INSTITUT FÜR INFORMATIK

Watch Your Compiler Work
Compiler Models and Environments

Steven Smyth and Alexander Schulz-Rosengarten
and Reinhard von Hanxleden

Bericht Nr. 1806
July 2018
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Department of Computer Science
Kiel University
Olshausenstr. 40
24098 Kiel, Germany

Watch Your Compiler Work
Compiler Models and Environments

Steven Smyth and Alexander Schulz-Rosengarten and
Reinhard von Hanxleden

Report No. 1806
July 2018
ISSN 2192-6247

E-mail: {ssm,als,rvh}@informatik.uni-kiel.de

An abridged version of this work will be published at the proceedings of
the 8th International Symposium On Leveraging Applications of
Formal Methods, Verification and Validation,
Limassol, Cyprus, Oct 2018.

Abstract

We can observe many similarities between classical programming paradigms and model-driven engineering. A chain of model-to-model transformations often prescribes a particular work process, while executing such a chain generates a concrete instance of this process. Modeling the entire development process itself on a meta-model level extends the possibilities of the model-based approach to guide the developer. Besides refining tools for model creation, this kind of meta-modeling also facilitates debugging, optimization, and prototyping of new compilations. A compiler is such a process system. In this paper, we share the experiences gathered while we worked on the model-based reference compiler of the KIELER SCCharts project and ideas towards a unified view on similar prescribed processes. We exemplify our approach in two case studies.

Contents

1 Introduction 1

2 An Abstract User Story 3

3 Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) Compilation 7

4 Observing Compiler Optimizations 13

5 Curing Compilation Problems 18

6 Related Work 21

7 Conclusions 22

1 Introduction

In our previous publications towards a unified view of modeling and programming [10,15] we focused on the program/model that should be compiled. Using the model-based approach, we showed how a model-based compiler can transform a program to the desired target platform step-by-step while preserving the intermediate results. The approach, named SLIC for Single-Pass Language-Driven Incremental Compilation, was used to create the compiler for the synchronous language SCCharts [22].

While working on the model-based compiler, we since recognized that providing meaningful guidance for and resources to the developer does not solely depend on the artifact that should be compiled, but also on the process which performs the transformations. Instead of using a compiler that is particularly developed for a specific use case, we built upon the experiences we gained during the development of the SLIC approach to model the entire compilation process. Modeling the process provides us with new possibilities to aid the developer in their pursuit to create complex products, such as (1) arbitrary annotated intermediate models, and (2) the ability to change the compilation model at any time. We will demonstrate our generic framework and two technical use cases in the following sections.

To illustrate the process and to continue the story told previously [10, 15], we use the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[1] SCCharts language implementation as running example. KIELER is an academic open-source project that serves as a proof-of-concept platform. However, the approach presented here is not restricted to the SCCharts compiler, SCCharts, or KIELER as every system of consecutive processes may be modeled and executed in a similar way. For example, our reference implementation within the KIELER project also includes compilers for languages such as C, Esterel, and various domain-specific languages as well as non-compilation tasks such as simulation of compiled artifacts.

Contributions and Outline

We give an exemplary, abstract user story on classical programming and modeling in Chap. 2 and compare the concepts to our generic process systems. We explain the abstract system models within the KIELER project in Chap. 3. This demonstration serves as an example for similar process system modeling and is not restricted to the work done within KIELER. Then, we present two case studies for generic compilation models that are used in the SCCharts compilations.

[1] http://rtsys.informatik.uni-kiel.de/kieler

1. We will show how to add and inspect the results of additional optimizations to the existing SCCharts compilation chain interactively in Chap. 4. As example we include the commonly applied Sparse Conditional Constant Propagation (SCCP) to the compilation, but other frequently used optimizations such as copy propagation and smart register allocations are also possible.

2. The second case-study (Chap. 5) illustrates how the interactive compilation approach can be used to spot and solve compilation problems. As example, we focus on a common compilation problem of synchronous languages, namely schizophrenia. Even in cases where a compilation cannot be fixed, the compiler framework can help to guide the modeler.

The case-studies also show how to use the framework to gather experimental results.
Subsequent to related work in Chap. 6 we conclude in Chap. 7.

2 An Abstract User Story

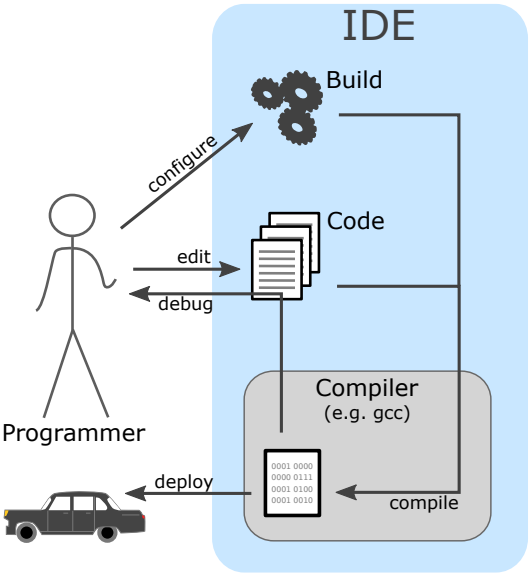
In this section we take a closer look at three alternative development processes sketched in Fig. 1. We assume that the developer uses a common Integrated Development Environment (IDE) to work on a particular software project. Usually, the build process and/or project has to be configured by either the developer themselves or by another build expert. Typically, the developer works directly on the artifact in question. However, the work foci differ.

