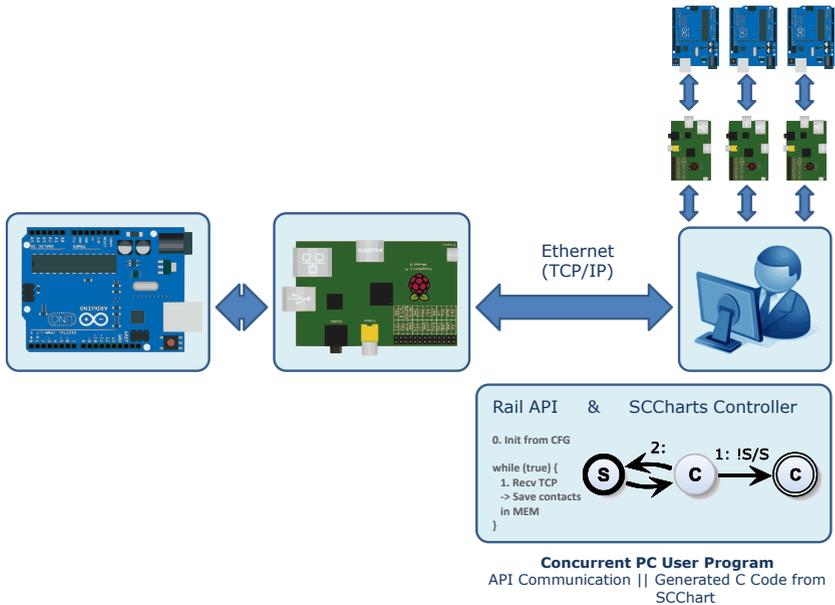


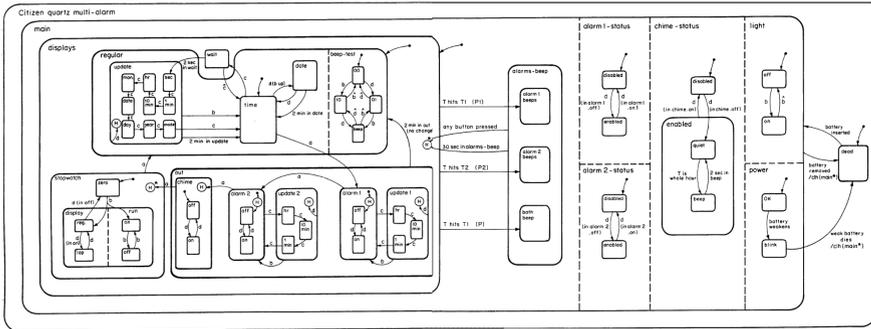
Figure 7.1.2. The SCCharts controller is running centralized on a PC, concurrently using the model railway API. The generated C tick function representing the reactive SCCharts model is called in a cyclic fashion, where inputs are reed contact values and outputs are track speed and switch point direction commands (cf. Figure 1.0.2 on page 3).

API hooked up to the Raspberry Pis. Outputs from the SCCharts model are commands to set the speed of tracks or change switch point directions. The commands are also functions of the model railway API.

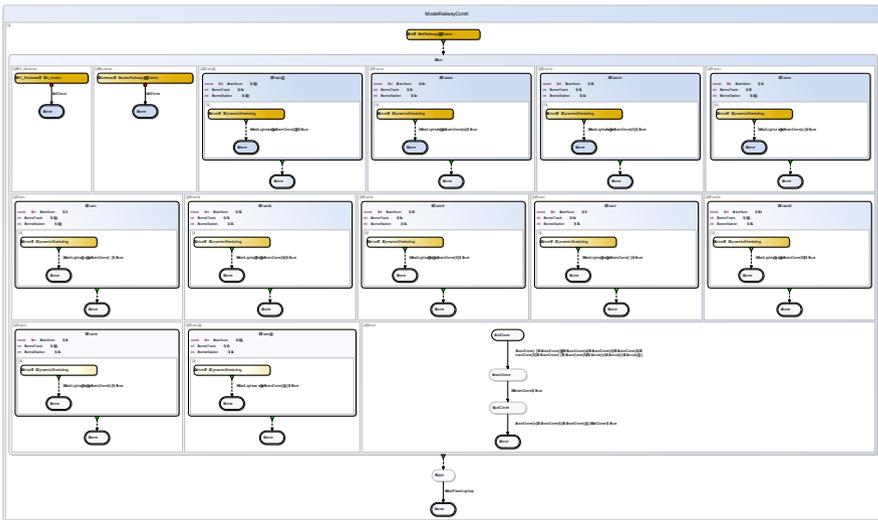
7.2.2 Complex System Modeling with SCCharts

The final controller is able to drive eleven trains simultaneously with integrated dead-lock and live-lock avoidance. Figure 7.2.1b shows only the top level of the hierarchical SCChart of this controller which fully expands to 135,000 states, 152,000 transitions, and 17,000 concurrent regions after

7.2. SCCharts Model Railway Project



(a) Harel's Wristwatch example as taken from *Statecharts: A Visual Formalism for Complex Systems* [Har87] with about 100 states



(b) Top layer of SCCharts train controller with 57 visible states containing 1,628 modeled states and 135,000 states after expansion

Figure 7.2.1. SCCharts controller vs. Harel Wristwatch: SCCharts language & tooling turned out to be practically usable for complex systems.

7. Practicality & Validation: The Model Railway Demonstrator

eliminating all reference states by a reference state compiler transformation. 1,628 states were modeled manually together with 2,219 transition and 183 concurrent regions. Figure 7.2.1 compares the model railway controller with 1,628 modeled states to the size of David Harel’s Wristwatch example (cf. Fig 7.2.1a) that has about 100 states. Hence, compared to the Wristwatch example, which was considered a *complex statechart* back in 1986, we would also call the SCCharts model of the railway controller at least a medium-size real-world complex system.

The model railway SCChart compiles using the presented SLIC-based KIELER tool chain in about two minutes and generates about 650,000 lines of C code. Still, the response time of the running controller was measured to be smaller than two milliseconds on a standard PC.

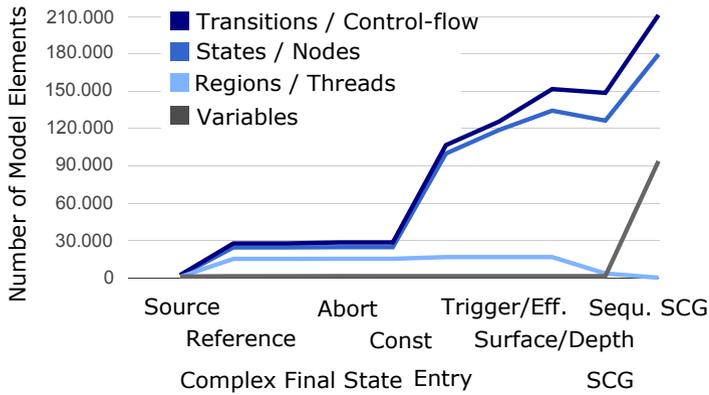
These results indicate that the SLIC approach as well as the SCCharts language and tooling is practically usable for at least medium-size real-world complex reactive systems. In Section 7.3, we additionally support this hypothesis by evaluating some survey evaluation results.

Hiding Complexity, Extendability and Maintainability

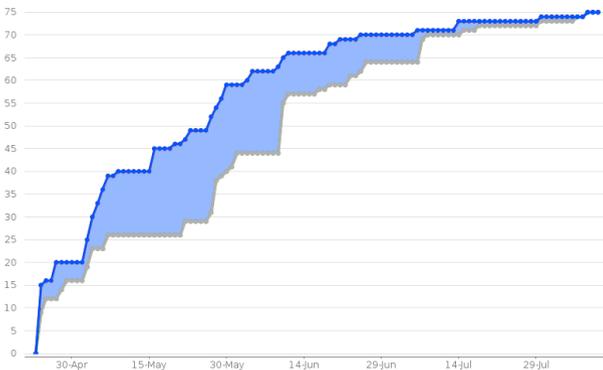
We measured the number of model elements for the SCCharts model railway controller example at every intermediate stage of the SLIC compile chain (cf. Figure 5.0.1 on page 114). Figure 7.2.2a shows the result and suggests how much complexity of the resulting sequentialized SCG model could be hidden by using Extended SCCharts features for modeling the complex behavior of this controller.

The students were not only using our SCCharts compiler tool chain, but also struggling with teething troubles of our early prototype compiler. This resulted in a number of bug reports, especially in the middle of the project when the students started modeling. As Figure 7.2.2b attests, we were able to quickly resolve most of the problems without introducing more new problems. This validates maintainability of the model-based SLIC approach used for the compiler. Additionally, new feature requests such as reference state expansion arose during the project and could be integrated into the existing compiler, which validates extendability of the overall approach. These features are documented elsewhere [SMSR⁺15].

7.2. SCCharts Model Railway Project



(a) Hiding complexity by using Extended SCCharts features: Expansion of SCCharts features down to sequentialized SCG elements gives an idea of the complexity of this model.



(b) Tickets as opened (blue) and closed (gray) in the bug tracker during the period of this project, which validates maintainability and extendability of the model-based SLIC compiler approach

Figure 7.2.2. The SCCharts SLIC-based compilation approach turns out to be practically usable even for complex models and to be maintainable and extendable (from [SMvH15]).

7.2.3 SCCharts Compilation Performance

We tested [SMSR⁺15] the performance of the SCCharts compiler w. r. t. three different versions of its development 1. June 2014, 2. August 2014, and 3. January 2015. The June and August compiler versions were influenced directly from the running railway project. Many improvements could be made and incorporated into the August version. The January version shows that these improvements are still retained in the most current version of the SCCharts compiler.

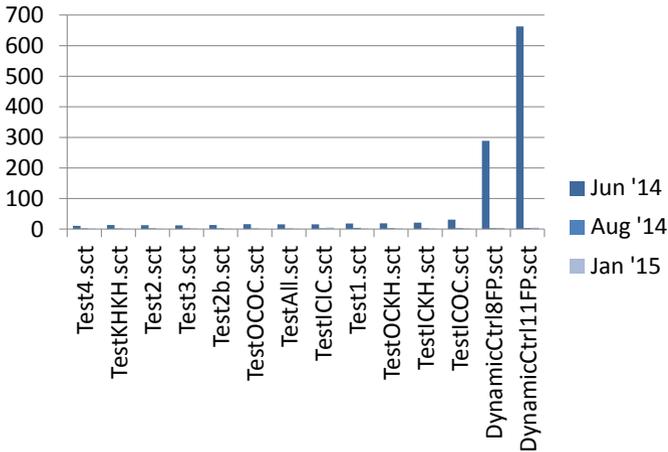
We measured the compile-time in seconds, the pretty-printed target code size in lines of code and a normalized compile-time per line of generated target code. We performed our experiments on an Intel Core 2 Duo T9800 @ 2.93GHz system with 8GB RAM. For presentation purposes, we ordered the models in ascending order of their (model and target) size measured in lines of code where smaller models are on the left and larger ones are on the right side of each diagram. For our experiments we used valid test models (cf. Figure 7.2.4a) from the railway project which served as regression tests during the project.

The tests showed that the June compiler version has major problems when it comes to larger models.

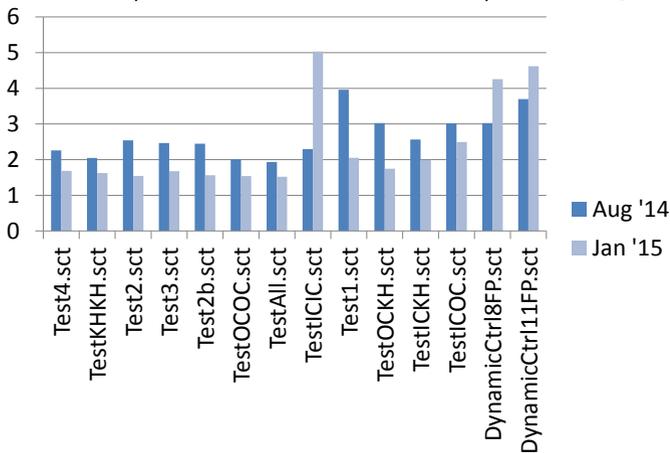
Figure 7.2.4b shows the normalized compile-time for all tested compiler versions. It reveals an interesting point. That is, the relative compile-time per lines of code of the August and January compiler versions are effectively (bounded by a) constant. It still increases for the June compiler version. It may be inferred that the compile-time of the June compiler version increases more than linearly with the model size. This hypothesis was confirmed by code inspection where code parts in the June compiler version, which had exponential complexity, were discovered and could be removed during the project.

Note that the regression test models from the model railway project have a maximum size of 50,000 lines of C code where the eleven train model railway controller in contrast had a size of roughly 400,000 lines of C code and the eight train model railway controller still had a size of about 300,000 lines of generated C code. The tests showed that the August and the January version were able to compile the eleven train controller in about

7.2. SCCharts Model Railway Project



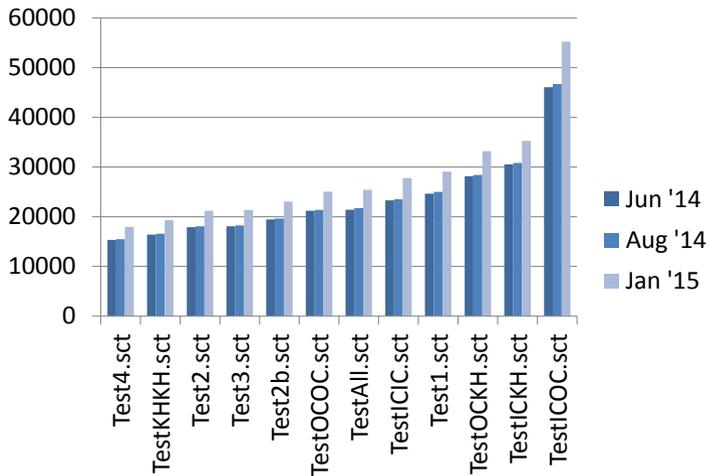
(a) Normalized compile-time for all three evaluated compiler versions [10*ms/loc]



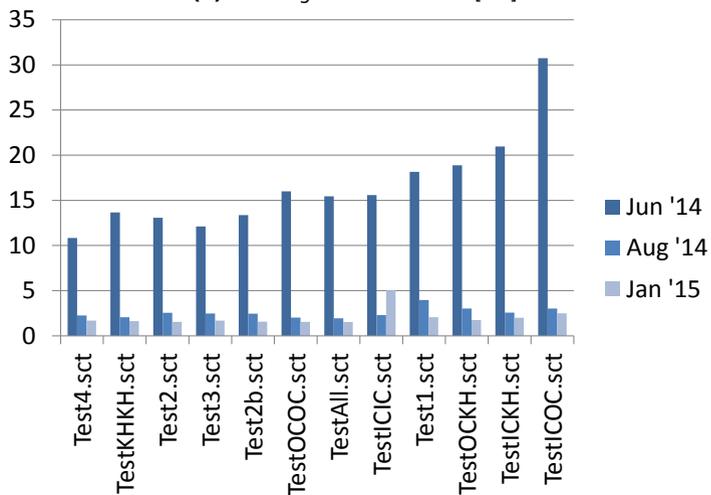
(b) Normalized compile-time for June and January compiler versions [10*ms/loc]

Figure 7.2.3. Performance of SCCharts SLIC-based compiler for regression test models and for the eight and eleven train controller of the model railway project (from [SMSR⁺ 15])

7. Practicality & Validation: The Model Railway Demonstrator



(a) Size of generated C code [loc]



(b) Normalized compile-time [10*ms/loc]

Figure 7.2.4. Performance of SCCharts SLIC-based compiler for regression test models of the model railway project (from [SMSR⁺15])

120 seconds, were the June version of the compiler needed 22,680 seconds due to the unoptimized code parts with exponential complexity. This is visualized in Figure 7.2.4b where the eight and eleven train controllers are compared to the regression test models from Figure 7.2.4. Figure 7.2.3a shows the normalized compile-time results for all three considered versions of the compiler where Figure 7.2.3b only compared the June and the January version where the code parts with exponential complexity have been removed.

Figure 7.2.3a reveals that the January compiler version behaves a bit worse w. r. t. compactness than the August compiler version. The main reasons for that are additional guards and variables for conditionals in the low-level compilation that were introduced to aid the debugging process. These extra variables do not alter the semantics of the generated code. Further versions of the compiler should clean-up redundant artifacts using standard compiler techniques, i. e., *copy propagation* [Bus16].

The tests and the compiler enhancements conducted during the model railway project and afterwards support the hypothesis that the SLIC compiler approach is practically usable and advantageous when compiler-maintainability is important, as it is in the context of safety-critical systems.

7.3 SCCharts Survey

After the project, we asked the students about their experience with SCCharts and the KIELER SCCharts tooling. The students had a background of a synchronous languages class. Hence, they already had used Esterel and SyncCharts. Also, the students are at least in their 3rd year such that they had also worked with C, Java, Haskell, and partly also with Ptolemy.

It is noteworthy that the number of seven participants for this first survey is quite small. In the future there are plans to continue this survey series to further underpin the message of the results. Hence, the current results should only serve as a first indication for evidence to the answers of the asked questions.

7. Practicality & Validation: The Model Railway Demonstrator

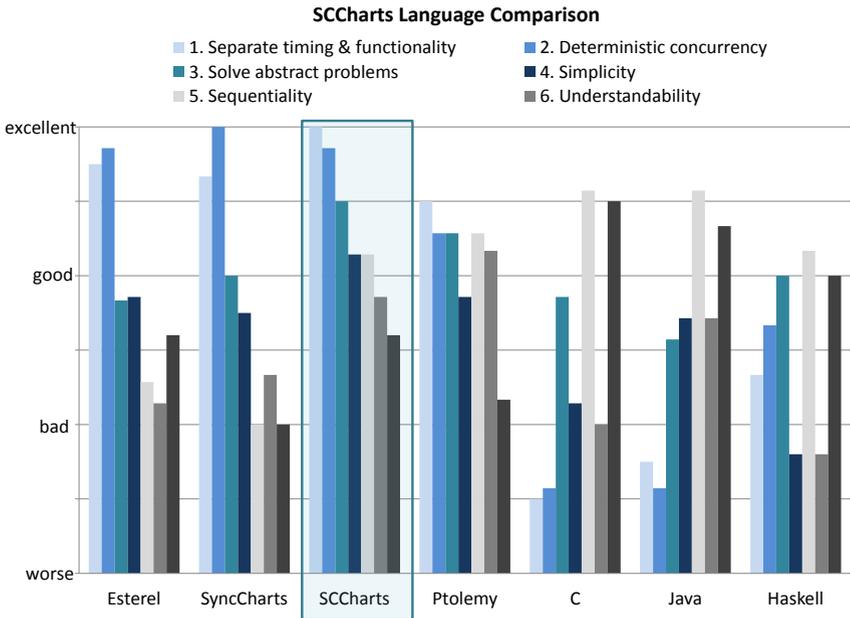


Figure 7.3.1. The SCCharts language compared to other languages (adapted from [SMSR⁺15])

7.3.1 SCCharts vs. Other Languages

Figure 7.3.1 shows the survey results for comparing SCCharts to other synchronous languages such as Esterel and SyncCharts, to the general purpose modeling language Ptolemy, to general purpose programming languages such as C and Java, and to a major functional language representative, Haskell. SCCharts and the other languages were rated according to the following set of criteria:

Separate timing & functionality: This expresses whether the correct function of a task implemented in the language, when executed, is independent of the actual timing of the host system.

Deterministic concurrency: This expresses how easy it is to achieve strict deterministic behavior in especially concurrent parts of a program written in the language.

Solve abstract problems: This expresses how easy it is to obtain a solution for a problem in the language on a high abstraction level, not bothering hardware or implementation details.

Simplicity: This expresses how easy it is to learn the language in order to be able to fully utilize its features to build comprehensive programs. It plays a key role for working on larger team projects.

Sequentiality: This expresses how easy it is to express sequential behavior in the language. For safety-critical systems this is relevant because often sequential behavior is described within separate (often concurrent) tasks. Also, embedded programmers often have an imperative programming background with languages such as C where sequentiality is naturally inherent.

Understandability: This expresses how easy it is to read and understand a program written in the language where the program may be written by the reader itself or by some other person. E. g., understandability reveals how easy it is to get an overview of large projects. It also plays a key role for working on larger team projects.

Solve low-level problems: This expresses how easy it is to obtain a solution for very low-level, hardware-related problems in the language.

The evaluation in total shows that for the first four criteria, separating timing & functionality, achieving deterministic concurrency, solving abstract problems, and simplicity, SCCharts got top ratings. For sequentiality, SCCharts are quite comparable to non-synchronous languages and better rated than other synchronous languages. For understandability, SCCharts and Ptolemy outshine the other languages. For the last criterion, solving low-level problems, SCCharts at least do not get bad or worse ratings. The following paragraphs pin-point some details on the given ratings.

Separate timing & functionality: For safety-critical systems it is mandatory that they function correctly, independent of the exact timing of the

7. Practicality & Validation: The Model Railway Demonstrator

system. It is only necessary to prove that the hardware in the end can compute a reaction (tick) in time which can be asserted using worst case reaction time analysis [BTvH08]. As synchronous languages are inherently designed to separate timing and functionality, they handle this task naturally well. On the contrary, classical programming languages often heavily depend on the implementation with respect to timing and are restricted to specific systems. This is also reflected in the results shown in Figure 7.3.1. Overall, SCCharts is the only language, from the set of given languages, that is rated to be well usable for expressing sequentiality and at the same time still to separate timing and functionality.

Deterministic concurrency: As synchronous programs inherently react deterministically, this especially is true for concurrent parts of a program. Classical sequential programming languages such as Java or C usually build upon a thread concept to support concurrency. Threads introduce high potential for non-determinism. To get a handle on this non-determinism, one needs synchronization mechanisms such as semaphores or monitors [And00]. However, this makes programs hard to read and is very error prone as Lee [Lee06] describes in detail. Hence, expectedly, the participants rated all synchronous languages including SCCharts and also Ptolemy clearly ahead of the other programming languages.

Solve abstract problems: As synchronous languages are designed to specify tasks on a high abstraction level, they naturally score in this category, where in contrast lower-level programming languages come up short. Also, SCCharts as a synchronous language and a modeling language got high ratings from the participants.

Simplicity: According to Figure 7.3.1, SCCharts got top grades for simplicity. Even though SCCharts comprises numerous of extended features, each feature is based on a small set of core constructs. It follows that it is quite easy to first learn only about the small set of core constructs and then widen the view and subsequently learn about extended features that build upon each other. Additionally, the stepwise compilation tool chain of

the KIELER compiler aids the modeler to reconstruct complex features and also inspect language features in detail for certain usage of these features in models. This is even more supported by being able to simulate intermediate models in order to inspect also the dynamic behavior of certain language features.

Sequentiality: Naturally, this is a strength of sequential programming languages. It is also a common drawback for synchronous programs where programmers often tend to run into causality issues when trying to express sequential variable value changes within one reaction computation. To overcome this common drawback, one advantage of SCCharts over classical synchronous languages is the combination of the synchronous MoC with the imperative sequential programming paradigm. The results of the survey show that the participants also rated SCCharts nearly head to head with traditional sequential programming languages and distinctively better than other synchronous languages. This validates the effort in enhancing the synchronous constructive MoC with sequentiality.

Understandability: In this category, SCCharts performs better than other synchronous and classical imperative languages. Only Ptolemy got a better rating than SCCharts. Ptolemy has a very clear and lean graphical syntax which is very intuitive because of its data-flow nature. Also, interfaces are mostly visible in Ptolemy which emphasizes the communication between model components. It is a quite good indication that SCCharts was rated comparable to Ptolemy w. r. t. the understandability aspect. A benefit of SCCharts that facilitates understandability might be that SCCharts have a clear textual syntax but still a graphical representation. For some scenarios the textual and for others the graphical representation may be more convenient. On the one hand for tasks such as copy and paste editing, merging models, parsing expressions, or declarations one may favor a textual representation. On the other hand for getting an overview of the structure, for browsing a model, or for comparing different versions of a model the graphical representation may be more convenient. The possibility of switching between both representations enhances the understandability of *textual*

7. Practicality & Validation: The Model Railway Demonstrator

modeling that has been chosen for the KIELER SCCharts implementation (see Section 6.4.4 on page 327).

Solve low-level problems: Contrarily to solving abstract problems, solving low-level problems is more the domain of lower-level programming languages than it is for higher-level modeling languages. Thus, Ptolemy or the synchronous languages in general must accept worse ratings in this field. Nevertheless, SCCharts is capable to handle low-level tasks with an implemented host code feature that is also capable of host function calls. However, we still advise to consider a clean separation between low-level code parts and a high-level model, see Section 3.3.4 on page 79. Altogether, this leaves potential for future improvements but SCCharts still do get at least midrange ratings for this last category.

Related Projects

This chapter discusses other synchronous modeling languages, tools, compilers, and work that is related to this thesis. The work presented here served to integrate these languages and/or their compilers into KIELER in order to improve the KIELER SCCharts tooling and compiler.

8.1 Ptolemy and KIELER

8.1.1 Ptolemy

Ptolemy [EJL⁺03, Lee03] is a framework that supports heterogeneous modeling, simulation, and design of concurrent systems as introduced in Section 6.1.1 on page 303. Ptolemy has heavily influenced the development of SCCharts on the one hand, but on the other hand the development of SCCharts also had some influence on Ptolemy. The purpose of this chapter is to explain the connections in more detail. In order to do that, this chapter introduces Ptolemy briefly. A more comprehensive introduction to Ptolemy can be found on the Ptolemy website¹ or in the Ptolemy book [Pto14].

Heterogeneous Actor Oriented Modeling

Actor oriented modeling [LNW03] is a term coined by the Ptolemy group headed by Prof. Edward Lee at the UC Berkeley. In this spirit, models are formed by connected and interacting *actors* as key components. Figure 8.1.1 shows a Ptolemy model for generating a sine wave. It consists of several actors like Ramp or AddSubtract that are connected by their ports. An actor

¹<http://www.ptolemy.org>

8. Related Projects

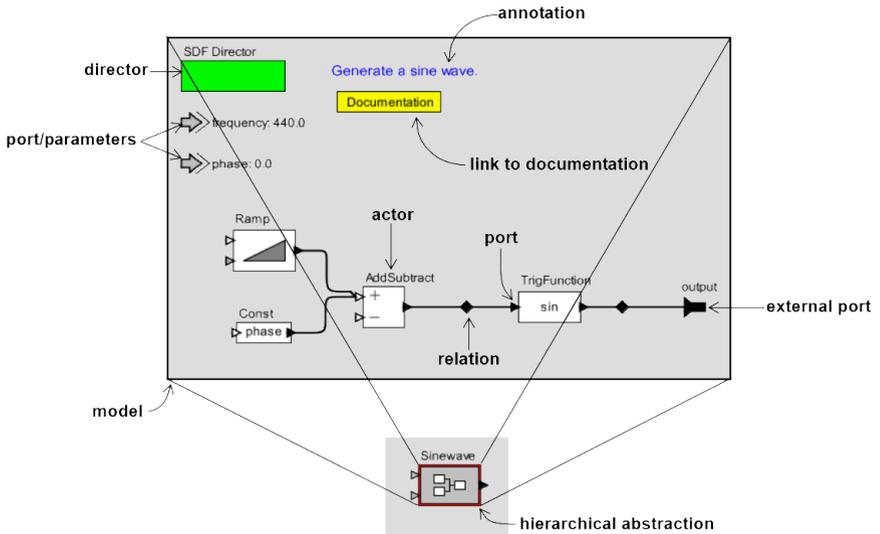


Figure 8.1.1. Ptolemy actor model example [BLL⁺08]

is a component with input and output ports of specific types. Each actor can be *fired*. It then computes a reaction, i. e., output tokens considering input tokens and a possible internal state. Actors can be Ptolemy models itself or they can be special Java classes that implement the `Actor` interface. For example, the hierarchical Ptolemy model shown in Figure 8.1.1 is a sine wave actor itself. It consists of more basic actors such as the `AddSubtract` actor, which is a Java class implementing the `Actor` interface. The firing is done by a *director* which is another central component that each Ptolemy model contains. It is recognizable by its green color. The model of Figure 8.1.1 has an SDF Director. A director defines in which order and how often actors of a model (on the same hierarchy level) are fired. There can only be one director per hierarchy level. A director is typically used to implement a certain MoC.

Models of Computation (MoC)

A model of computation defines the dynamics of a Ptolemy model. Hence, it defines how and when actors of a Ptolemy model compute and interact with each other. A director is used to implement a certain MoC. At run-time, the director decides when to invoke computation or communication actions. Computation events are calls to the `fire()` method of actors while communication events are calls to the actor's `Receivers` which can be specific for a certain MoC. Actors may also be specific for use with certain MoCs only. However, there also may be other actors that can be used with arbitrary MoCs.

Examples for the most common MoCs are given in the following. A more detailed and profound description of the most common MoCs used for embedded systems can be found elsewhere [LS11].

Continuous Time (CT): The interaction between actors is done using continuous time signals. Actors implement differential relations between their inputs and outputs. An execution of a CT model is to solve the differential equations using differential equation solvers.

Discrete Events (DE): The interaction between actors is done using sequences of events. An event is defined to be a pair of a time stamp and a value. Upon firing, actors react to their input events with producing events on their outputs.

Process Networks (PN): The interaction between actors is realized as asynchronous message passing. The channels used for communication are buffered. Upon firing, actors react to an incoming message with producing messages on their outputs. The PN MoC is an implementation of Kahn Process Networks [Kah74].

Synchronous Data-flow (SDF): This MoC is a more specific PN MoC where the number of messages per firing is fixed.

Synchronous Reactive (SR): The Synchronous Reactive MoC enables the modeling of reactive systems in Ptolemy. Synchronous Reactive (SR) systems establish determinism for concurrent applications such as embedded control systems. The interaction between actors is done by establishing values for every channel. Tokens represent present values where clearing a channel represents the absence of a value.

8. Related Projects

As common in synchronous languages, also in the SR MoC, signals can be present or absent. Signals are represented as tokens on a data link between concurrent actors in Ptolemy. In SR, the time proceeds tick-wise where each tick is a reaction computation as common in the synchronous programming paradigm. In such a reaction computation, the task is to establish a fixed-point for every link between concurrent actors telling whether there is a token or not at the end of the tick computation and also telling about the (possibly combined) value. This tick computation is equivalent to what is shown in Figure 2.1.1 on page 20. Likewise, the fixed-point computation needs to end after a finite amount of computational steps. Hence, an actor in SR may be fired to do computations multiple times but only a finite number of times for each tick computation. An important property is that in the SR MoC, actors can react to the absence of a signal where the absence itself is defined deterministically by the model. If there is no unique fixed-point, i. e., a deterministic unambiguous outcome for each signal to be present or absence after each tick computation, this reveals a causality problem of the model. Ptolemy rejects such models as faulty.

Finite State Machines (FSM): Entities here are special actors that represent states. Connections between these states are interpreted as transitions. An execution is a strictly ordered sequence of taken state transitions. The built-in expression language is used to evaluate guards and set outputs. States are allowed to have a refinement which can be another FSM again.

Typically, in reactive systems, the output for one reaction is computed based 1. on the inputs and 2. on an internal state of the system (cf. Figure 1.0.2 on page 3). A state of a program can be any kind of information held in a register, e. g., variable values. For a Ptolemy actor, a state may be encoded in a class member variable of the specific actor instance. In a reactive system, a state is typically used to express a system state of the physical controlled system, e. g., On or Off. Many reactive systems involve heavy control-flow oriented tasks with computations on such states and state transitions. The FSM MoC reflects these states directly in the model. For example, Figure 8.1.3 shows two FSM actors that

are used to represent two concurrent regions of an SCChart, R1 and R2 (details are given in Section 8.1.2). Inputs of FSM actors can be used throughout the whole state machine, e. g., within transition triggers as guard: L_i isPresent. Outputs are also accessible from all transition output actions as output: $L_o=1$. FSMs can be hierarchical [LL98] meaning that states can contain state refinements which again are FSMs. Other types of refinements are not allowed for FSMs.

Modal Models: This MoC generalizes FSMs. State refinements can be arbitrary Ptolemy models, each with a different MoC. Modal models [Lee09] are meant to reflect systems with different modes of operation. Mode changes are represented as transitions, triggered by external or internal events. Typically, there is one refinement for each mode. However, it is also possible for two distinct modes to share one refinement. Further, it is possible for a mode to have multiple (e. g., concurrent) refinements.

Model Time: There is a notion of time supported by Ptolemy and some MoCs. That is, every two communication or computation actions can be strictly ordered or are simultaneous. Modal models support the hierarchical combination of MoCs that have different notions of time.

Example: Figure 8.1.2 shows a Ptolemy model where timed SR and untimed modal model MoCs are interleaved. It implements a traffic light controller that has inputs Sec, Error, and Ok at the top level. Pgo_o and Pstop_o are fed back as Pgo_i and Pstop_i, respectively. Internally, they are used for communication between the concurrent CarLight and PedestrianLight controller. The CarLight controller is the main controller which takes Sec as an input and produces Pstop_o whenever pedestrians should not cross because cars have a green signal. It produces Pgo_o whenever pedestrians are allowed to safely cross because cars have a red signal. The behavior is cyclic and alternating between modes green, yellow, red, yellow, and then green again for the car light. Such mode changes are triggered externally by Sec. This is the behavior in the Normal mode. In the Error mode, there is no such communication but the pedestrians have a permanent red light and the car's light is blinking yellow.

8. Related Projects

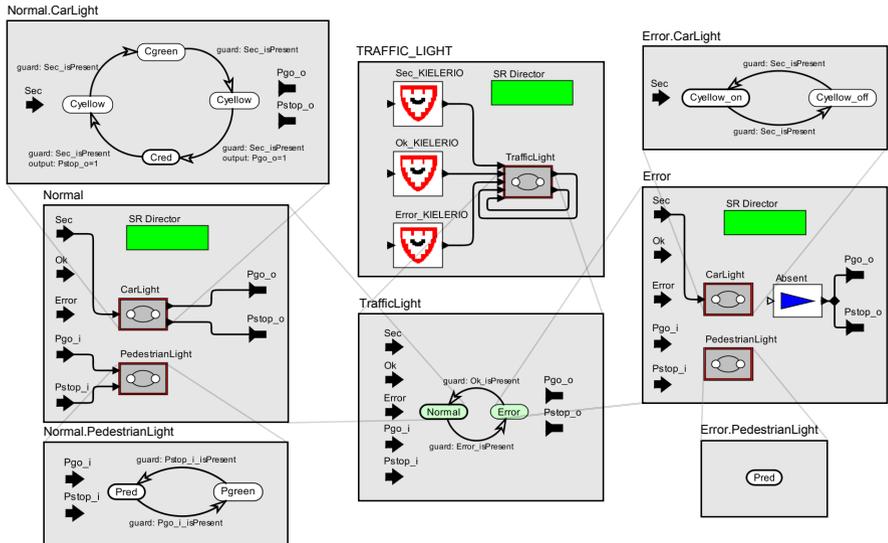


Figure 8.1.2. A hierarchical and heterogeneous Ptolemy model with alternating modal models and SR MoCs [vHLMF12b]

8.1.2 KIELER leveraging Ptolemy (KlePto)

The hierarchical alternating combination of the SR and the modal model domain enables to express complex synchronous control systems in Ptolemy as described earlier [MFvHL12, MFvH10] for SyncCharts, the predecessor of SCCharts. That work studies how to automatically synthesize a Ptolemy model from a SyncChart model in the spirit of Figure 8.1.2 and Figure 8.1.3. Note that SCCharts borrow most of their syntax from its predecessor SyncCharts and that any valid SyncChart is a valid SCChart with the same semantics modulo different design decisions (e. g., actions and aborts). Hence, for this Berry-constructive subclass of SCCharts it is possible to re-use the synthesis of Ptolemy models as described in the following. It further would be interesting to explore how to fully express all valid sequentially constructive SCCharts in Ptolemy, e. g., by using the Coroutine MoC [SL12].