Fig. 1a gives an abstract view on a **classical programming** development process, which is fairly straightforward. The developer often has to be a programming expert and generally also configures the build process. While they usually work on one file at a time, they must keep an eye on the whole project, which is usually a collection of files, because it might influence the compilation. When they complete a development step, they issue a compilation command. An embedded (often external) compiler then compiles the source files to binary code that can be executed or embedded elsewhere if the source code is error free. Errors and warnings are fed back to the editor inside the IDE. They mark the erroneous line and give more or less processed information about the actual error or warning.

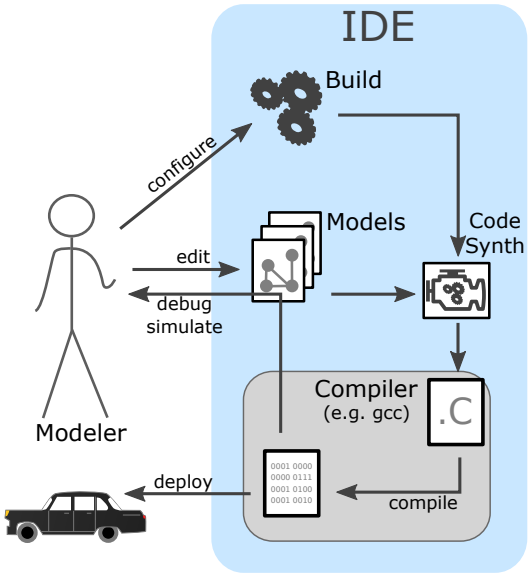
The **classical modeling** work-flow (Fig. 1b) looks actually quite similar. The modeler has to configure their project and can explore the project's files. Instead of editing a text file, the modeler usually works on a domain-specific, often graphical, model. The IDE uses an integrated code synthesis, such as the Run-Time Workshop (RTW) in Matlab/Simulink, to synthesize code. Similar to the classical programming paradigm, as soon as a development step is finished, the source models are compiled to a classical, general purpose language, such as C. Afterwards, they are compiled to binary code like before, with the addition that the user feedback often includes some sort of simulation. Here it depends on the concrete design choices if the simulation runs inside the RTW or on the compiled product.

Although the development processes are quite similar, there is a subtle shift in the focus on the developer. In the first case, the developer has to be a programming expert, whereas the models in the second case are typically maintained by a domain expert. However, even in the second case programming experts are sometimes required to aid the modeler with special requirements or IDE extensions.

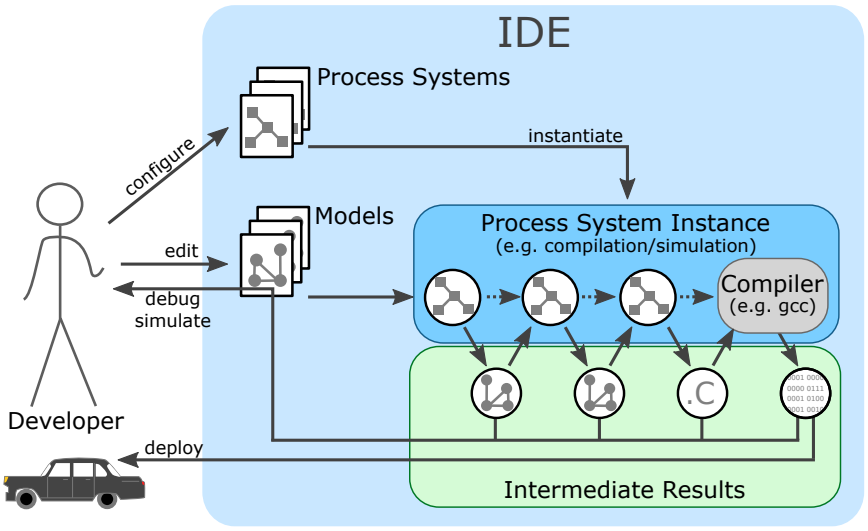
Fig. 1c depicts the more interactive approach that we advocate here. With *interactive process systems*, operating procedures, such as compilation or simulation, can be created and modified by the developer. Here, these process systems are simply models just like the working artifact, but perhaps models of another meta-model. When the modeler wants to compile (resp. simulate) the actual status of the model, the respective process



(a) An abstract view on classical programming development process



(b) An abstract view on classical modeling development process



(c) Development story with interactive process system instances

Figure 1: Three alternative development processes

system gets instantiated. Afterwards, the issued command can be processed by that system's instance. The feedback, including errors, *intermediate*, and *final results*, is directly available as individual model instances of appropriate meta-models. They can be inspected by the modeler or used as source for further systems. In the figure, we see an instantiated process system. The artifact is processed sequentially by single processors, e. g., model-to-model transformations, of the instance. All intermediate steps are observable. Eventually, the user wants to deploy binary code and one of the intermediate results may be general purpose source code that can be sent to an external compiler as

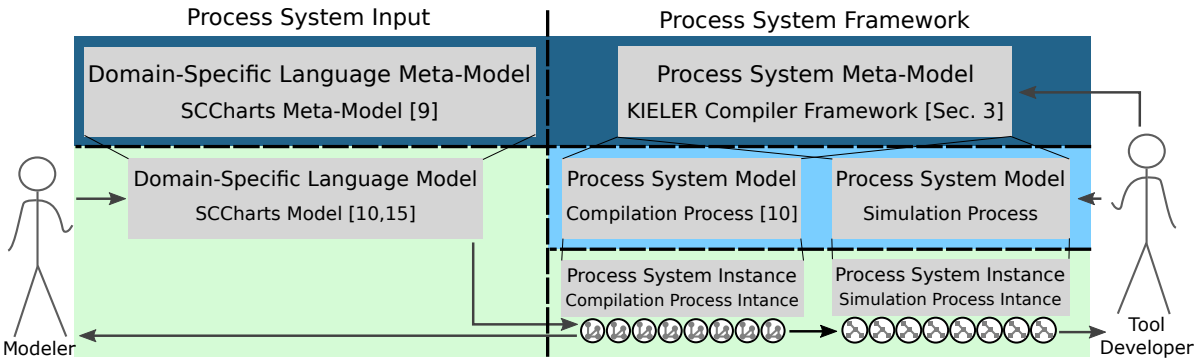


Figure 2: Process system's different model layer and user roles

before. Conceptually, the compiler call is just another process in the system's sequential chain.

Note that this approach is agnostic to the question whether the intermediate results (or, in fact, the original model) are graphical or textual. In the case of graphical syntax, a key enabler to be able to represent the artifact is the integration of automatic layout facilities into the modeling tool. In KIELER, we make use of the Eclipse Layout Kernel (ELK)[1] to synthesize all graphical views. We argue that this is also an example of *pragmatic modeling* concepts [6, 23], which aim to enhance modeler productivity by allowing to seamlessly switch between textual and graphical representations tailored to specific use cases. To quote a practitioner: "In our experience over many years my colleagues and I concluded that textual modeling is the only practical way, but that a graphical view of the models is a must-have as well. Your technology closes exactly that gap."[2]

The interactivity of the approach becomes apparent in the ability to observe all intermediate steps, to run system instances as they are needed and to create new or change existing systems all during run-time. There is no need to go through long re-build and re-start cycles. As described before, a process system can basically perform any kind of job. The example depicts a compilation and a simulation system. Here, technically, the term interactive subsumes the dynamic nature of the approach, meaning that instances of systems are generated dynamically as they are needed. These instances carry dynamic properties on their own and live as long as they are needed. This also resembles the classical class–object hierarchy of the object-orientated paradigm.

Due to the interactivity of the approach, tool developers and modelers can easily create, explore, and modify different aspects of the whole development process. The difference is not disparate work-flows, but the diverse work-flow artifacts that are being worked on. Fig. 2 shows the different layers of models and the two main roles of users. On the left side, the modeler mainly works on the system's input, e. g., a particular model in a specific Domain-specific language (DSL). The model's meta-model also belongs to the

[1] https://www.eclipse.org/elk/

[2] Dr. Andreas Seibel, BSH Hausgeräte GmbH, E-Mail from Oct. 6, 2017.

system's input, but is usually outside of the modeler's scope w.r.t. of making changes during a particular project. In the example in the figure, the modeler works on an SCCharts model, whose syntax is defined in the corresponding meta-model. Again, it is not important here whether the syntax is textual or graphical. This decision can be made as the case arises depending on the preference of the domain expert. At this, the form of editor must not be the same as the graphical view of the model. Transient view and automatic layout technologies [16] may help the modeler to explore the model without getting distracted by tedious tasks, such as placement and alignment of graphical elements.

On the framework's side on the right, there is also the framework's meta-model for defining system models. Derived from this, different systems can be created that hold the necessary instructions. These systems can be instantiated to be applied on a specific artifact. In the example shown, the created SCCharts model is fed into an compilation system instance. During compilation, several observable intermediate results are created. The result of the whole context also serves as input for a simulation instance.

In general, the modeler will be more interested in the actual project's model and the systems' results, whereas the tool developer's focus will lie on the systems and the underlying framework, including the relevant meta-models. However, both can utilize all aspects of the development process to drive their work. For example, the modeler may also change a particular system to toggle optimizations if necessary. More obviously, the tool developer can use different model inputs to test and extend the framework. This could also lead to closer feedback loops between domain experts and tool developers.

3 KIELER Compilation

In the KIELER Compiler (KiCo) the smallest compilation unit is called a *processor*. We moved away from the generic term *transformation* to emphasize that a processor does not have to perform a transformation. Instead processors are categorized into *transformer*, *optimizer*, and *analyzer* to specify their role. A variety of tasks can be implemented as processors, such as model-to-model (M2M) transformations, optional optimizations, and, e. g., object counting, but this should be restricted to this atomic task to facilitate modularity and reuse.

A list of processors forms a *process system*. These systems describe a single compilation from a certain source type down to the desired target. When compared to the object-orientated programming paradigm, process systems can be seen as classes. They can be instantiated to perform a task for a concrete artifact. In the previous publications we described two compilation approaches, the netlist-based and the priority-based approaches, for SCCharts [10,22]. Each of these is an own process system, more specifically a *compilation system*.

When a system is instantiated, an instance for every processor within the system is created. A *processor instance* is then connected to a *source environment* from which it will receive input data and a *target environment* to work on and store data for the next processors. The simplest system possible is shown in Fig. 3. It consists of a single processor with its corresponding source and target environments. Once the system is fully instantiated, a new *compilation context* (gen. process context) exists, which can be used to compile an artifact. While the context, including all environments and all data, is observable during compilation, it will remain accessible even after the compilation finished until discarded, so that all data and results can be inspected as long as desired.

Figure 3: The atomic compilation unit, a processor p, receives a source and a target environment when instantiated.

Conceptionally, the developer is free to choose the nature of their environments. In KIELER we use typed, but arbitrary data storages. Hence, processors may store arbitrary (ancillary) data in the environments, but have a form of type-safety when accessing it.