In the following section, more details about the SyncCharts to Ptolemy transformation are given. After that, possible optimizations for the transformation are sketched. A section about the KIELER integration of the Ptolemy simulator that uses this transformation follows. The section about Ptolemy and KIELER is completed by some notes on extending Ptolemy. These extensions were motivated by observations learned while trying to fully support all SyncCharts features in the Ptolemy synthesis.

SyncCharts to Ptolemy Transformation

The goal of synthesizing runnable Ptolemy models from arbitrary SyncCharts models is to give semantics to SyncCharts by defining a SyncCharts to Ptolemy model transformation.

The model transformation maps each SyncCharts element to one or more Ptolemy actors. It utilizes the hierarchical combination of SR and modal models. Each modal model represents a concurrent SyncCharts region. Explicit broadcast of signals is incorporated by appropriate ports at the modal model. These are added for all signals which can be read or written to in the scope of a SyncCharts region. The ports are connected appropriately.

The SR fixed-point semantics guarantees finding a fixed-point for a signal status assignment w. r. t. the signal coherence rule. A present signal status in Ptolemy is represented as an existing token on a connection link. An absent signal status in Ptolemy is represented as a non-existing token, a *clear* operation on a connection link.

For each synchronous tick, the fixed-point computation in SR starts with unknown signal statuses on all data links which must be known, i. e., either present or absent, by the time the fixed-point is found. If any link is still unknown, the original SyncCharts model is considered to be non-constructive.

Example: Figure 8.1.3 shows an SCCharts model with two concurrent regions R1 and R2 that communicate with each other using a local signal L. Region R1 contains two states, S0 and S1, where S0 is the initial state. A state transition from S0 to S1 with an implicit true trigger has an output

8. Related Projects

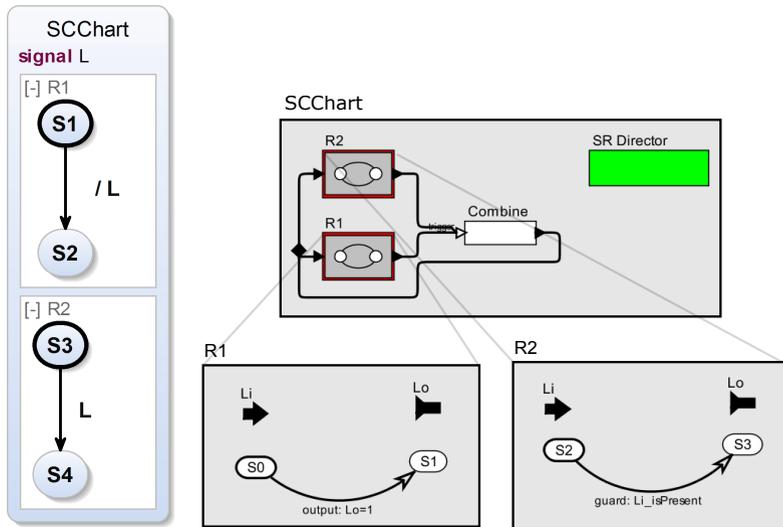


Figure 8.1.3. An SCChart (left) and an automatically generated Ptolemy model (right). It uses the SR director to implement explicit instantaneous communication between the concurrent parts R1 and R2 (adapted from [MFvHL12]).

action which emits L. This means it makes L present for the tick in which this transition is taken. Region R2 has two other states, S2 and S3, where S2 is the initial state. A state transition from S2 to S3 is triggered by L. This means that the transition is taken if the region R2 is in state S2 and L becomes present in a tick. The Ptolemy model contains two modal model actors which represent both concurrent SCCharts regions. The implicit broadcast of concurrent SCCharts regions is represented by explicit links between the concurrent modal model actors in the Ptolemy model.

The structural transformation ensures that for each concurrent region, every signal is represented as an input and output port (e. g., Li and Lo) because conceptually, each region could emit a signal or may react to a

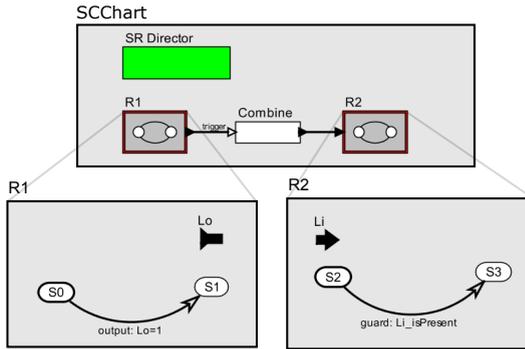


Figure 8.1.4. A Ptolemy model generated from the same SCChart than the Ptolemy model in Figure 8.1.3 but with optimized inputs and outputs (adapted from [MFvHL12])

present or absent signal, or even both. The latter implies the requirement of a feedback structure for each signal using a special combine actor. If the combine actor receives a token of any concurrent region, it immediately outputs a token to the feedback loop. However, if the combine actor notices a clear on each connected incoming channel, it also clears its output.

A mapping annotation of created Ptolemy elements ensures that signals and states can be mapped back to the original SyncCharts elements for visualization purposes or in order to set/inspect inputs/outputs of the model. This is the duty of the overall Ptolemy simulator integration which is explained on page 391.

Transformation Optimizations

Input/Output Signals: To allow the emission of input signals or to test for the presence of output signals, the transformation is modified to add additional auxiliary SCCharts regions for each signal. These get transformed into auxiliary modal models at the top level of the Ptolemy model. In case there are no emitted input signals or tested output signals, these auxiliary regions are superfluous and are eliminated.

8. Related Projects

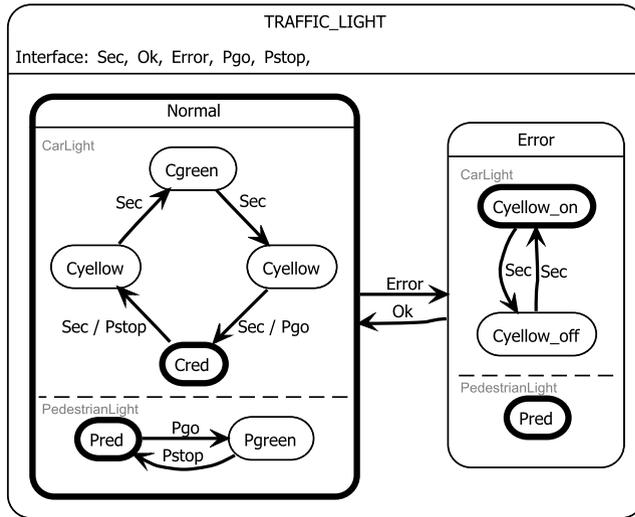


Figure 8.1.5. Traffic Light SyncChart that KlePto is able to automatically transform into a Ptolemy model as shown in Figure 8.1.2 (from [vHLMF12b])

Actor Signature and Feedback Loop: In the Ptolemy model of Figure 8.1.3, the input L_i is available to the actor R_1 although this actor does not use this input. In the general case, this input must be available to the actor that implements the SCCharts region R_1 . However, in case inputs are not used, they unnecessarily add complexity to the generated Ptolemy model. We extended the transformation to optimize the Ptolemy model in such way that superfluous inputs are removed. We also did this optimization for output signals. This results in a generally clearer actor structure with minimal actor interfaces and data-flow channels. Also, unnecessary feedback loops are removed automatically by this procedure. Figure 8.1.4 shows an optimized version of the Ptolemy model where superfluous inputs and outputs of L were removed. Also consider a more sophisticated SyncChart, the traffic light example shown in Figure 8.1.5 together with its

8.1. Ptolemy and KIELER

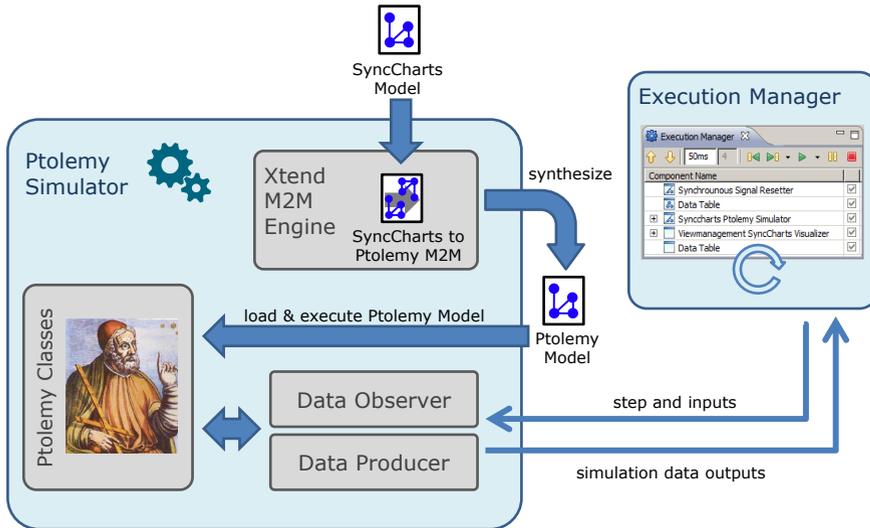


Figure 8.1.6. Transformation and execution scheme of the Ptolemy-based SyncCharts KlePto simulation (adapted from [MFvHL12])

generated corresponding Ptolemy model of Figure 8.1.2. The signals P_{go} and P_{stop} are used to communicate from the Car controller region to the Pedestrian controller region. Since they are defined on the top level in the Ptolemy model, feedback loops are necessary for both signals within the TRAFFIC_LIGHT actor. However, the inputs and outputs of each modal model implementing the region, i. e., CarLight and PedestrianLight, are optimized and restricted to only used signals.

Ptolemy Simulator

Figure 8.1.6 shows the underlying concept where the description of the transformation is defined in the file `SyncCharts to Ptolemy M2M`. For KIELER this model transformation is implemented using the *Xtend* language. The Ptolemy Simulator takes a SyncCharts Model as an input and applies the transformation to it. This way, a semantically equivalent Ptolemy Model

8. Related Projects

is created. The model is instantiated by Ptolemy's Modeling Markup Language (MOML) parser as part of the Ptolemy Classes. After the model is instantiated, it can be stimulated with inputs and stepwise executed. A specialized DataObserver is used to transfer inputs to the Ptolemy model and a specialized DataProducer is used to process the outputs of the executing Ptolemy model. Both are part of the KIELER simulation infrastructure, i. e., the *KIELER Execution Manager* (see Section 6.3 on page 316). Its GUI makes the input and outputs accessible for the user of the KIELER tool. It further completely hides Ptolemy which operates in the background.

Conceptually, this Ptolemy simulator works for any Domain-Specific Language (DSL) and not only for SyncCharts. Modifications to the transformation and to the mapping could become necessary, but in principle the Ptolemy simulator can be considered a generic simulator for KIELER.

Extensions to Ptolemy

As shown earlier in Section 8.1.2, the hierarchical alternating combination of the SR and the modal model domain allows to express complex synchronous control systems such as SyncCharts. This combination led to specific enhancements to the absence resolution logic and immediate transitions implemented in the FSM domain (*FSMACTOR*), which is the basis also for modal models. These enhancements are fully described in the Ptolemy book [Pto14].

Initially, the SyncCharts to Ptolemy transformation was implemented for main SyncCharts concepts only. Modal models for example did not allow to represent immediate transitions yet. These are transitions that can be taken immediately, i. e., in the same tick, when their source state is entered. In SCCharts, these transitions are denoted by a dashed transition.

As explained earlier, whenever predefined Ptolemy domains, i. e., predefined directors, seem to limit the expressiveness of Ptolemy, one can freely choose to 1. add special actors to the Ptolemy framework, 2. combine existing Ptolemy domains and actors, or 3. design a derived or even completely different director. In the special case of immediate transitions, we even proposed to enhance the Ptolemy modal models themselves. We helped to apply this feature to the modal model domain with the advantage of

8.2. KIELER Esterel Integration

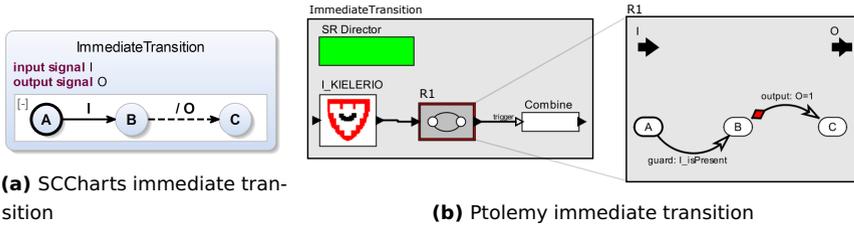


Figure 8.1.7. New immediate transitions in Ptolemy for simulating SyncCharts/SCCharts immediate transitions and states acting as transient SCCharts connector nodes. Both transitions are taken in the same synchronous tick when the input signal I becomes present. Hence, the output signal O is emitted instantaneously (adapted from [MFvHL12]).

having it available for expressing SyncCharts immediate transitions in most cases. Immediate transitions can also be attractive to a modeling tool like Ptolemy because they allow to structure combinatorial chains of transitions following the WTO principle. Fig 8.1.7 shows both an SCChart with an immediate transition that is taken immediately when the state B is entered and a corresponding Ptolemy model. The Ptolemy modal models now support immediate transitions which are denoted by a red diamond. The color indicates that they are always preemptive.

8.2 KIELER Esterel Integration

Esterel is a synchronous language similar to SCCharts. It is the textual counterpart to SyncCharts, which can be seen as the predecessor of SCCharts. Esterel is integrated in KIELER by means of a fully-featured textual Esterel editor, a compiler integration for simulating Esterel programs, and a simulation visualization. Esterel is used for generating ESO trace files to validate the SCCharts compiler and for studying and teaching synchronous languages. In particular, a sequentially constructive extension to Esterel termed SCEst can be studied using the KIELER Esterel infrastructure. This section will discuss Esterel and the KIELER Esterel infrastructure.

8. Related Projects

```
1 module ALDO:
2 input A;
3 output D;
4 output O;
5
6 signal L in
7 [
8     await A;
9     emit L;
10    ||
11    abort
12        pause;
13        sustain D;
14    when immediate L;
15    emit O;
16    halt;
17 ];
18 end signal;
19 end module
```

Listing 8.2.1. ALDO example as an Esterel program

8.2.1 Esterel

Esterel [Ber00b] is a textual synchronous control-flow oriented language. It is based on the Synchrony Hypothesis separating the design of logical control-flow from any timing constraints of a program. It has a formal semantics [Ber02]. It is specially designed for software but also for hardware [Ber91, BK00] development. There are also specialized processors [LvH06, YAY⁺08] that are designed to execute Esterel code natively.

Listing 8.2.1 shows ALDO (cf. Section 3.1 on page 48) as an Esterel program. Recall that ALDO has one input signal *A*, a local signal *L*, and two output signals *D* and *O*. ALDO consists of two concurrent threads. The first thread waits for the input signal *A* and then emits *L*. The second thread concurrently waits for *L* to become present and, while waiting, it continuously emits *D*. Whenever *A* becomes present — other than in the initial tick — *L* is emitted by the first thread and the second concurrent thread immediately reacts to *L* by emitting *O*. Furthermore, it will not emit *D* in that tick because the `sustain` statement is (strongly) aborted. Then, the program halts in the second thread while the first thread terminates after emitting *L*.

8.2. KIELER Esterel Integration

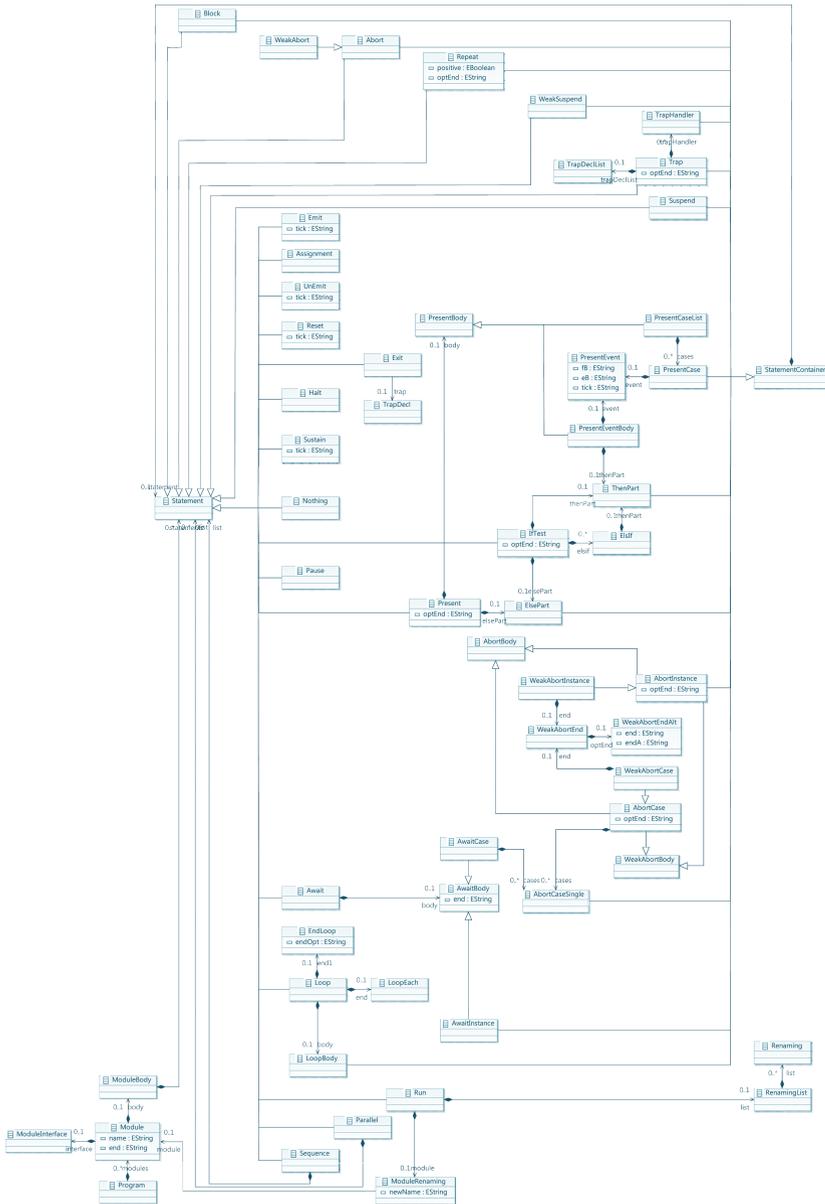


Figure 8.2.1. Eclipse EMF Esterel meta model (simplified)

8. Related Projects

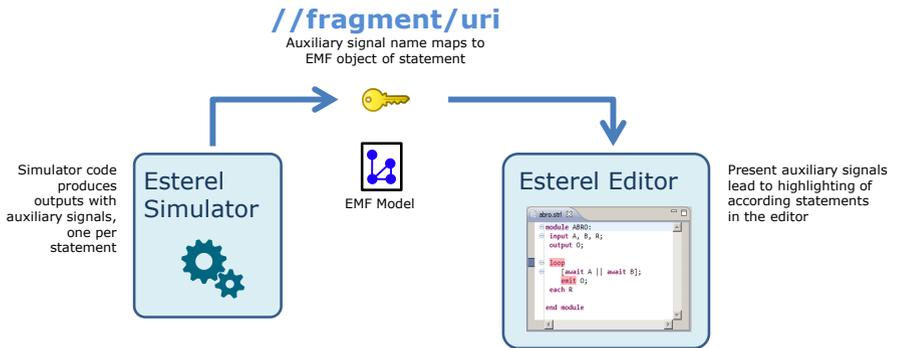


Figure 8.2.2. External compiler integration and simulation visualization concept (auxiliary signal mapping only)

8.2.2 Meta Model

The textual Esterel editor, shown later in Figure 8.2.5, is based on the Xtext framework. Xtext allows to specify a language grammar together with concrete syntax elements and to automatically generate a fully-featured Eclipse textual editor with syntax highlighting and code completion. Xtext is backed by the Eclipse Modeling Framework (EMF) and the generated editor also comes along with a generated parser and serializer for a meta model that covers the language as specified by the grammar.

Regarding the Esterel meta model, we choose to have it also be generated by Xtext. The meta model for Esterel is shown in Figure 8.2.1 in a slightly simplified version due to space limitations and readability aspects.

Any Esterel source-to-source or other model transformation can be defined upon this meta model.

Examples for such model transformations on this Esterel meta model are the generic Esterel simulation visualization (cf. Listing 8.2.2) presented in the following or the Esterel to SyncCharts transformation [RMvH11] (cf. Figure 8.3.1 on page 406).

8.2. KIELER Esterel Integration

```
1 output auxActiveP;
2
3 signal AP
4 in
5 [
6   P; emit AP
7   ||
8   weak abort
9   sustain auxActiveP
10  when immediate AP
11 ]
12 end signal
```

Listing (8.2.2) Esterel source-to-source visualization transformation for a statement P

```
1 module IO:
2 input I;
3 output O;
4
5 await I;
6 emit O;
7 end module
```

Listing (8.2.3) Simple Esterel IO.strl example before simulation visualization transformation

```
1 module IO:
2 input I;
3 output O;
4 output aux1493174180;
5 output aux1493174181;
6 [
7   [
8     signal AP1 in
9     [
10      await I;
11      emit AP1
12      ||
13      weak abort
14      sustain aux1493174180
15      when immediate AP1
16    ]
17    end
18  ];
19  [
20    signal AP2 in
21    [
22      emit O;
23      emit AP2
24      ||
25      weak abort
26      sustain aux1493174181
27      when immediate AP2
28    ]
29    end
30  ]
31 ]
32 end module
```

Listing (8.2.4) Transformed version IO.simviz.strl after applying the visualization transformation to IO.strl

8.2.3 Simulation and Debugging

As described earlier, the Xtext framework is used to generate a fully-featured textual Esterel editor that is backed by an adequate meta model as well as automatic parsing and serialization.

Esterel programs can be modified by using model transformations. They can be executed by compiling them to executable binary code, which can be run in a separate thread using the KIELER Execution Manager

8. Related Projects

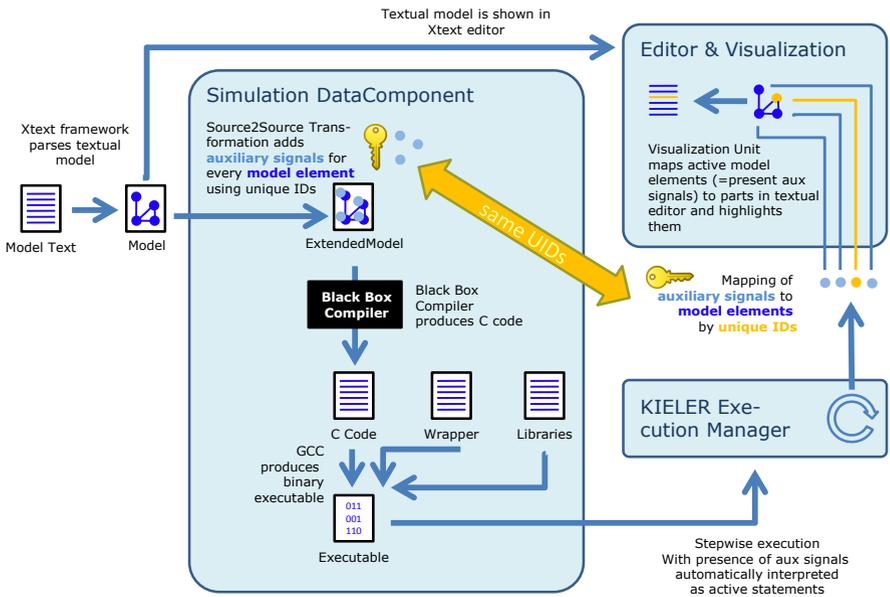
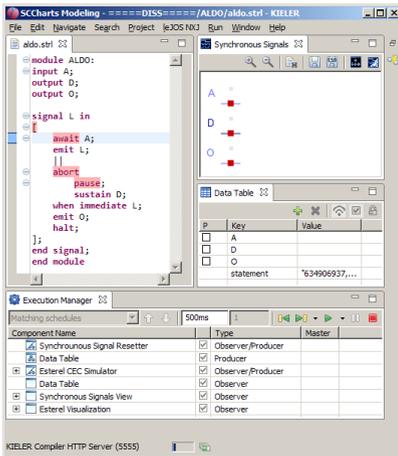


Figure 8.2.4. External (black box) compiler integration and simulation visualization concept (full picture). The Source2Source Transformation could for example be the Esterel visualization transformation shown in Listing 8.2.2.

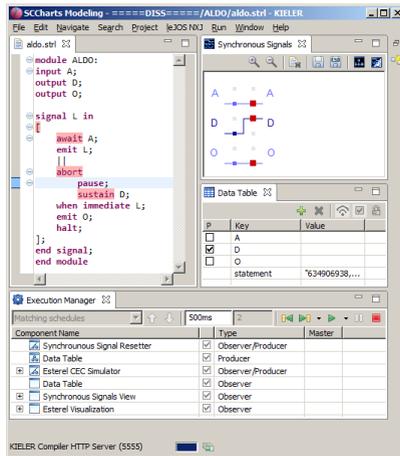
infrastructure (see Section 6.3.1 on page 316) and GUI elements as the Data Table for letting the user stimulate the program with inputs and for letting the user inspect outputs.

The Execution Manager is shown in the lower part of the screenshots in Figure 8.2.5. It allows to stepwise execute the so called *DataComponents* listed in its schedule. A *DataComponent* can be a simulator such as the Esterel CEC Simulator that takes the current Esterel program, compiles, and executes it. Other examples are the Data Table for user input/output or a visualization component like the Esterel Visualization for visualizing active statements in the editor by highlighting them.

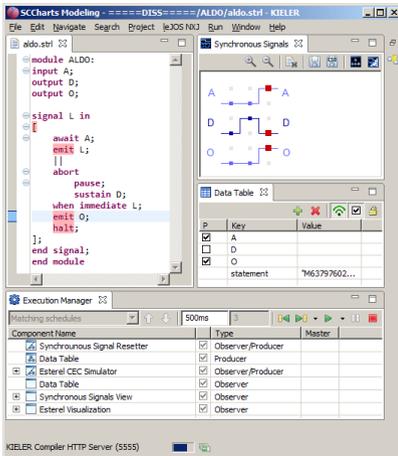
8.2. KIELER Esterel Integration



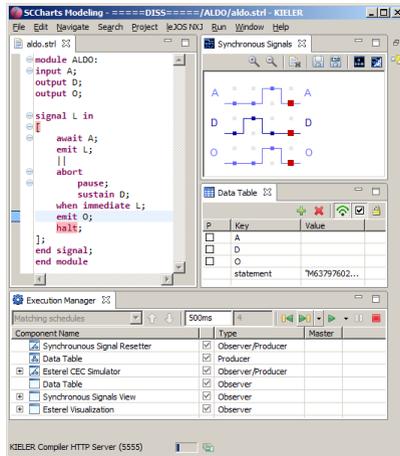
(a) Tick 0



(b) Tick 1



(c) Tick 2



(d) Tick 3

Figure 8.2.5. The KIELER Esterel simulator with visualization is running ALDO by leveraging the CEC.

8. Related Projects

Esterel Active Statement Visualization

8.2.1 Definition (Active Esterel Statement). An Esterel statement is considered *active* iff it is executed in a given tick. It is considered *not active* otherwise.

In order to retrieve an adequate simulation visualization, similar to the one of the Esterel v5.92 simulator (cf. Figure 6.1.5 on page 310), active statements should be highlighted in the Esterel editor directly. This is the purpose of the Esterel Visualization DataComponent that is listed in the Execution Manager schedule, visible in the screenshots of Figure 8.2.5.

However, this DataComponent performs the highlighting of specific statements only. It does not compute which of them are active by itself. This information must be provided by an Esterel simulator component. Hence, there are two parts involved for highlighting active statements:

1. Computing active statements and
2. Highlighting these statements.

An overview is given in Figure 8.2.2. The Esterel simulator computes auxiliary output signals that have a specific name which is a unique ID according to exactly one statement. This auxiliary signal is present iff the statement is active. It is absent iff the statement is not active. The highlighting Esterel visualization DataComponent must then search the model of the editor for all statements according to present auxiliary signals from the simulator. It must highlight these active statements. Additionally, it must possibly undo previous highlighting for inactive statements that have been active before. The IDs of statements are computed as so called *fragment URIs* [SBPM09, p. 447].

Computing Active Statements: A more detailed view of how an external compiler can be leveraged from KIELER for simulation purposes is given in Figure 8.2.4. The Esterel CEC and the Esterel v5.92 compiler are both integrated into KIELER according to this approach.

Computing active Esterel statements for a black box is not trivial because most likely there are no adequate hooks that can be used. Even more, such

8.2. KIELER Esterel Integration

hooks would not be standardized across different Esterel compilers and hence there is no generic way to use such hooks. To overcome this situation, the approach is to have a source-to-source transformation which adds auxiliary signals to an Esterel program by not changing the semantics of the program w. r. t. existing signals. For each Esterel statement, an auxiliary signal is added in such way that this auxiliary signal is present iff the statement is active.

This transformation is presented in Listing 8.2.2. It transforms any Esterel statement P into a construct according to the listing. $\text{auxActive}P$ is the additional auxiliary signal for statement P which is added to the Esterel program's interface as an additional output (line 1). The additional output will later be used to highlight the statement P whenever P is active. A local signal scope is added where AP is a local abort signal. AP is used to detect if P has terminated. It is used in a fresh concurrent region that sustains (line 9) the auxiliary signal as long as P is active. The sustain can immediately be aborted (line 10) by the abort signal in case P is instantaneous such that the whole construct also is instantaneous.

Mapping: The auxiliary signal gets a name that consists of a unique ID which is derived from the statement P . This ID can later be used by the visualization unit for mapping it back to the original Esterel statement P .

Scalability: For each statement, a constant number of auxiliary Esterel statements are added to the original code. The code size increases linear to the number of Esterel statements. Additionally, one could further drastically reduce code blow-up by selectively transforming only Esterel statements of interest and not all Esterel statements. The modeler could possibly select these statements of interest before they start debugging their Esterel program.

Generality: The shown transformation works with any Esterel compiler. No special hooks in the compiler or the resulting target code are required. This makes the approach to make use of such a source-to-source transformation generic and re-usable. The same approach is used in KIELER for visualizing simulations of SyncCharts or SCCharts.

Maintainability: It is fully transparent what happens when this transformation is applied because the semantics of Esterel is used for defining *active*

8. Related Projects

statements. One can use the approach itself to visualize it by applying the transformation twice and visualizing the result of the first application. This transparency and the fact that it is applied on a high abstraction level makes the approach understandable and maintainable.

Figure 8.2.4 shows the general approach for making use of such a source-to-source *simulation visualization transformation* for integrating a black box compiler. Source2Source Transformation, e. g., the transformation shown in Listing 8.2.2, is applied to the original Model which is the original Esterel program in the Esterel case. Auxiliary signals (light-blue dots in Figure 8.2.4) are added for each model element (blue dots in Figure 8.2.4) using a derived unique ID as the signal name. The transformed Esterel program can then be passed to any Esterel Compiler, e. g., the CEC. The resulting C code is compiled using a C compiler like the gcc together with wrapper code and libraries that allow to set inputs and read outputs in a cyclic, stepwise fashion. The KIELER Execution Manager is leveraged to stepwise execute the resulting executable on the user's request. Additional outputs of the auxiliary signals are hidden to the user. Instead, they are forwarded to the *Visualization Unit* that is described in the following.

Highlighting Active Statements: The *Visualization Unit* is a separate Execution Manager DataComponent which knows about the active editor and the active model. It takes a list of auxiliary variables that represent active model elements and is capable of performing the actual highlighting in the model editor for active model elements. It also is responsible for un-highlight previously active model elements that became inactive.

The Esterel Visualization DataComponent is part the Execution Manager schedule of Figure 8.2.5. It takes auxiliary Esterel signals as an input where the names must follow a convention. That is, the name of each auxiliary signal must be derived from the fragment URI of the statement it belongs to. This generic visualization DataComponent works with all Esterel simulators. For KIELER, these currently are CEC, Esterel v5.92, and SCEst.

Figure 8.2.4 on its right part shows how the visualization is part of the compiler simulation integration in KIELER. The light-blue dots represent the auxiliary signals that come from the Esterel simulator. These are either

8.2. KIELER Esterel Integration

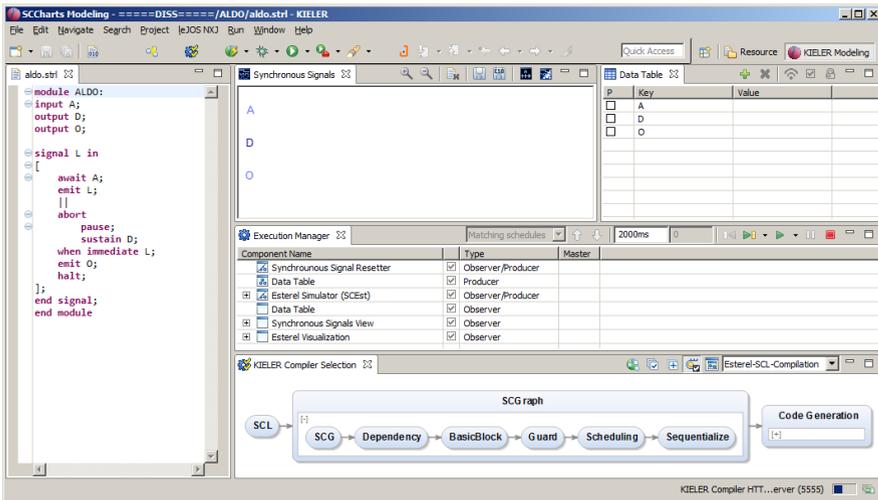


Figure 8.2.6. ALDO simulation with KIELER SCEst compiler integration as a CEC alternative

present or absent. In the initialization phase, the Visualization Unit searches the model for the according Esterel statement for every such auxiliary signal and caches this mapping. Later, it can easily use this mapping to tell which Esterel statements are to highlight and which possibly are to un-highlight during execution.

The editor is then 1. requested for the certain text parts that correspond to the Esterel statements in question and 2. asked to change colors for these text parts.

8.2.4 Sequentially Constructive Esterel (SCEst)

The SCEst compiler is an Esterel compiler that is based on mostly the same principles as the SCCharts compiler that is part of this thesis. Hence, it is briefly described here for comparison. SCEst is a conservative extension to the Esterel language w. r. t. to the sequentially constructive semantics. This implies that all valid Esterel programs have the same meaning, interpreted

8. Related Projects

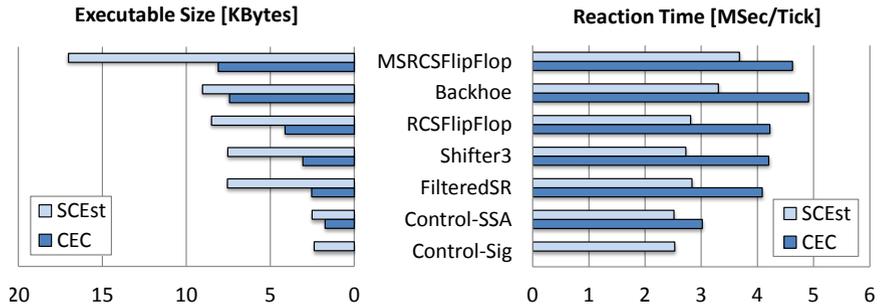


Figure 8.2.7. Size and speed comparison of compilation with SCEst and the CEC. The Control-Sig example is not Berry-constructive and hence rejected by the CEC (from [RSM⁺15]).

or compiled as a SCEst program. Additionally, some sequentially constructive SCEst programs are not valid/Berry-constructive Esterel programs. Details on SCEst were published elsewhere [RSM⁺15].