The developer does not have to bother with all the instances and environments. The KiCo framework will do most of the work. In general, when invoking a compilation programmatically, one only needs two lines of code. First, a context has to be created. The context needs to know which system model it should use and on which artifact the compilation should be invoked. Once the context is created, the compilation can be executed. List. 1 shows an excerpt from the KIELER project where a compilation is started

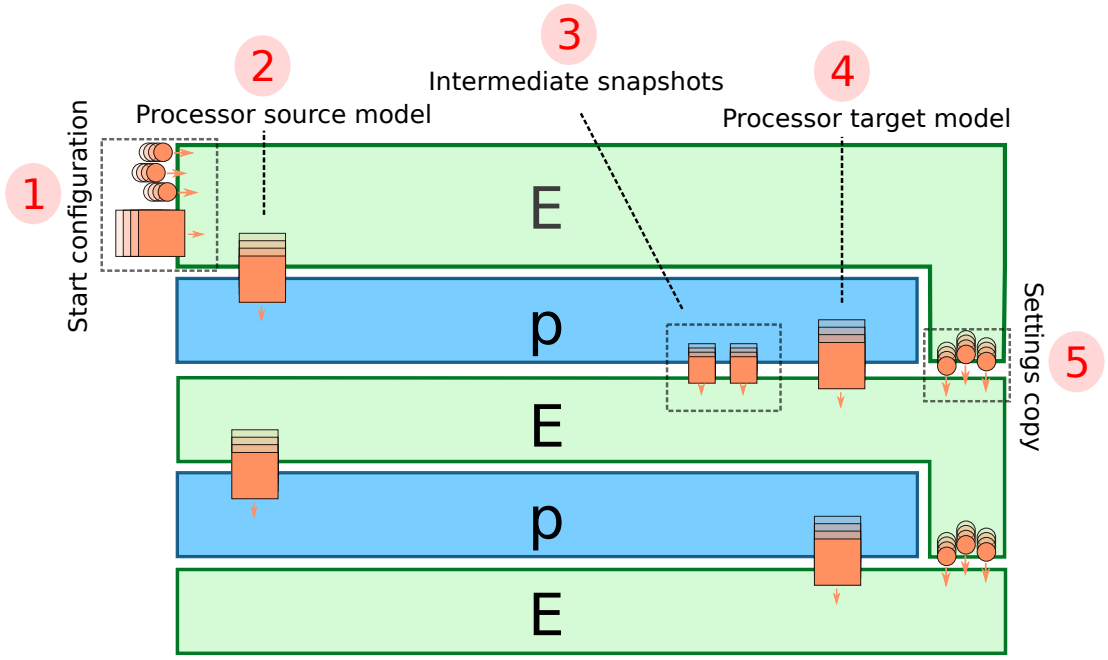


Figure 4: Concept of a compilation context with two processors.

asynchronously as soon as the user presses on a particular button. The programmer could make adjustments to the context before the compile method is executed, but in this case, it is not necessary.

```
1 val context = Compile.createCompilationContext(view.activeSystem, model)
2 context.compileAsynchronously
```

Listing 1: Compilation invocation excerpt from the KIELER project

Usually a compilation system includes more than one processor. Fig. 4 shows how processors interact with their environments to orchestrate the entire compilation. As described before, once a context has been created, it needs an input artifact and can be configured if necessary. The first environment in the context receives the *start configuration* 1 as can be seen in the figure. After the invocation of the compilation, the first processor begins its work. It fetches the model from its source environment 2 and begins its computations. That model is the *source model* from the processor's perspective. While working on the model, the processor can do several snapshots of the current state and store them in its *target environment* 3. These intermediate states can be inspected during or after the compilation. At the end of the processor's job, the result is saved 4. In the example, the shapes in the figure indicate that the result is of the same metamodel as the source model. However, any type can be used. E. g., as targets are often other programming-languages, the backends usually give simple text as results. Once the processor terminates, the next processor starts its job. From its perspective, the former target environment now becomes the source environment and the processor can work on the next one. The framework takes care that all settings, model references, and

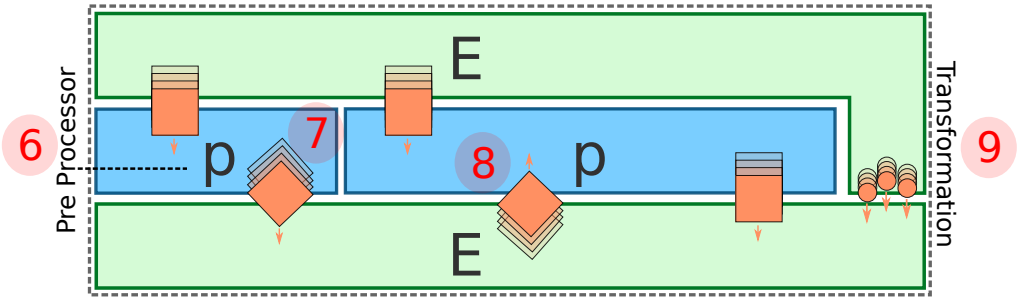


Figure 5: To save resources several processors can be grouped together. Generally, everything that happens between two environments is commonly called a transformation.

additional auxiliary data get copied to the new environments if necessary [5].

To facilitate modularity and to be less consuming w.r.t. resources, processors often perform pre- or post-processing jobs for transformers without the need of dedicated environments as depicted in Fig. 5. Hence, a processor can run with the same environments as another one [6]. In the example, the job saves a second model with a different metamodel in the environment [7]. This secondary model may store ancillary data (e. g. loop information from a loop analyzer), which can be picked up by subsequent processors [8]. Usually, what is commonly called *transformation* is everything that happens between the source and the target environments [9]. The result that is stored in the last environment represents the result of the whole compilation.

Note that pre- or post-processors also store these data in the target environment of the main processos as they are not allowed to change the source environment. However, the developer is not required to handle these inputs differently as the framework will ensure correct accesses. In fact, technically, KiCo processors internally always only work on the target environment. The framework automatically creates a copy from the source environment before a processor is called.

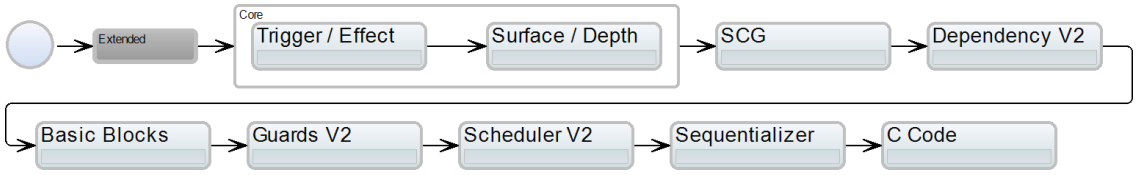
To be even more resource-saving, a compilation can be set to *in-place*. Compiling in-place does not create new model instances to work on. The processors all work on the same models and hence, intermediate results are only observable during compilation and only one at a time. At the end of the compilation, only the final result remains. Conceptually, this would also look like the schema in Fig. 5 where only two environments exist and all processor instances live in between.

SCCharts Compilation

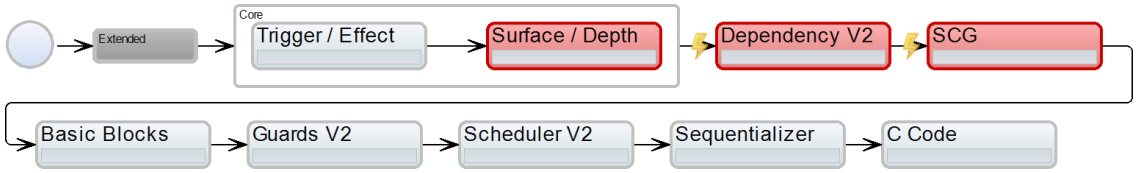
The default version of the netlist-based compilation system that we use in the case-studies uses 33 processors. List. 2 shows a shortened description for the netlist-based compilation of SCCharts. Every processor has its own unique identifier. However, compilation systems are often composed of other systems, which can be referenced. Here, the downstream compilation builds upon the standard high-level SCCharts compilation

```
1  public system de.cau.cs.kieler.sccharts.netlist
2      label "Netlist-based Compilation"
3
4  system de.cau.cs.kieler.sccharts.extended
5  system de.cau.cs.kieler.sccharts.core
6  de.cau.cs.kieler.sccharts.scg.processors.SCG
7  post process de.cau.cs.kieler.scg.processors.threadAnalyzer
8  de.cau.cs.kieler.scg.processors.dependency
9  de.cau.cs.kieler.scg.processors.basicBlocks
10 post process de.cau.cs.kieler.scg.processors.expressions
11 de.cau.cs.kieler.scg.processors.guards
12 de.cau.cs.kieler.scg.processors.scheduler
13 de.cau.cs.kieler.scg.processors.sequentializer
14 de.cau.cs.kieler.scg.processors.codegen.c
```

Listing 2: Model description of the netlist-based SCCharts compilation



(a) SCCharts netlist-based compilation system. In this view, the Extended system is collapsed and the Core system is expanded.



(b) While modeling, errors, such as type incompatibility, can be highlighted immediately.

Figure 6: Example of an automatically generated graphical (view) of a compilation system

(line 4–5) and nine further processors identified by their identifiers.

As these descriptions define compilation models interactively, we use concepts such as transient views [16] to visualize the system graphically and, if necessary, point to problems such as unknown processors or type incompatibility between processors. Here, *Interactively* means that we can inspect, change, and save the model during runtime to invoke altered compilation runs without the need of long re-configure and re-start cycles. Fig. 6a shows the automatically generated graphical representation of the netlist-based compilation system during editing. This view is synchronized with the editor of the model's description and instantaneously re-generated upon change. The referenced