We integrated the SCEst compiler into the KIELER tool as an alternative to the CEC in order to simulate Esterel programs. Figure 8.2.6 shows the Esterel simulation in the KIELER tool using the SCEst compiler. Note that the Esterel Visualization component in the Execution Manager is re-used as well as the visualization transformation (cf. Listing 8.2.2). In the lower part of Figure 8.2.6, the *KIELER Compiler Selection* shows the white box compilation approach where Esterel code is first translated into a control-flow graph (SCG) and further incrementally compiled to C code. The SCEst compiler is planned to replace the CEC as the default compiler for Esterel in the future. Currently, it is still under development and not yet mature.

Some first evaluation results are shown in Figure 8.2.7. We evaluated our data-flow-based compilation approach. For that, we compared the sizes and reaction times of the generated code from the SCEst with the CEC compiler. The used system was an Intel Core 2 Duo T9800 (2.93GHz) architecture. The reaction times for this evaluation were averaged for 1,000 ticks. These first benchmarks are rather small (29 lines of code for the Control-Sig, 93/171 lines of code before/after module expansion for MSRCSFlipFlop), and we did not

extensively try all the compile options offered by the CEC. Thus, the results should be treated with care.

However, the evaluation showed that, in terms of the reaction time, SCEst is quite competitive. It is about 30% faster on average than the CEC. In terms of the code size the picture looks different: SCEst generates on average about twice as much code than the CEC which leaves much room for future improvements.

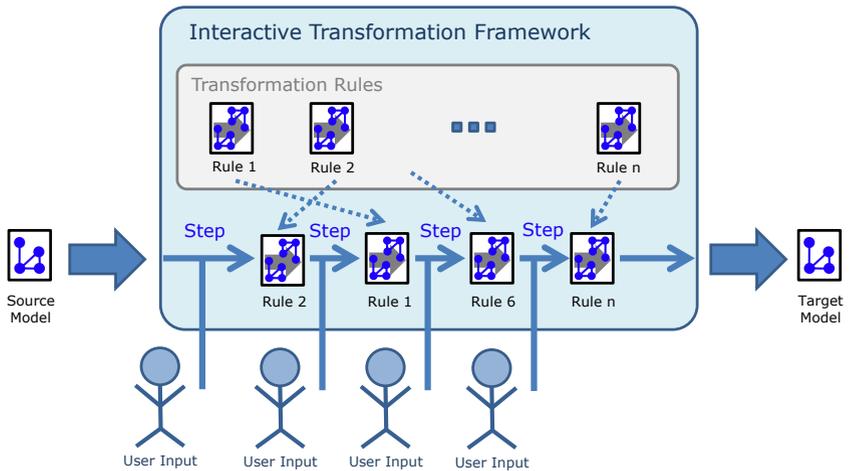
8.3 Esterel to SyncCharts Transformation

Since SyncCharts and Esterel have a very similar synchronous semantics and also very similar language features there has been work to transform one into the other. For example, Seshia et al. [SSBD99] propose a transformation from Statecharts, a SyncCharts predecessor, to Esterel for verification purposes. Esterel Studio [Est04] was able to transform SSMs, the commercial version of SyncCharts, to Esterel code. Prochnow et al. [PTvH06] developed first ideas of going the other direction that is from Esterel code to SSMs. A first exemplary implementation was given by Kühl [Küh06] also showing the correctness of transformation rules. The implementation was done in a monolithic tool named Kiel Integrated Environment for Layout (KIEL) (see Figure 6.4.2 on page 323). KIEL is a predecessor of the current Eclipse-based and more modular KIELER framework.

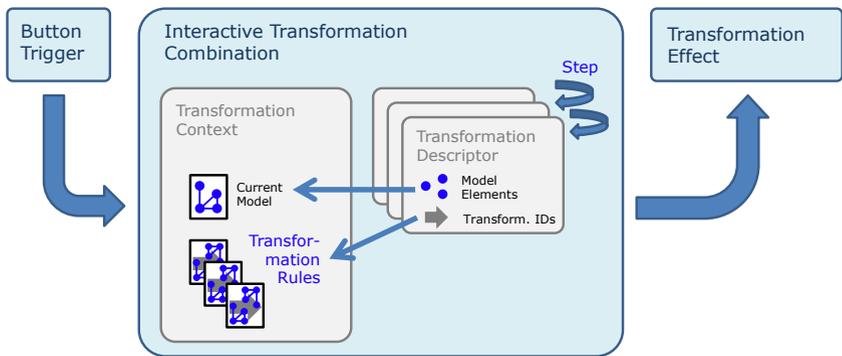
One drawback of earlier implementations is that the whole transformation is applied at once. The Esterel to SyncCharts transformation presented elsewhere [RMvH11, Mot11] tries to enhance comprehensibility of the transformation. It provides a mechanism to gain insights to intermediate transformation steps and it gives a novel interactive transformation control to the user and developer. Rüegg [Rue11] studied and implemented most parts of this transformation.

Figure 8.3.1 presents an overview of this work. Figure 8.3.1a shows the schematics of the incremental processing and the user interaction. Different transformation rules, e. g., Rule1, Rule2, are held in a pool of registered transformations. These transformations are defined in one and the same meta model. Hence, they are called *inplace model transformations*.

8. Related Projects



(a) Interactive transformations processing



(b) Interactive transformations integration

Figure 8.3.1. Interactive transformations for visual models
(from [RMvH11])

Basically, the user selects the feature to transform using the SyncCharts editor. The interactive transformation framework then derives the appro-

appropriate transformation rule to apply. The user uses provided GUI control buttons to apply the model transformation rule to the current state of the model. After the last rule is applied, the Target Model is the resulting SyncChart which is semantically equivalent to the original Esterel program. All intermediate models are SyncCharts with additional Esterel parts in it.

Figure 8.3.1b gives some insights of the interactive transformation integration into the GUI of KIELER. The user uses a Button to trigger a transformation rule. The Current Model is held in a Transformation Context, along with possible transformation rules. The lightweight Transformation Descriptor contains the user selection of model elements to transform. It also contains an ID of the transformation rule that can be applied according to the selected features. After applying transformation steps, a Transformation Effect will present the resulting intermediate or final SyncChart to the user.

This work has motivated the user interactivity of the current SCCharts compilation process. Furthermore, the incremental compilation is a generalization of the rule-based step-by-step application of the Esterel to SyncCharts transformation. Using this transformation, we were also able to gather many SyncCharts examples from a collection of Esterel programs, e. g., from Esterel Studio. These SyncCharts could easily be transformed into SCCharts. Along with their ESO trace files, these imported models play a key role for validating the current SCCharts compiler.

8.4 Synchronous Java (Light)

Synchronous Java (SJ) is an approach that allows to embed deterministic reactive control-flow in Java which encompasses preemption and concurrency. This is accomplished without the traditional Java thread concept. Hence, concurrency in SJ does not rely on an — from an application point of view — unpredictable scheduler. Instead, a lightweight application-level thread concept combines coroutines [Con63] with a synchronous MoC.

SJ is a Java-based application-level runtime environment that is meant to be used for both, direct programming synchronous Java programs for embedded targets and as a code generation target from more high-level modeling languages such as SyncCharts, SCCharts, or Ptolemy, which is

8. Related Projects

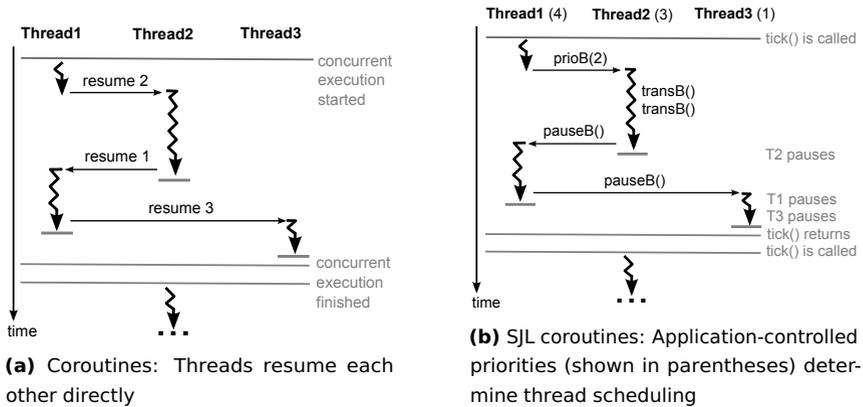


Figure 8.4.1. Comparison of SJ and coroutine thread concepts (from [MvH14])

why we give a short overview of SJ here. Further details on SJ and its successor SJL have been published earlier [MvHH13, MvH14]. Note that SJL is basically a lightweight redesign of SJ. In the following we will refer to the newer SJL but most general design aspects apply in the same way to SJ. Implementation details and differences between both are discussed in Section 8.4.3 on page 415.

8.4.1 Coroutines and Deterministic Concurrency

SJL's underlying thread scheduling is a cooperative, coroutines-like scheduling. The idea of coroutines [Con63] is to let threads cooperate, with themselves in charge of passing on control, instead of using a scheduler. Figure 8.4.1a shows an example schedule of an execution with three coroutine threads. Thread1 resumes Thread2 at some specific and well-defined point during its execution. After Thread2 has finished its work completely, it resumes Thread1 again. After finishing its work, Thread1 gives control to Thread3. SJL threads run concurrently and hand over control from one thread to another. This is unlike normal Java programs, where control-flow

8.4. Synchronous Java (Light)

is characterized by method invocations and method returns. This cooperative thread scheduling is inspired by coroutines [Con63], but in contrast to typical coroutines, in SJL it is not the yielding thread that has to specify which other thread should resume. The yielding thread merely relinquishes control, by reaching a `break` statement. Then, the scheduler chooses the thread to resume via the `state()` method. This choice is driven by the thread priorities which are application-controlled and typically static. Hence, the priorities are crucial for ordering accesses to shared data within a tick.

E. g., we can enforce a writers-before-readers discipline, which is commonly part of the synchronous MoC, by giving threads that write to a particular variable a higher priority than threads that read from that variable. The priority-based code generation for SCCharts (see Section 5.4 on page 263) is based on that principle. However, even if we do not require strict writers-before-readers, the SJL program is still deterministic, as determinism is already implied by the underlying sequential nature of the `tick()` function that does not use the Java scheduler. This is exploited, e. g., in the sequentially constructive MoC [vHMA⁺13a]. Hence, by using SJL, one is able to model the constructive semantics [Ber00b] of SyncCharts [And96] and the sequentially constructive semantics of SCCharts.

8.4.2 Synchronous C

SJL has been largely inspired by SC, also known as *SyncCharts in C* [vH09], which introduces deterministic and lightweight threads for the C language. The principles are the same. However, C and the gcc have some capabilities that Java does not have, notably *computed gotos* and a powerful preprocessor. Hence, the C variant allows to hide most of the low-level control logic in SC macros.

8.4.3 Synchronous Java — Concept and Realization

Cooperative Threads

Figure 8.4.1b shows an example schedule of three threads. Thread1 starts the control because it has the highest priority 4 when `tick()` is called.

8. Related Projects

Table 8.4.1. Similarities and differences of coroutines and SJL cooperative thread scheduling, additionally compared to Java threads (based on [MvH14])

	Coroutines	SJL	Java Threads
<i>Similarities</i>			
Threads decide to yield	Yes	Yes	No
Arbitrary interleaving	No	No	Yes
Coarse program counter	Yes	Yes	No
Deterministic	Yes	Yes	No
<i>Differences</i>			
Scheduler present	No	Yes	Yes
Next thread selected by	Thread	Scheduler	Scheduler
Selection based on	Code	Priorities (highest)	JVM+OS dependent
Threads yield by	Explicit resume	<code>prioB()</code> , <code>pauseB()</code> , <code>termB()</code>	n/a

Thread1 executes some code. It then lowers its priority to 2 by calling `prioB(2)`. The “B” at the end of the SJL keyword indicates that it must be followed by a `break` statement as discussed later in Section 8.4.3. After this priority change, Thread2 has the next highest priority 3 and is selected by the `state()` method for continuation. In the same synchronous tick, Thread2 then executes some code including two transition changes with the `transB()` operator. This means that the coarse program counter maintained by SJL for Thread2 is changed for continuation to some other label. However, this does not involve a thread re-scheduling, i. e., `transB()` is not yielding. After this, Thread2 calls `pauseB()` to indicate that it finished execution for this tick. `state()` now selects Thread1 again because it has the highest priority 2 of all running threads. Thread1 also calls `pauseB()` to indicate it has finished execution for this tick. Finally, Thread3 with priority 1 is selected to run its code. When Thread3 calls `pauseB()`, no other thread needs to be scheduled for execution in this tick. Hence, the `tick()` method returns. The first thread to run in the next tick is again the one with the

highest priority. Table 8.4.1 summarizes the similarities and differences of the SJL thread scheduling compared to coroutines. Additionally, it compares standard Java threads with both.

Program Structure

SJL is an extension to Java that is written in pure Java itself. An SJL program extends the abstract class `SJProgram` which provides the *SJL operators* (`fork`, `gotoB`, `pauseB`, ...). An enumeration lists the possible *states* that this program or system can be in. These states correspond to locations in the program which in SJL are expressed as different cases in a switch statement; if Java had a `goto` statement, these states could simply be statement labels.

A `tick()` method defines the behavior of the program for one tick. Listing 8.4.1 gives the SJL tick method of an example program. It implements the behavior of the SCChart shown in Figure 8.4.2 with two concurrent threads and possibly instantaneous communication between both threads.

ThreadA starts in state `StartA` and waits for a boolean input `In` to become true. Once `In` is true, the transition to `Sent` is taken and the internal boolean variable `Snd` is set to true. ThreadB concurrently waits for `Snd` to become true. Hence, when ThreadA transitions from `StartA` to `Sent`, it sets `Snd` to true which triggers the transition from state `StartB` to `DoneB` of ThreadB. This transition sets `Ack` to true where ThreadA is concurrently waiting for in state `Sent`. ThreadA immediately reacts to `Ack` becoming true by transitioning from state `Sent` to state `DoneA`. This finally also sets the boolean output `Out` to true. The example illustrates possibly back-and-forth communication between the two threads within one tick.

The relevant SJL code for this example is given in Listing 8.4.1. The `while` loop ensures that the computation of the complete reaction (tick), which in general may consist of several computational steps, is run until `isTickDone()` returns true. If `isTickDone()` returns true then all threads have finished their computation for the current tick. During each iteration of the `while` loop, the `state()` method call invokes a priority-based scheduler that returns the current state of the thread to be executed next which is then used in the switch statement.

8. Related Projects

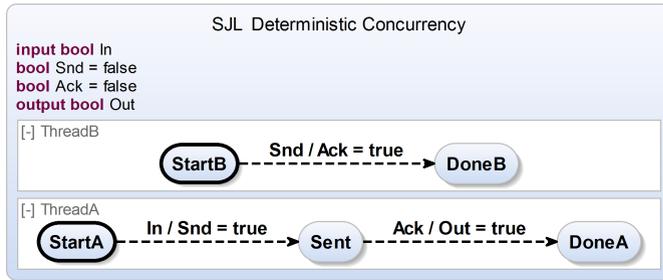


Figure 8.4.2. SCChart of SJL example

```

1 public final void tick() {
2   while (!isTickDone()) {
3     switch (state()) {
4       case Concurrency: // Prio 3
5         fork(StartB, 2);
6         forkEndB(StartA);
7         break;
8       case StartA: // Prio 3
9         if (In) {
10          Snd = true; // Prio 3
11          prioB(Sent, 1); // Prio 1
12          break;}
13        pauseB(StartA);
14        break;
15       case Sent: // Prio 1
16         if (Ack) {
17          Out = true;
18          gotoB(DoneA);
19          break;}
20        pauseB(Sent);
21        break;
22       case StartB: // Prio 2
23         if (Snd) {
24          Ack = true;
25          gotoB(DoneB);
26          break;}
27        pauseB(StartB);
28        break;
29       case DoneB: // Prio 2
30        haltB();
31        break;
32       case DoneA: // Prio 1
33        haltB();
34        break;
35     } // end switch state
36   } // end while tick
37 }

```

Listing 8.4.1. tick () method of SJL example from Figure 8.4.2

Thread Priorities

The example demonstrates the use of priorities to achieve a writers-before-readers scheduling policy for the concurrently used variables Snd and Ack. Each thread can only lower its priority within one tick. The fork statements in lines 5 and 6 fork the two threads. The initially forking thread is started with priority 3 which is not shown here and which is part of the initialization. The thread with priority 3 forks ThreadB (line 5) with distinct priority 2

8.4. Synchronous Java (Light)

and resumes itself as ThreadA (line 6) with priority 3. The coroutine-like cooperative scheduling is realized by reaching a break that terminates the current case of the switch statement and leads to the next scheduler call. Therefore, the SJL operators that upon their completion require a scheduler call must always be followed by a break statement. These operators are called *breaking* operators and are denoted by a postfix “B” in their name.

The forkEndB statement in line 6 is such an SJL operator. The break in line 7 terminates the switch statement and enforces a re-entry of the outer loop. The state() call in line 3 effectively is the scheduler call. Since ThreadA has the highest priority 3, it is the first to be selected. The reason why ThreadA has a higher priority than ThreadB is that ThreadA is the writer to variable Snd and ThreadB is the reader. Hence, ThreadA’s code for case StartA in line 9 is executed first. If input In is true then Snd is set to true in line 10. The prioB statement in line 11 sets the priority of ThreadA to 1 which is lower than ThreadB’s priority 2. The reason is that ThreadB is the writer to variable Ack and ThreadA the reader. Hence, ThreadA has to wait until ThreadB has written to Ack. The break in line 12 again terminates the switch statement and enforces another re-entry and re-scheduling. Now, ThreadB’s code for case StartB in line 23 is executed. It sets Ack to true in line 24. The gotoB statement in line 27 will make the current thread continue at this case at the next re-scheduling which is triggered by the break in line 25. Hence, ThreadB which has still the higher priority 2 is continued at case DoneB in line 30. The haltB statement stops the execution for the current tick and all further ticks. After the break in line 31 another re-scheduling is triggered. This time, there is only one active thread left for this tick. This is ThreadA with priority 1. Hence, ThreadA’s code for case Sent is executed because ThreadA has set this case label as a continuation point when reducing its priority from 3 to 1 in line 11 with prio(Sent, 1). Now, ThreadA reads the Ack variable which has been set to true and sets the output Out to true. It continues at case DoneA where it also halts. As there are no more active threads for this tick, the tick method returns true. In any next ticks both threads ThreadA and ThreadB will continue in their DoneA and DoneB cases, respectively.