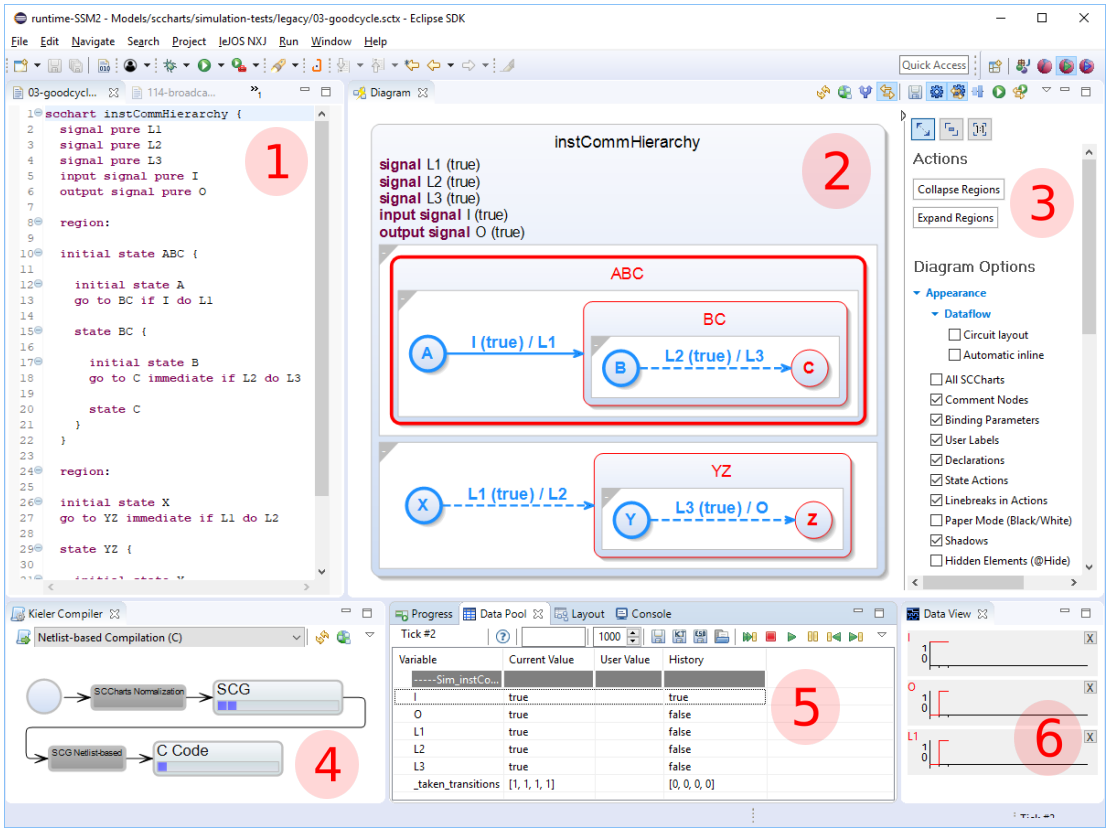


Figure 7: Complete example of a running KIELER instance during simulation.

high-level SCCharts systems can be expanded and collapsed for readability. Problems appear in red. The generated views are also used as control panel in the KIELER project to invoke the compilations and to select intermediate results.

In the example depicted in Fig. 6a, the Surface / Depth processor creates an SCCharts model which is then transformed to its corresponding Sequentially Constructive Graph (SCG), a sequentially constructive variant of a control-flow graph, by the SCG processor. The subsequent Dependency processor expects an SCG as input. If one would swap the SCG and Dependency processors, the compilation chain becomes type incompatible as depicted in Fig. 6b.

SCCharts Run-Time Example

Fig. 7 shows a complete example of a running KIELER instance during simulation. In the SCCharts editor tool, the abstract model is described with an textual syntax 1. A graphical view of the model is instantaneously generated by the transient view framework [16] 2. The user can further influence the visualization of the presented data via options on the right sidebar 3. However, these options consist mainly of rather coarse convenience settings to set the current focus to specific points of interests. 4 – 6 show

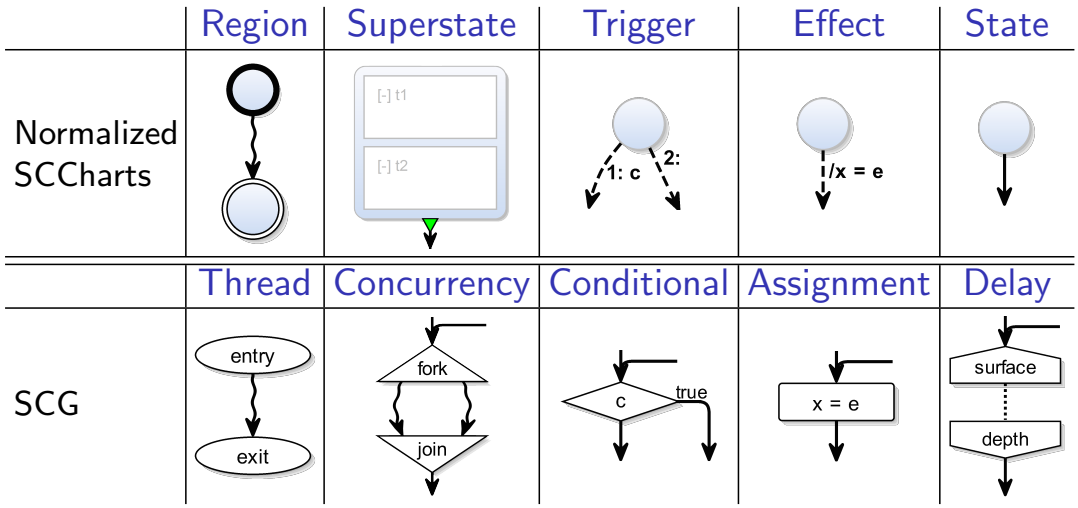


Figure 8: Mapping from SCCharts to a semantically equivalent Sequentially Constructive Graph (SCG)

examples of different information views. These can be configured (and saved per perspective) individually. Together with the transient live visualization 2, they resemble the systems and intermediate result regions from the previous figures. The selected compilation system is depicted in 4. A view to manipulate the running simulation is open in 5. Selected data observers can be inspected in 6. Note that information of the running simulation is visible in the model diagram 2, the simulation view 5, and the observers 6 simultaneously. The variable states and current active model elements can be highlighted directly in the model. The user can input new environment settings in the simulation view. Here, one can also control single forward and backward steps of the simulation. Furthermore, the actual and past data of selected variables can also be visualized in the data observer 6.

SCCharts To SCG Mapping

To understand the following two examples, we briefly recapitulate the building blocks of SCCharts. Eventually, all SCCharts get transformed to their normalized form. Here, they only include combinations of the five kernel patterns listed in Fig. 8, namely regions, superstates, triggers, effects, and states. In the downstream compilation, a normalized SCCharts is transformed into its SCG. The figure illustrates that the five kernel pattern directly correspond to the five elementary constructs of an Sequentially Constructive Graph (SCG), namely threads, concurrency, conditionals, assignments, and delays. Please consider the previous works on SCCharts and the SCCharts compilation for more details [10, 15, 22].

4 Observing Compiler Optimizations

The process of compiling programs may consist of more than directly producing machine code. It usually includes various analyses, restructuring and optimizations often based on different intermediate representations. Using or developing such analyses and optimizations clearly benefits from the possibility to inspect intermediate results in an open model-based compiler framework as presented here. Giving access to the results of each compilation step, in different granularity and detail, improves the understanding of the processor and its effects, not only for debugging purposes.