To conclude, the example showed how priorities in SJL can be used to express deterministic concurrency.

8. Related Projects

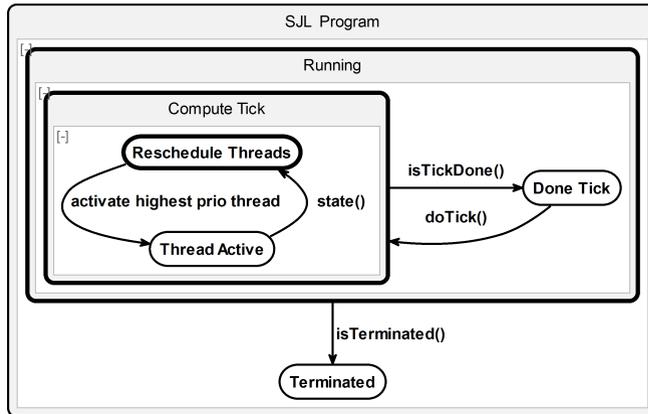


Figure 8.4.3. State diagram of an SJL program's life cycle (adapted from [MvH14])

Thread Scheduling

Figure 8.4.3 shows the life cycle of a complete SJL program as a state diagram. Initially, the program is running, computes a tick, and reschedules the active threads. The thread with the highest priority is chosen to be dispatched and run next. It is then executed at its current label. Such labeled case of the switch case statement serves as a coarse grain program pointer for a thread. In a coroutine-style, the thread is not preempted but must at some point give control back. In contrast to coroutines, one thread does not pass control directly to another thread but gives control back to the scheduler. This is done when a break is executed and the loop is restarted which results in a scheduler call by the `state()` method. A break should occur by convention after special SJL operators as `pauseB()` or `prioB()`. When there is no active thread any more for a tick then `isTickDone()` eventually returns true. A call to the wrapper method `doTick()` starts the reaction computation of the `tick()` method for the next tick until all threads have been terminated, i. e., `isTerminated()` returns true. Then, the whole program terminates.

8.4. Synchronous Java (Light)

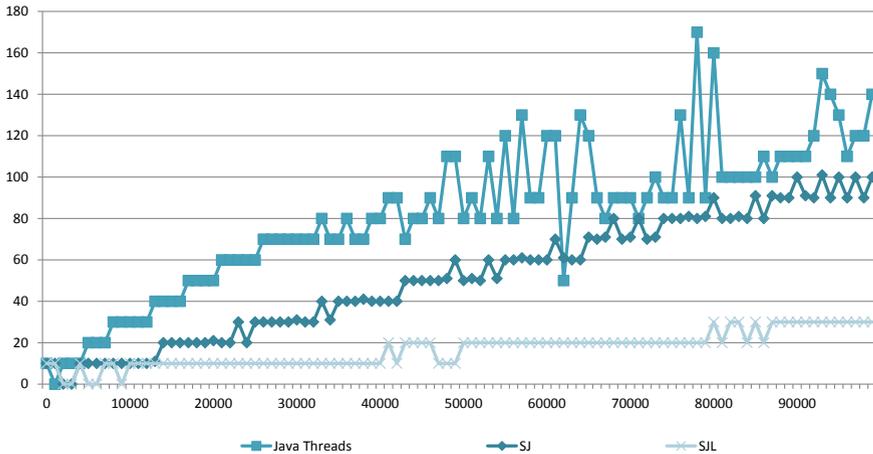


Figure 8.4.4. Worst-case runtimes, SJL vs. SJ vs. standard Java threads, for a concurrent producer consumer example (from [MvH14])

Implementation Notes

SJ [MvHH13] is implemented in Java and hence platform independent. It basically provides an abstract class that can be used to derive SJ programs. The implementation provides the SJ operands as well as the scheduler and its bookkeeping of application-level SJ threads that come with the central methods `tick()` and `isTickDone()`.

SJL [MvH14] is the successor of SJ. It is even more lightweight but exploits the same principles. Where SJ came with explicit support for synchronous Esterel-style signals, SJL does not have this feature for two reasons: 1. The signal implementation of SJ made use of Java reflection which is not available on all platforms, 2. SJ is less efficient compared to SJL and has higher memory requirements during runtime because simple communication results in various method calls on various necessary signal objects. Furthermore, the bookkeeping in SJL was redesigned to use significantly more efficient bit-level operations.

8. Related Projects

Experimental Results

SJ and SJL bring concepts borrowed from synchronous languages to Java in order to specify deterministic concurrency. In our experiments, our goal was to measure the gain in predictability and efficiency of these constructs. Hence, we compared Java *with* synchronous concepts to Java *without* synchronous concepts.

To illustrate the predictability and the efficiency of the SJL approach compared to Java threads, we compared the runtimes of the Java thread version and the SJL version as well as a version of its successor SJL for a common and highly concurrent *producer consumer example*. The example is discussed elsewhere [MvH14].

We ran all three programs on an Intel Core 2 Duo T9800 @ 2.93 GHz machine with 8 GB of RAM and a 64 Bit Java Virtual Machine (JVM) with a variable number of ticks, that were equal to the number of data produced/observed by each implementation. So, for each variable number of ticks, all implementations had the same task to fulfill.

Figure 8.4.4 shows the runtime of each implementation over the variable number of ticks of the SJL variant compared to SJ and to the Java threads implementation. For getting reasonable results, we made three experiments for each number of ticks and took the worst execution time. We considered tick numbers between 0 and 10,000 in linear steps of 1,000.

The SJ version is already faster (average of 1.75 times faster) compared to the Java threads version that has to struggle with more overhead due to possibly poorly scheduled executions. Moreover, the SJL version is considerably faster (average of 4.5 times faster) compared to the Java threads version due to its very lightweight bit-level implementation strategy. Another, perhaps more important difference is the variability of the worst-case runtime. While the Java threads version is heavily unpredictable, especially when it comes to more duty, i.e., more ticks, the SJL as well as the SJ variants are much closer to a linear growth and hence more predictable. Both facts support our hypothesis that SJ and, even more so, SJL is much more lightweight and predictable compared to the Java threads variant. Both is important for safety-critical reactive embedded systems.

Conclusion and Future Work

9.1 Conclusion

This thesis proposes SCCharts as a language for specifying, modeling, and implementing software that is desired to run on embedded systems (cf. Figure 9.1.1). As such systems often are safety-critical, the focus for SCCharts is to provide reliability w. r. t. (1) the process of modeling (cf. Chapter 3, Chapter 4, and Chapter 6) as well as (2) the compilation of SCCharts

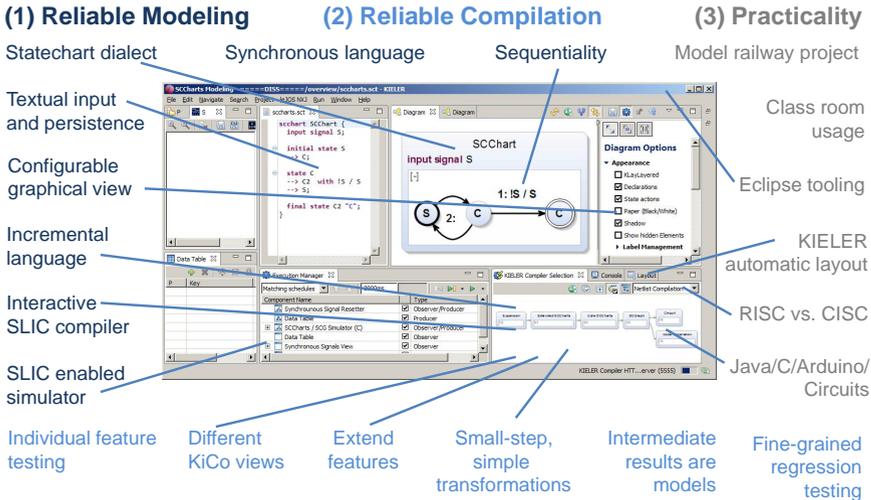


Figure 9.1.1. SCCharts reliable modeling and compilation

9. Conclusion and Future Work

(cf. Chapter 5). Finally, (3) the tooling for modeling and implementing safety-critical embedded software must be practically usable (cf. Chapter 7).

9.1.1 Reliable Modeling

For gaining reliable models, the SCCharts language is designed to be grounded on a small set of base features that are incrementally extended by optional advanced features (cf. Section 3.2 on page 52). These help to hide complexity of the model for a better readability (cf. Figure 7.2.2a on page 371).

The incremental concept helps to flatten the learning curve for SCCharts (cf. Figure 7.3.1 on page 376). Additionally, the interactive incremental SLIC concept is introduced (cf. Section 4.1 on page 89). This helps to understand each language feature separately and in terms of the concrete model, in which a specific feature is supposed to be used. Thus, the modeler is able to learn about all language features and their exact meaning in general and about their specific use-cases (cf. Section 4.1.6 on page 103).

Furthermore, a flexible graphical view helps to maintain an overview and to understand and maintain models (cf. Section 4.2.2 on page 106). The textual input and persistence format comes with numerous benefits for modeling and code management (cf. Table 6.4.1). SCCharts was chosen to be a statechart dialect because control software for safety-critical systems often needs to reflect system states (cf. Section 1.0.3 on page 7).

SCCharts is grounded on a synchronous semantics to provide deterministic concurrency. The sequentially constructive MoC ensures that more valid models are accepted than classical synchronous languages would accept. It further makes SCCharts intuitively usable by programmers that are used to imperative languages such as C or Java (cf. Section 2.6 on page 36).

9.1.2 Reliable Compilation

The incremental definition of SCCharts features enables to apply the SLIC approach for compilation (cf. Chapter 5). In SLIC, all intermediate results are valid models that can be inspected by the tool smith or even by the

modeler. This helps to validate and maintain the compiler as each transformation can be verified and fixed separately (cf. Section 4.2.2 on page 106). The interactive compiler offers different views (cf. Figure 6.5.7 on page 346) to validate, e. g., the configuration of transformation interdependencies and categories. It is fully integrated into a regression testing suite for a build automation system which aids maintaining the compiler (cf. Section 6.6 on page 353).

9.1.3 Practicality

To demonstrate that the SCCharts tooling and compilation is practically usable, a student course (cf. Section 7.2 on page 367) covered the task of modeling a highly concurrent train controller for a medium-sized model railway system. Additionally, SCCharts along with the KIELER SCCharts tooling and compiler was used in the class room. Both applications proved a good maintainability (cf. Figure 7.2.2b on page 371) of the overall approach. The performance could even be optimized. The code size is already reasonable (cf. Section 7.2.2 on page 368), but it still leaves potential for optimization to target even embedded systems with smaller memory like Lego Mindstorms. Different code generation strategies such as RISC or CISC were discussed including a short comparison of a circuit-based and a priority-based compilation approach (cf. Section 5.6 on page 276). To show that the SLIC concept not only applies to one target, Java code, C code, Arduino code, and even HW circuits were generated (cf. Section 5.4 on page 263, Section 5.5 on page 273, and Section 5.7 on page 283).

9.2 Future Work

The SCCharts language, the tooling, and its SLIC-based compilation already proved itself to be usable even for medium-sized projects like the model railway controller. Still, there are several possibilities to extend the presented work. Furthermore, there are some open and ongoing discussions and questions about language features, usability aspects, and implementation details. Some of these open topics are documented in the next sections.

9. Conclusion and Future Work

9.2.1 SCCharts

Language

- ▶ The sequentially constructive MoC permits an explicit unemit as proposed for the conservative Esterel extension SCEst [RSM⁺15]. Since SCCharts also are based on the sequentially constructive MoC, it seems reasonable to discuss the introduction of an unemit also for signals in SCCharts.
- ▶ SyncCharts entry actions are preemptable by a strong abort and exit actions are not. SCCharts entry actions are not preemptable. Hence, they can be used for representing initializations which are expected to happen even if a state is immediately left again. Though, to completely represent SyncCharts entry actions, it might be desirable to have a separate *before action* that is non-preemptable, i. e., will be executed even if the state is left with a strong immediate abort transition. It is rather easy to implement a preemptable SyncCharts-like *entry action*. Therefore, such entry actions must be transformed before the abort transformation takes place. A not-handled-by dependency from entry to abort is sufficient. The current entry action transformation should then be renamed to *before action*. Note that both transformations share nearly the same code for expanding the appropriate feature. Hence, these transformations should be implemented using the same KiCo processors (cf. Section 6.5.4 on page 337). Further note that initializations must then be expanded to before actions rather than entry actions.

Similarly, it might be convenient to have a preemptable exit action, such as a *leave action*. Alternatively, one could make the current exit action preemptable and introduce another *after action* which is non-preemptable and has the SyncCharts exit action semantics.

- ▶ The current count delay feature can only be used with transition triggers. However, as described on page 205, it could be desirable to allow also other triggers with a count delay.
- ▶ The current compiler implementation has drawbacks in situations when a hierarchical state is left more than once in a tick. This was explained earlier in Section 5.2.8 on page 175. One reason is that the decision which path to leave a state in the control-flow is depending on data and

is not a distinct control-flow path. However, such a distinct path may help resolving dependency cycles. It could be promising to investigate whether a conditional termination in Core SCCharts and on SCG level might help to prevent dependency cycles induced by the above mentioned situations. It should be further investigated on alternative abort feature transformations, e. g., using conditional terminations only.

- ▶ The weak suspend feature expansion currently is not schedulable for the above mentioned reasons. Once the compiler can handle such cases, the weak suspend transformation may also need some tweaking, especially for the hierarchical case.
- ▶ SCADE allows to model (synchronous) data-flow combined with control-flow. Reactive systems often have a control-flow oriented task. Hence, SCCharts are a perfect match to model system states and transitions in a control-flow oriented fashion. Still, sometimes input or internal data needs to be transformed. This is a use-case that was often captured by entry or during actions in recent SCCharts projects. For such and other use-cases, where manipulating data is in focus, a data-flow extension for SCCharts and a suitable SLIC-compatible compilation strategy is desirable.
- ▶ Immediate transitions are a building block for both, Extended and Core SCCharts. Especially when applying the SLIC transformations in order to compile SCCharts, it turns out that there are significantly more immediate transitions compared to the number of delayed transitions. Even for modeled SCCharts this seems to be the case which should be investigated further. Another issue are “implicit immediate transitions” such as the transitions going out from a connector state (see Section 3.2.8 on page 67). An alternative could be to replace the immediate keyword in SCT and the immediate flag for transitions in the meta model of SCCharts by a *delayed* keyword and a delayed flag. Hence, immediate transitions would become the default transition type which may even make more sense and simplify the implicit/explicit immediate perspective.
- ▶ Static signals: The current implementation allows to specify static variables only (cf. Section 5.2.16 on page 210). However, it could make sense to also allow static valued signals. A not-handled-by dependency must

9. Conclusion and Future Work

be added from the signal transformation to the static feature such that the static feature is eliminated for signals first. The reason is that a (valued) signal gets transformed into several variables and during actions which should be all raised to the root state. Alternatively, one may allow *static during actions* and attach the static keyword in the signal transformation to the created variables and during actions. This way, one could avoid a dependency between the SyncCharts and the SCADE feature group.

- ▶ The history feature is currently transformed before the suspend feature. This causes the effect that the a re-entered state that is immediately suspended may be *entered late* when the suspend trigger will not hold any more. To change this, the not-handled-by dependency could be altered to Suspend \rightarrow_{nhb} History.
- ▶ Deferred transitions can take shape as a shallow or deep variant. Both feature transformations are defined and implemented (cf. Section 5.2.17 on page 211). However, the SCT syntax currently lacks a “deep” keyword for differentiating both variants. Additionally, the SCCharts diagram synthesis needs an extension to visually draw the asterisk.
- ▶ The transformations defined in this thesis are given in pseudocode, reasoned by various examples, and implemented in the Xtend language. Still, a formal semantics for defining SCCharts and the SLIC transformations seems desirable. This would also help to prove semantic equivalence for two SCCharts, e. g., gained by compilations that use different SLIC schedules or compilers.

Compilation

- ▶ The current SCCharts compiler evolved as discussed in Section 5.1 on page 116. It uses a circuit-based RISC compilation strategy. This compensates execution time drawbacks of the RISC approach as explained earlier. The circuit-based low-level synthesis also evolved as the default for SCCharts compilation because time and memory predictability is key for targeting safety-critical embedded real time systems. Improving time and memory efficiency of the RISC (compared to the CISC) compilation strategy is an area of interest for future work.

- ▶ SCCharts target the development of reliable software for safety-critical systems. These are often embedded systems which come with limited resources. Currently, the code generation does not make use of much optimizations to produce smaller code. However, KiCo 2.0 (see Section 6.5 on page 332) has been prepared to support processors. Re-usable optimization steps should be implemented as such processors.
- ▶ Nested pre operators are supported conceptually and also by the implemented transformation as discussed in Section 5.2.12 on page 193. Still, there is no detection of the nesting level yet and the pre transformation is currently just called once per compilation pass. One might investigate if the multiple call of one transformation should be part of the transformation or lifted conceptually to the feature/KiCo level.
- ▶ KiCo 2.0 mainly re-uses the GUI of the first KiCo implementation. Drawbacks are that processors are not shown in the GUI. Hence, the user cannot enable or disable processors at this time. It is desired to significantly revise the GUI of KiCo to fulfill all needs of future interactive compilation. This includes the processors but also the GUI may highlight all features that are contained in the model automatically. In principle, KiCo is already capable of providing this kind of information.
- ▶ The internal graph representation of KiCo 2.0 does not distinguish produces and not-handled-by dependencies. However, this is desirable in order to only follow produces dependencies as described in the algorithm in Section 6.5.4 on page 340.
- ▶ Since the new HW circuit compilation (see Section 5.7.2 on page 298) is fully part of the SCCharts KIELER compiler, it can be used together with the online compiler, with the command line compiler, and with other KIELER compiler integrations. The developed SSA representation for SCGs of the new project conceptually is based on results of its predecessor. However, as it seamlessly integrates into the current compile chain, it can easily be re-used, e. g., for generating SSA-style Esterel or Lustre code from SCGs for alternative SCCharts code synthesis paths.
- ▶ Since the HW circuit visualization (see Figure 5.7.8 on page 300) is part of KIELER, specially developed data-flow layout algorithms [SSvH14] could be validated/optimized for dedicated HW circuit design.

9. Conclusion and Future Work

- ▶ The new circuit compilation (see Section 5.7.2 on page 298) does not make use of HW description languages such as VHDL. It might be desirable to implement a separate transformation from SSA-SCG to VHDL in order to be able to download and run SCCharts on FPGAs again.
- ▶ The choice of core and extended features was made carefully but evolved over time (see Section 3.2 on page 52 and Section 5.6 on page 276). It would be beneficial to back this or another choice with more updated evaluation data as the new compiler infrastructure is ready to easily compare different compilation paths.
- ▶ As explained in Section 3.2.7 on page 66, initializations of local and output variables may be crucial to avoid any kind of (unexpected) non-determinism. Therefore, already the high-level compilation shall support *default initializations* for variables which are not explicitly initialized. Additionally, regarding robustness, the tooling shall check for uninitialized variables and bring these to the modeler's attention.

Tooling

- ▶ The element tracing, introduced earlier as an extension to the SLIC approach, is already in use to trace timing information [FBSvH14]. The simulation visualization of, e.g., active states or taken transitions is currently based on source-model-transformations (see Section 8.2.3 on page 397). This has the advantage of being able to use various (black box) compilers in addition to KiCo. However, a drawback can be that source-model-transformations change the model for visualization purposes. Such changes should not change the semantics but it still can make a difference if the compiler accepts the original model and conservatively rejects the transformed model. Hence, it might be preferable, as a default, to use the element tracing also for simulation visualization purposes.

Validation

- ▶ The number of participants of the model railway practical course was rather small. Hence, it seems desirable to exploit a larger group of

participants to evaluate and validate modeling capabilities of the KIELER SCCharts implementation.

- ▶ Regarding the model railway, it could be valuable and challenging to evaluate the SCCharts language and the tooling for a distributed model railway controller with distributed code generation in future projects.

9.2.2 SCEst and Esterel

- ▶ Using a special reactive processor such as the KEP [LvH12] that is able to process arbitrary Esterel programs combines the advantages of a dedicated HW with fast development turn around times. The current SCCharts to Esterel transformation leads to large Esterel programs even for small models as the AO example (see Section 5.7.2 on page 296). This makes it practically not usable, especially for embedded systems that typically have limited resources. It is future work to come up with a more compact and efficient transformation.
- ▶ Even a special reactive processor similar to the KEP but for executing SCCharts could be envisioned.
- ▶ The current prototype for the SCCharts to Esterel transformation only accepts SCCharts that are Berry-constructive. Using techniques as transforming sequentially constructive SCCharts to SSA-style-Esterel code as presented elsewhere [RSM⁺15] could make it possible to loosen this limitation.
- ▶ Currently, the SCEst implementation is entangled with the KIELER Esterel implementation. It might be desirable to have a clearer separation of both implementations.
- ▶ One could envision a SLIC compilation approach for SCEst which may also involve a separation of core and extended language features like it was done for SCCharts.
- ▶ The evaluation showed that, in terms of the reaction time, SCEst is quite competitive. It is about 30% faster on average than the CEC. In terms of the code size the picture looks different: SCEst generates on average about twice as much code than the CEC which leaves much room for future improvements.

9. Conclusion and Future Work

9.2.3 Related Projects

- ▶ It further might be interesting to explore how to fully express all valid sequentially constructive SCCharts in Ptolemy, e. g., by using the Coroutine MoC [SL12].

Bibliography

- [All02] Tiziana Allegrini. Code generation starting from statecharts specified in UML. Master's thesis, Università Degli Di Pisa, Facoltà di Ingegneria, 2002.
- [Ame10] Torsten Amende. Synthese von SC-Code aus SyncCharts. Diploma thesis, Kiel University, Department of Computer Science, May 2010.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/tam-dt.pdf>.
- [And95] Charles André. Synccharts: A visual representation of reactive behaviors. Technical report, October 1995.
- [And96] Charles André. SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR 96–56, I3S, Sophia-Antipolis, France, Rev. April 1996.
- [And00] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison Wesley, 2000.
- [And03] Charles André. Semantics of SyncCharts. Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.
- [And15] Lewe Andersen. Quadrocopter flight control design using SCCharts. Bachelor thesis, Kiel University, Department of Computer Science, September 2015.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lan-bt.pdf>.
- [ASK04] Aditya Agrawal, Gyula Simon, and Gabor Karsai. Semantic translation of simulink/stateflow models to hybrid automata

Bibliography

- using graph transformations. *Electronic Notes in Theoretical Computer Science.*, 109:43–56, 2004.
- [ASU07] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, 2007.
- [AT00] Jauhar Ali and Jiro Tanaka. Converting Statecharts into Java code. In *Proceedings of the Fourth World Conference on Integrated Design and Process Technology (IDPT '99)*, Dallas, Texas, June 2000. Society for Design and Process Science (SDPS).
- [Bar99] Michael Barr. *Programming Embedded Systems in C and C++*. O'Reilly, 1999.
- [Bay09] Özgün Bayramoglu. The KIELER textual editing framework. Diploma thesis, Kiel University, Department of Computer Science, December 2009.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/oba-dt.pdf>.
- [BBF⁺01] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, Beatrice Berard, Michel Bidoit, Alain Finkel, Francois Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer-Verlag, Berlin Heidelberg, 2001.
- [BC84] Gérard Berry and Laurent Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, volume 197 of LNCS, pages 389–448. Springer-Verlag, 1984.
- [BCD⁺87] Patrick Borras, Dominique Clement, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valerie Pascual. CENTAUR: the system. Technical Report 777, INRIA Grenoble Rhône-Alpes, 1987.

- [BCE⁺03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. In *Proc. IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, Piscataway, NJ, USA, January 2003. IEEE.
- [Ber91] Gérard Berry. A hardware implementation of pure ESTEREL. Technical Report de recherche 1479, INRIA, 1991.
- [Ber00a] Gérard Berry. *The Esterel v5 Language Primer, Version v5_91*. Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 06565 Sophia-Antipolis, 2000.
<ftp://ftp.inrialpes.fr/pub/synalp/reports/esterel-primer.pdf.gz>.
- [Ber00b] Gérard Berry. The foundations of Esterel. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 425–454, Cambridge, MA, USA, 2000. MIT Press.
- [Ber02] Gérard Berry. *The Constructive Semantics of Pure Esterel*. Draft Book, Version 3.0, Centre de Mathématiques Appliquées, Ecole des Mines de Paris and INRIA, 2004 route des Lucioles, 06902 Sophia-Antipolis CDX, France, December 2002.
- [Ber15] Gerad Berry. Time and events, from physics to informatics. Presentation at the 22th International Open Workshop on Synchronous Programming (SYNCHRON'15), Kiel, Germany, December 2015.
- [BK00] Gérard Berry and Mike Kishinevsky. Hardware Esterel language extension proposal. In *Workshop on Synchronous Reactive Languages*, November 2000.
- [BLL⁺08] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, Haiyang Zheng, Shuvra S. Bhattacharyya, Elaine Cheong, John Davis, Mudit Goel, Bart Kienhuis, Man-Kit Leung, Jie Liu, Lukito Muliadi, , John Reekie,

Bibliography

- Neil Smyth, Jeff Tsay, Brian Vogel, Winthrop Williams, and Yuhong Xiong. Heterogeneous concurrent modeling and design in java (volume 1: Introduction to Ptolemy II). Technical Report ADA519074, University of California, Berkeley, CA, 94720, USA, 2008.
- [BTvH08] Marian Boldt, Claus Traulsen, and Reinhard von Hanxleden. Compilation and worst-case reaction time analysis for multi-threaded Esterel processing. *EURASIP Journal on Embedded Systems*, 2008:1–21, 2008.
- [Bus16] Jonas Busse. SCCharts modeling for embedded systems with limited resources. Master thesis, Kiel University, Department of Computer Science, September 2016.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jbu-mt.pdf>.
- [Cal10] John Calcote. *Autotools: A Practioner's Guide to GNU Autconf, Automake, and Libtool*. No Starch Press, 2010.
- [Car10] John Julian Carstens. Datenvisualisierung in grafischen modellen. Bachelor thesis, Kiel University, Department of Computer Science, September 2010.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jjc-bt.pdf>.
- [CDH⁺00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448, 2000.
<http://ti.arc.nasa.gov/m/tech/rse/publications/papers/ICSE00/bandera.pdf>.
- [CEC] CEC: The columbia Esterel compiler.
<http://www1.cs.columbia.edu/~sedwards/cec>.

- [CFR⁺91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
- [Con63] Melvin E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.
<http://doi.acm.org/10.1145/366663.366704>.
- [CPHP87] P. Caspi, D. Pilaud, N. Halbwegs, and J. A. Plaice. Lustre: a declarative language for programming synchronous systems. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'87)*, pages 178–188, Munich, Germany, 1987. ACM.
- [CPP05] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, pages 173–182, New York, NY, USA, September 2005. ACM.
- [DGHKL80] Veronique Donzeau-Gouge, Gerard Huet, Gilles Kahn, and Bernard Lang. Programming environments based on structured editors: The Mentor experience. Technical Report 26, INRIA Grenoble Rhône-Alpes, 1980.
- [Dou99] Bruce Powel Douglass. UML Statecharts. pages 22–42, January 1999.
<http://m.eet.com/media/1118718/f-dougla.pdf>.
- [dRB06] Jim des Rivieres and Wayne Beaton. Eclipse platform technical overview. *Eclipse Corner Articles*, April 2006. <http://www.eclipse.org/articles>.
- [Dud12] Björn Duderstadt. A statechart dialect with sequential constructiveness. Diploma thesis, Kiel University, Department of Computer Science, December 2012.

Bibliography

- [EJL⁺03] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.7.2273&rep=rep1&type=pdf>.
- [Est04] Esterel Technologies. *Esterel Studio User Manual*, 5.2 edition, July 2004.
- [Est05] Esterel Technologies. *The Esterel v7 Reference Manual Version v7_30—initial IEEE standardization proposal*. Esterel Technologies, 679 av. Dr. J. Lefebvre, 06270 Villeneuve-Loubet, France, November 2005.
- [Est11] Esterel Technologies. *SCADE Language Reference Manual*, sc-irm - sc/u1-62 edition, March 2011.
- [Est16] Esterel Technologies, Inc. *SCADE Suite: Control and Logic Application Development*, last visited 03/2016.
<http://www.esterel-technologies.com/products/scade-suite>.
- [EZ07] Stephen A. Edwards and Jia Zeng. Code generation in the Columbia Esterel Compiler. *EURASIP Journal on Embedded Systems*, Article ID 52651, 31 pages, 2007.
- [FBSvH14] Insa Fuhrmann, David Broman, Steven Smyth, and Reinhard von Hanxleden. Towards interactive timing analysis for designing reactive systems. Reconciling Performance and Predictability (RePP’14), satellite event of ETAPS’14, April 2014.
- [Fis95] Paul A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1995.

- [FSKdR05] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P. de Roever. 29 new unclarities in the semantics of UML 2.0 State Machines. In *ICFEM*, volume 3785 of *LNCS*, pages 52–65. Springer, 2005.
- [Fuh11] Hauke Fuhrmann. *On the Pragmatics of Graphical Modeling*. Dissertation, Kiel University, Faculty of Engineering, Kiel, 2011.
- [GHL⁺13] John C. Grundy, John Hosking, Karen Na Li, Norhayati Mohd Ali, Jun Huh, and Richard Lei Li. Generating Domain-Specific Visual Language Tools from Abstract Visual Specifications. *IEEE Transactions on Software Engineering*, 39(4):487–515, April 2013.
<http://www.ict.swin.edu.au/personal/jgrundy/papers/tse2012.pdf>.
- [GR83] S. Moisan G. Berry and J-P. Rigault. Esterel: Towards a synchronous and semantically sound high-level language for real-time applications. In *IEEE Real-Time Systems Symposium*, pages 30–40, 1983. IEEE Catalog 83CH1941-4.
- [Han10] Sören Hansen. Configuration and automated execution in the KIELER execution manager. Bachelor thesis, Kiel University, Department of Computer Science, March 2010.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/soh-bt.pdf>.
- [Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [Har13] Wahbi Haribi. A SyncCharts editor based on YAKINDU SCT. Master’s thesis, Kiel University, Department of Computer Science, March 2013.
- [HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel

Bibliography

- Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.
- [Hei10] Mirko Heinold. Synchronous Java, September 2010. Bachelor thesis, Kiel University, Department of Computer Science.
- [HN96] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.
- [Höh06] Stephan Höhrmann. Entwicklung eines modularen Feldbussystems zur Steuerung einer Modellbahnanlage. Diploma thesis, Kiel University, Department of Computer Science, March 2006.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sho-dt.pdf>.
- [HP85] D. Harel and A. Pnueli. On the development of reactive systems. *Logics and models of concurrent systems*, pages 477–498, 1985.
- [HR04] Grégoire Hamon and John Rushby. An operational semantics for Stateflow. In M. Wermelinger and T. Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *LNCS*, pages 229–243, Barcelona, Spain, April 2004. Springer.
- [JM94] Farnam Jahanian and Aloysius K. Mok. Modechart: A specification language for real-time systems. *IEEE Trans. Softw. Eng.*, 20(12):933–947, December 1994.
<http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=368134>.
- [Joh13] Gunnar Johannsen. Hardwaresynthese aus SCCharts. Master thesis, Kiel University, Department of Computer Science, October 2013.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/gjo-mt.pdf>.

- [Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland Publishing Co., August 1974.
- [Klo10] Paul Klose. Beispielmanagement in KIELER. Bachelor thesis, Kiel University, Department of Computer Science, September 2010.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/pkl-mt.pdf>.
- [Kri07] Jens Krinke. A study of consistent and inconsistent changes to code clones. In *Proceedings of the 14th Working Conference on Reverse Engineering, WCRE '07*, pages 170–178, Washington, DC, USA, 2007. IEEE Computer Society.
<http://www.computer.org/csdl/proceedings/wcre/2007/3034/00/30340170.pdf>.
- [Küh06] Lars Kühl. Transformation von Esterel nach SyncCharts. Diploma thesis, Kiel University, Department of Computer Science, February 2006.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lku-dt.pdf>.
- [LCFH14] Ivan Llopard, Albert Cohen, Christian Fabre, and Nicolas Hili. A parallel action language for embedded applications and its compilation flow. In *17th International Workshop on Software and Compilers for Embedded Systems, SCOPES '14, Sankt Goar, Germany, June 10-11, 2014*, pages 118–127, 2014.
<http://doi.acm.org/10.1145/2609248.2609257>.
- [Lee03] Edward A. Lee. Overview of the Ptolemy project. Technical Memorandum UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2003.
- [Lee06] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

Bibliography

- [Lee09] Edward A. Lee. Finite state machines and modal models in Ptolemy ii. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley, Nov 2009. URL:
<http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-151.html>.
- [LHBJ06] Denivaldo Lopes, Slimane Hammoudi, Jean Bézivin, and Frédéric Jouault. Mapping specification in mda: From theory to practice. In *In: Konstantas, D., Bourrières, J.-P., Léonard, M., Boudjlida, N. (Eds). Interoperability of Enterprise Software and Applications - INTEROP-ESA*, pages 253–264. Springer, 2006.
- [LHS10] M.W. Lew, T.B. Horton, and M.S. Sherriff. Using Lego mindstorms nxt and lejos in an advanced software engineering course. In *Software Engineering Education and Training (CSEET), 2010 23rd IEEE Conference on*, pages 121–128, March 2010.
<http://www.intercom.virginia.edu/~sherriff/papers/CSEET2010-Lew.pdf>.
- [LL98] Bilung Lee and Edward A. Lee. Hierarchical concurrent finite state machines in Ptolemy. In *CSD '98: Proceedings of the 1998 International Conference on Application of Concurrency to System Design*, page 34, Washington, DC, USA, 1998. IEEE Computer Society.
- [LM01] Gerald Lüttgen and Michael Mendler. Statecharts: From visual syntax to model-theoretic semantics. In *GI Jahrestagung (1)*, pages 615–621, 2001.
- [LNW03] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers (JCSC)*, 12(3):231–260, 2003.
<http://ptolemy.eecs.berkeley.edu/papers/02/actorOrientedDesign/newFinal.pdf>.

- [Loc92] C. Douglass Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–53, March 1992.
<http://www.douglocke.com/Downloads/CylicvPriorityExecs.pdf>.
- [LS11] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. Lulu, 2011.
<http://LeeSeshia.org>.
- [LvH06] Xin Li and Reinhard von Hanxleden. A concurrent reactive Esterel processor based on multi-threading. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06), Special Track Embedded Systems: Applications, Solutions, and Techniques*, Dijon, France, April 23–27 2006.
- [LvH12] Xin Li and Reinhard von Hanxleden. Multi-threaded reactive programming—the Kiel Esterel Processor. *IEEE Transactions on Computers*, 61(3):337–349, March 2012.
- [Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.
- [Mac15] Felix Machaczek. Collision avoidance of safety-critical real-time systems. Bachelor thesis, Kiel University, Department of Computer Science, September 2015.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/fma-bt.pdf>.
- [Mar92] Florence Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*. Springer Verlag, LNCS 630, August 1992.
- [Mea55] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34:1045–1079, September 1955.
- [MFvH10] Christian Motika, Hauke Fuhrmann, and Reinhard von Hanxleden. Semantics and execution of domain specific

Bibliography

models. In *2nd Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2010) INFORMATIK 2010*, GI-Edition – Lecture Notes in Informatics (LNI), pages 891–896, Leipzig, Germany, September 2010. Bonner Köllen Verlag.