Sparse Conditional Constant Propagation (SCCP) [24] is a powerful compiler optimization for imperative programs based on the Single Static Assignment (SSA) form [14]. It efficiently detects and propagates constant variable values in combination with a dead code elimination, which is more effective than executing these optimizations separately. SCCP is performed on an intermediate representation in SSA form because SSA provides a clear and useful structure for variable definitions and usages. The idea of SSA is to rename each variable such that no two assignments define the value of the same variable. To keep the semantics of the original program, every time the definition scopes of two renamed variables merge, for example at the end of two conditional branches, an SSA-function, here a Φ-function, is placed to merge the values of the variables defined in these branches and create a new definition. This way, each reader to a variable has only a single reaching definition giving the value of the variable. An SSA intermediate representation is created based on the algorithm by Cytron et al. [4]. This includes a dominator tree generation using the basic blocks of the program, the placement of SSA-functions based on dominance frontiers, and the renaming of variables.

The SSA intermediate representation is important when understanding or debugging the optimization and its effect on the final result. Hence, most compilers allow to access such intermediate representations in some way. One of the most common example for a compiler is the GNU Compiler Collection (GCC)[1], which includes an SSA intermediate representation and SCCP optimization. List. 4 shows an extract of the SSA representation of the C program in List. 3. It illustrates the basic block separation (lines 7 and 10), renaming of variables (lines 1, 2, 8, 11 and 12), and placement of a Φ-function (line 11) in the partially translated code.

[1] https://gcc.gnu.org/

```
1 x = 3
2 if (x > 0) {
3     x = x * 7;
4 }
5 y = x * 2;
```

Listing 3: A small C program

```
1 x_3 = 3;
2 if (x_3 > 0)
3 goto <bb 3>; [0.00%]
4 else
5 goto <bb 4>; [0.00%]
6
7 <bb 3> [0.00%]:
8 x_4 = x_3 * 7;
9
10 <bb 4> [0.00%]:
11 # x_1 = PHI <x_3(2), x_4(3)>
12 y_5 = x_1 * 2;
```

Listing 4: Code snippet of the SSA intermediate representation in GCC generated with the `-fdump-tree-ssa` option.

However, the accessibility and understandability, which should directly benefit a user or developer, heavily depends on the compiler itself or on additional affiliated tools that further process these intermediate results. In the use case of SCCharts compilation, the KIELER project also provides an SSA form and SCCP for the SCG. In contrast to regular imperative programming languages, SCCharts has a synchronous semantics with explicit built-in concurrency. This requires a suitable variant of the SSA form. Here, we use an adapted version of the Concurrent SSA (CSSA) form by Lee et al. [8]. It introduces additional SSA-functions for concurrency: Π-functions to handle interleaving due to inter-thread communication and Ψ-functions to merge the variable values when threads join. It is sufficient as intermediate representation for SCCP but does not handle the semantics of SCCharts entirely, hence for more semantics specific optimizations the compiler also contains an SSA form especially tailored for the SCCharts semantics [17, 18].

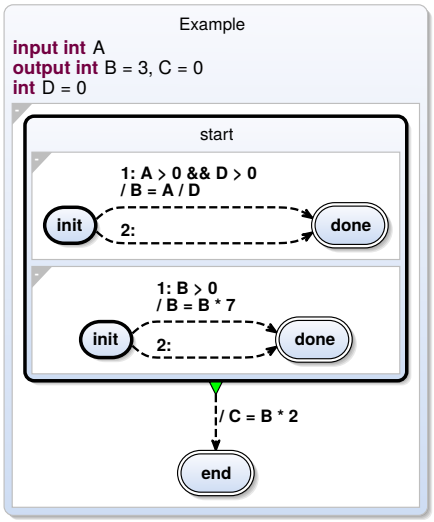
In Fig. 9 and 10 we present some of the intermediate models and additional information available when compiling an SCChart using KiCo and including the SCCP optimization. Initially, this requires a compilation system using optimization processor. Such a system is built by adding the SCCP processor and SSA processors which first produce and afterwards clear the SSA intermediate representation. In our example we create a new system by inserting the configuration shown in List. 5 to the netlist-based compilation system presented in List. 2 at line 9.

```
1 post process de.cau.cs.kieler.scg.processors.ssa.scssa
2 post process de.cau.cs.kieler.scg.processors.ssa.optimizer.sccp
3 post process de.cau.cs.kieler.scg.processors.ssa.unssa
```

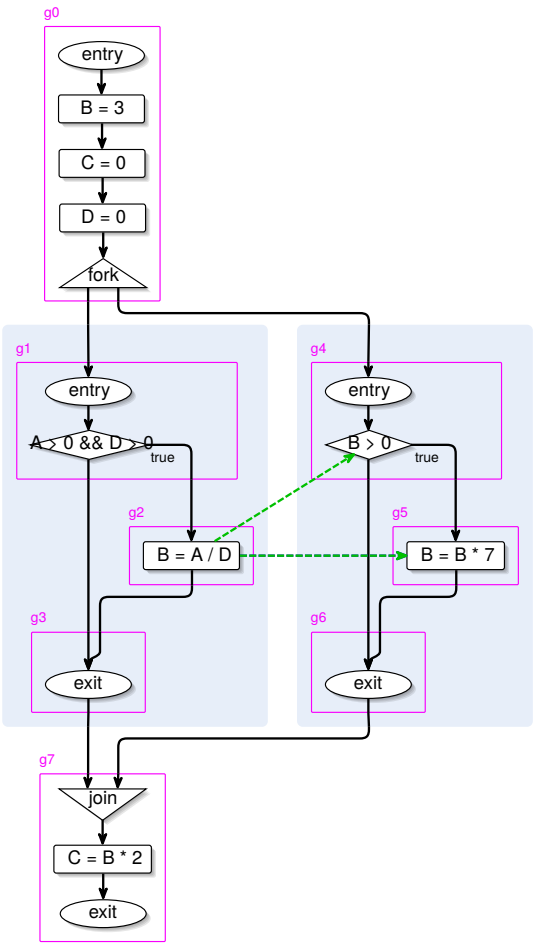
Listing 5: Additional configuration for activating the SCCP optimization.