- [MFvHL12] Christian Motika, Hauke Fuhrmann, Reinhard von Hanxleden, and Edward A. Lee. Executing domain-specific models in Eclipse. Technical Report 1214, Kiel University, Department of Computer Science, October 2012. ISSN 2192-6247.
- [Mot09] Christian Motika. Semantics and execution of domain specific models—KlePto and an execution framework. Diploma thesis, Kiel University, Department of Computer Science, December 2009.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf>.
- [Mot10] Christian Motika. Executing SyncCharts with Ptolemy. Presentation at the 17th International Open Workshop on Synchronous Programming (SYNCHRON'10), Frejus, France, December 2010.
- [Mot11] Christian Motika. Interactive Esterel to SyncCharts transformation for executing Esterel with Ptolemy. Presentation at the 18th International Open Workshop on Synchronous Programming (SYNCHRON'11), Dammarie-les-Lys, France, December 2011.
- [Mot14a] Christian Motika. Model Railway 2.0 - An Exploration of Alternatives for Interface Hardware, 2014.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/talks/railway2.0-cmot-handout.pdf>.
- [Mot14b] Christian Motika. SCCharts in motion — interactive model-based compilation for a railway system. Presentation at the 21th International Open Workshop on Synchronous Programming (SYNCHRON'14), Aussois, France, December 2014.

- [MPP10] Louis Mandel, Florence Plateau, and Marc Pouzet. *Mathematics of Program Construction: 10th International Conference, MPC 2010, Québec City, Canada, June 21-23, 2010. Proceedings*, chapter Lucy-n: a n-Synchronous Extension of Lustre, pages 288–309. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
<http://www.springer.com/us/book/9783642133206>.
- [MR01] F. Maraninchi and Y. Rémond. Argos: An automaton-based synchronous language. *Computer Languages*, 27(27):61–92, 2001.
- [MS15] Christian Motika and Steven Smyth. Updates on SCCharts. Presentation at the 22th International Open Workshop on Synchronous Programming (SYNCHRON’15), Kiel, Germany, December 2015.
- [MSF⁺11] Christian Motika, Miro Spönemann, Hauke Fuhrmann, Christoph Krüger, John Julian Carstens, and Reinhard von Hanxleden. KIELER Actor Oriented Modeling (KAOM). Poster presented at 9th Biennial Ptolemy Miniconference (PTCONF’11), Berkeley, CA, USA, February 2011.
- [MSvH14] Christian Motika, Steven Smyth, and Reinhard von Hanxleden. Compiling SCCharts — A case-study on interactive model-based compilation. In *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014)*, volume 8802 of LNCS, pages 443–462, Corfu, Greece, October 2014.
<http://dl.acm.org/citation.cfm?id=2770188>.
- [MSvHM13] Christian Motika, Steven Smyth, Reinhard von Hanxleden, and Michael Mendler. Sequentially Constructive Charts (SCCharts). Poster presented at 10th Biennial Ptolemy Miniconference (PTCONF’13), Berkeley, CA, USA, 7 November 2013.

Bibliography

- [MvH14] Christian Motika and Reinhard von Hanxleden. Light-weight Synchronous Java (SJL) — an approach for programming deterministic reactive systems with java. *Journal of Computing, Special Issue on Software Technologies for Embedded and Ubiquitous Systems*, 97(3):281–307, 2014.
<http://link.springer.com/article/10.1007/s00607-014-0416-7>.
- [MvHH12] Christian Motika, Reinhard von Hanxleden, and Mirko Heinold. Synchronous Java: Light-weight, deterministic concurrency and preemption in Java. Technical Report 1213, Kiel University, Department of Computer Science, October 2012. ISSN 2192-6247.
- [MvHH13] Christian Motika, Reinhard von Hanxleden, and Mirko Heinold. Programming deterministic reactive systems with Synchronous Java (invited paper). In *Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2013)*, IEEE Proceedings, Paderborn, Germany, 17/18 June 2013.
- [MvHT09] Michael Mendler, Reinhard von Hanxleden, and Claus Traulsen. WCRT Algebra and Interfaces for Esterel-Style Synchronous Processing. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'09)*, Nice, France, April 2009.
- [Nas15] Stanislaw Nasin. Transformation from SCCharts to Esterel. Master thesis, Kiel University, Department of Computer Science, October 2015.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sna-mt.pdf>.
- [NTI⁺14] Stefan Naujokat, Louis-Marie Traonouez, Malte Isberner, Bernhard Steffen, and Axel Legay. *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 6th International Symposium, ISoLA 2014*,

- Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, chapter Domain-Specific Code Generator Modeling: A Case Study for Multi-faceted Concurrent Systems, pages 481–498. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.
<http://www.springer.com/de/book/9783662452301>.
- [Obj15] Object Management Group. *Specification of the Unified Modeling LanguageTM* (UML[®]), June 2015.
<http://www.omg.org/spec/UML/2.5/>.
- [OW02] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen (Sav Informatik) (German Edition)*. Spektrum Akademischer Verlag, 4. aufl. edition, January 2002.
- [PBEB07] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, May 2007.
- [Pei15] Lars Peiler. Modeling simulations of autonomous, safety-critical systems. Bachelor thesis, Kiel University, Department of Computer Science, September 2015.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lpe-bt.pdf>.
- [PLDM02] Jeffrey Palm, Han Lee, Amer Diwan, and J. Eliot B. Moss. When to use a compilation service? *SIGPLAN Not.*, 37(7):194–203, June 2002.
<http://doi.acm.org/10.1145/566225.513862>.
- [PM03] Gergely Pintér and István Majzik. Program Code Generation based on UML Statechart Models. *Periodica Polytechnica*, pages 187–204, 2003.
- [Pto14] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.
<http://ptolemy.org/books/Systems>.
- [PTvH06] Steffen Prochnow, Claus Traulsen, and Reinhard von Hanxleden. Synthesizing Safe State Machines from Esterel. In *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages*,

Bibliography

Compilers, and Tools for Embedded Systems (LCTES'06), Ottawa, Canada, June 2006.

- [PvH07] Steffen Prochnow and Reinhard von Hanxleden. Enhancements of Statechart-modeling—the KIEL environment. In *Proceedings of the Design, Automation and Test in Europe University Booth (DATE'07)*, Nice, France, April 2007. With accompanying
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/date07-spr-poster.pdf>.
- [Rat15] Karsten Rathlev. From Esterel to SCL. Master thesis, Kiel University, Department of Computer Science, March 2015.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/krat-mt.pdf>.
- [Rie13] Leanna Rierson. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, January 2013.
- [RMvH11] Ulf Rüegg, Christian Motika, and Reinhard von Hanxleden. Interactive transformations for visual models. In *3rd Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2011) at conference INFORMATIK 2011, GI-Edition – Lecture Notes in Informatics (LNI)*, Berlin, Germany, October 2011. Bonner Köllen Verlag.
- [RNHL99] John Reekie, Stephen Neuendorffer, Christopher Hylands, and Edward A. Lee. Software practice in the Ptolemy project. Technical report, Gigascale Semiconductor Research Center, April 1999.
<http://ptolemy.eecs.berkeley.edu/publications/papers/99/sftwareprac>.
- [RSM⁺15] Karsten Rathlev, Steven Smyth, Christian Motika, Reinhard von Hanxleden, and Michael Mendler. SCEst: Sequentially Constructive Esterel. In *Proceedings of the 13th ACM-IEEE*

International Conference on Formal Methods and Models for System Design (MEMOCODE'15), Austin, TX, USA, September 2015.

- [RSM⁺16] Francesca Rybicki, Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. Interactive model-based compilation continued — interactive incremental hardware synthesis for SCCharts. In *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)*, LNCS, 2016.
- [RSS⁺13] Ulf Rüegg, Christian Schneider, Christoph Daniel Schulze, Miro Spönemann, Christian Motika, and Reinhard von Hanxleden. Light-weight synthesis of Ptolemy diagrams with KIELER. Presentation at the Tenth Biennial Ptolemy Miniconference, Berkeley, CA, USA, 7 November 2013.
- [Rue11] Ulf Rueegg. Interactive transformations for visual models. Bachelor thesis, Kiel University, Department of Computer Science, March 2011.
- [Rus10] David Russell. *Introduction to Embedded Systems: Using ANSI C and the Arduino Development Environment*. Morgan and Claypool, 2010.
- [Ryb16] Francesca Rybicki. Interactive incremental hardware synthesis for SCCharts. Bachelor thesis, Kiel University, Department of Computer Science, March 2016.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/fry-bt.pdf>.
- [SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF – Eclipse Modeling Framework*. Eclipse Series. Addison-Wesley, second edition, 2009.
- [Sch03] Martin Schoeberl. Using a java optimized processor in a real world application. In *in Proceedings of the First Workshop on*

Bibliography

Intelligent Solutions in Embedded Systems (WISES 2003, pages 165–176, 2003.

- [Sch09a] Matthias Schmeling. ThinKCharts—the thin KIELER SyncCharts editor. Student research project, Kiel University, Department of Computer Science, September 2009.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/schm-st.pdf>.
- [Sch09b] Klaus Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.
- [Sch10] Klaus Schneider. The synchronous programming language Quartz. Internal report, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2010.
- [Sch11] Christian Schneider. On integrating graphical and textual modeling. Diploma thesis, Kiel University, Department of Computer Science, February 2011.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/chsch-dt.pdf>.
- [SFvH09] Miro Spönemann, Hauke Fuhrmann, and Reinhard von Hanxleden. Automatic layout of data flow diagrams in KIELER and Ptolemy II. Technical Report 0914, Kiel University, Department of Computer Science, July 2009.
- [SL12] Chris Shaver and Edward A. Lee. *Model Driven Engineering Languages and Systems: 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30–October 5, 2012. Proceedings*, chapter The Coroutine Model of Computation, pages 319–334. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
http://link.springer.com/chapter/10.1007/978-3-642-33666-9_21.

- [SMSR⁺15] Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, Nis Boerge Wechselberg, Carsten Sprung, and Reinhard von Hanxleden. SCCharts: the railway project report. Technical Report 1510, Kiel University, Department of Computer Science, August 2015. ISSN 2192-6247.
- [SMvH15] Steven Smyth, Christian Motika, and Reinhard von Hanxleden. A data-flow approach for compiling the sequentially constructive language (SCL). In *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2015)*, Pörschach, Austria, 5-7 October 2015.
- [Smy13] Steven Smyth. Code generation for sequential constructiveness. Diploma thesis, Kiel University, Department of Computer Science, July 2013.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ssm-dt.pdf>.
- [SR14] Alexander Schulz-Rosengarten. Framework zum tracing von emf-modelltransformationen. Bachelor thesis, Kiel University, Department of Computer Science, March 2014.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/als-bt.pdf>.
- [SSBD99] Sanjit A. Seshia, R. K. Shyamasundar, A. K. Bhattacharjee, and S. D. Dhodapkar. A translation of Statecharts to Esterel. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 volume 2— World Congress on Formal Methods*, volume 1709 of LNCS, pages 983–1007. Springer-Verlag, 1999.
- [SSM⁺13] Miro Spönemann, Christoph Daniel Schulze, Christian Motika, Christian Schneider, and Reinhard von Hanxleden. KIELER: building on automatic layout for pragmatics-aware modeling (showpiece). In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'13)*, San Jose, CA, USA, September 2013.

Bibliography

- [SSvH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. Just model! – Putting automatic synthesis of node-link-diagrams into practice. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'13)*, pages 75–82, San Jose, CA, USA, September 2013.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/vlhcc13.pdf>.
- [SSvH14] Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. Drawing layered graphs with port constraints. *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout*, 25(2):89–106, 2014.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/jvlc13.pdf>.
- [Ste97] Bernhard Steffen. Unifying models. In *STACS 97, 14th Annual Symposium on Theoretical Aspects of Computer Science, Lübeck, Germany*, pages 1–20, March 1997.
<http://dl.acm.org/citation.cfm?id=695485>.
- [Tan09] Andrew S. Tanenbaum. *Modern Operating Systems*. Pearson, 3rd edition edition, 2009.
- [TAvH10] Claus Traulsen, Torsten Amende, and Reinhard von Hanxleden. Compiling SyncCharts to Synchronous C. Technical Report 1006, Kiel University, Department of Computer Science, Kiel, Germany, July 2010.
- [Tra07] Claus Traulsen. The Kiel Reactive Processor—reactive processing beyond the KEP. Presentation at the 14th International Open Workshop on Synchronous Programming (SYN-CHRON'07), Bamberg, Germany, November 2007.
- [Tra10] Claus Traulsen. *Reactive Processing for Synchronous Language and its Worst Case Reaction Time Analysis*. PhD thesis, Kiel University, Faculty of Engineering, 2010.

- [vdB94] Michael von der Beeck. A comparison of Statecharts variants. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 128–148. Springer-Verlag, 1994.
- [vH09] Reinhard von Hanxleden. SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. In *Proc. Int’l Conference on Embedded Software (EMSOFT’09)*, pages 225–234, Grenoble, France, October 2009. ACM.
- [vHDM⁺13a] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. Compiling SCCharts to hardware and software. Presentation at Synchronous Programming (SYNCHRON’13), Schloss Dagstuhl, Germany, November 2013.
- [vHDM⁺13b] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. SCCharts: Sequentially Constructive Statecharts. Presentation at Synchronous Programming (SYNCHRON’13), Schloss Dagstuhl, Germany, November 2013.
- [vHDM⁺13c] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. Technical Report 1311, Kiel University, Department of Computer Science, December 2013. ISSN 2192-6247.
- [vHDM⁺14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. ACM SIGPLAN Conference on Programming Language*

Bibliography

Design and Implementation (PLDI'14), Edinburgh, UK, June 2014. ACM.

- [vHLMF12a] Reinhard von Hanxleden, Edward A. Lee, Christian Motika, and Hauke Fuhrmann. Multi-view modeling and pragmatics in 2020 — Position paper on designing complex cyber-physical systems. In *Pre-Proceedings of the 17th International Monterey Workshop on Development, Operation and Management of Large-Scale Complex IT Systems*, Oxford, UK, 19–21 March 2012.
- [vHLMF12b] Reinhard von Hanxleden, Edward A. Lee, Christian Motika, and Hauke Fuhrmann. Multi-view modeling and pragmatics in 2020 — Position paper on designing complex cyber-physical systems. In *Proceedings of the 17th International Monterey Workshop on Development, Operation and Management of Large-Scale Complex IT Systems, LNCS*, volume 7539, Oxford, UK, December 2012.
- [vHMA⁺13a] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, and Owen O'Brien. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. In *Proc. Design, Automation and Test in Europe Conference (DATE'13)*, pages 581–586, Grenoble, France, March 2013. IEEE.
- [vHMA⁺13b] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, and Owen O'Brien. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. In *Proc. Design, Automation and Test in Europe Conference (DATE'13)*, pages 581–586, Grenoble, France, March 2013. IEEE. Long version: Technical Report 1308, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, August 2013, ISSN 2192-6247.

- [vHMA⁺13c] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O’Brien, and Partha Roop. *Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation*. Technical Report 1308, Kiel University, Department of Computer Science, August 2013. ISSN 2192-6247.
- [vHMA⁺14] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O’Brien, and Partha Roop. *Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation*. *ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design*, 13(4s):144:1–144:26, July 2014.
- [Wec15] Nis B. Wechselberg. *Model railway 4.0*. Master thesis, Kiel University, Department of Computer Science, March 2015.
<http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nbw-mt.pdf>.
- [YAY⁺08] Simon Yuan, Sidharta Andalam, Li Hsien Yoong, Partha S. Roop, and Zoran Salcic. *STARPro—a new multithreaded direct execution platform for Esterel*. In *Proceedings of Model Driven High-Level Programming of Embedded Systems (SLA++P’08)*, Budapest, Hungary, April 2008.

Acknowledgments

I would like to thank my “Doktorvater” *Reinhard von Hanxleden* for giving me the opportunity to graduate, for many advices but also for much freedom and his trust. I am grateful that he introduced me into the synchronous community. He made it possible for me to participate at the annual SYNCHRON workshops. He also arranged various journeys to the UC Berkeley which enabled very fruitful, interesting, and heavily inspiring discussions, meetings, and programming sessions with *Edward Lee* and members of his great, impressive group. I would like to thank *Edward Lee*, *Christopher Brooks*, *Chris Shaver*, and *Stavros Tripakis* for their very warm welcome and the inspiring time whenever I was visiting Berkeley. The *synchronous community* has much encouraged my work. Therefore I want to thank all members of the synchronous team. In particular I want to thank *Michael Mendler*, *Partha Roop*, and *Claus Traulsen* for the discussions and their comments and suggestions on my work. I am grateful to *Willem-Paul de Roever* and *Jens Schönborn* for giving me the opportunity to gain first experience in teaching and research while working as a student research assistant. I thank *Hauke Fuhrmann*, my former advisor of my diploma thesis. He helped me getting started when I began to graduate. I am very pleased for some outstanding students that contributed to the work on SCCharts and its compilation and who made me a very thankful advisor. These are *John Carstens*, *Achim Bleidiesel*, *Ulf Rüegg*, *Steven Smyth*, *Nis Wechselberg*, and *Francesca Rybicki* — it was a pleasure to work with you. I thank my colleagues, in particular *Christian Schneider* and *Steven Smyth* for their technical and personal support at work and beyond. Notably, I like to thank *Steven Smyth*, *Nis Wechselberg*, and *Alexander Schulz-Rosengarten* for keeping my work load balanced, particularly in the final phase of my dissertation. I thank *Steven Smyth* and *Francesca Rybicki* for their very valuable proof reading. I do not want to forget thanking my outstanding *parents* for their constant and reliable support whenever it was needed. This also counts to my *sister* who gave me lots of English improvement suggestions. I thank the *Lord*.