This positioning behind the basic block processor allows the SSA processor to use the required basic block and dependency information which are now calculated beforehand. The SCCP optimization is performed before the SCG structure is finally analyzed for scheduling.

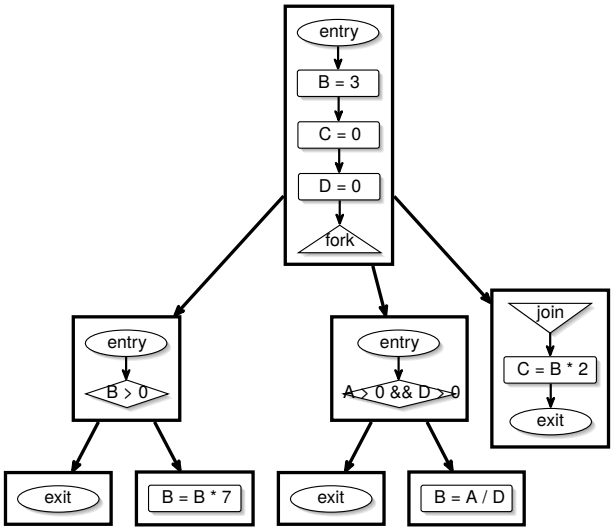
Now this system can be used to compile to our input SCChart in Fig. 9a, which offers quite some optimization potential and is in some parts similar to the previous C code example shown in List. 3. The SCChart has an input A, two outputs B and C, and a local variable D. B is initialized to 3, C and D to zero, and A does not require initialization since it is set by the environment. The execution begins in the initial start state which



(a) The input SCChart

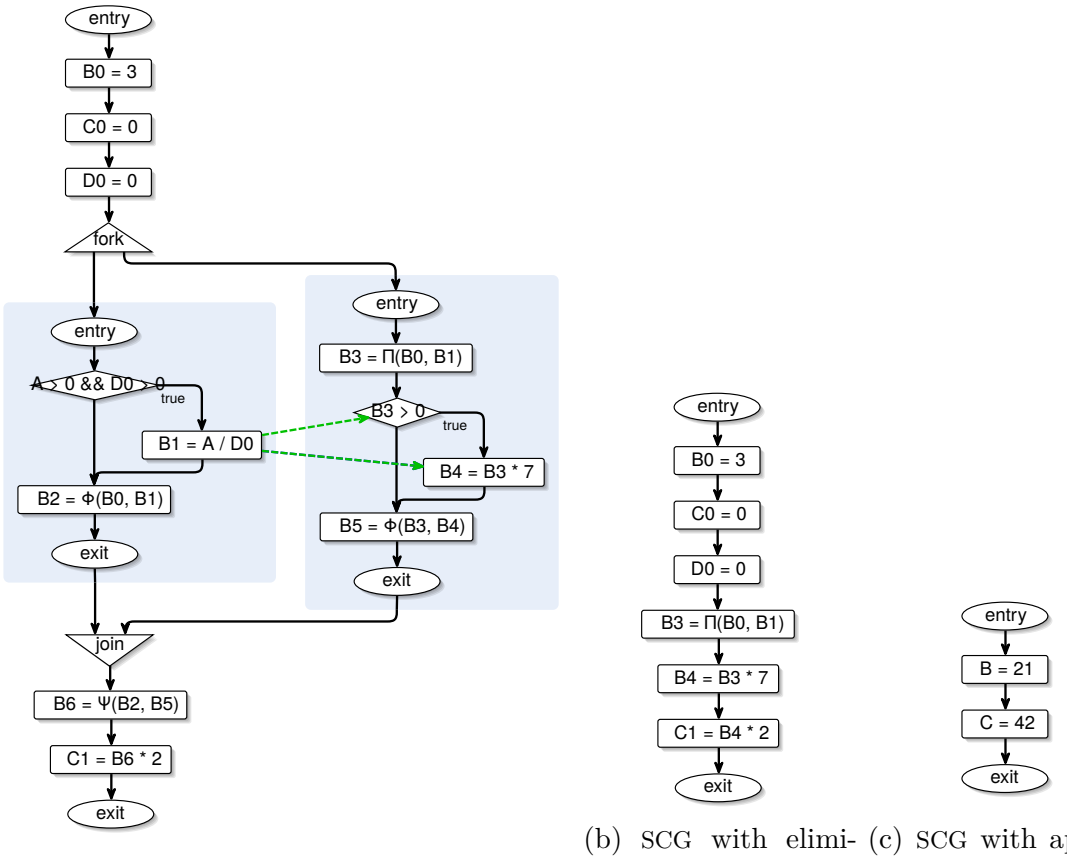


(b) SCG with basic blocks



(c) Dominator Tree

Figure 9: Stepwise SCCP optimization: Input SCChart and SCG structure



(a) SCG in SSA form

(b) SCG with eliminated dead code

(c) SCG with applied SCCP

Figure 10: Stepwise SCCP optimization: SSA form and optimized results

has two parallel regions. The first checks if both A and D are greater zero, if true it sets B to A divided by D otherwise nothing happens, indicated by the transition with priority 2. The lower region checks if B is greater zero and then multiplies B by 7, otherwise it does nothing. Both regions always terminate when their final done states are reached, causing the start state to terminate and transitioning to the end state while assigning C to the double of B. Due to the initialization of D, the first transition of the upper region is never taken, and the initialization B causes the first transition of the lower region to be always active, rendering the second transition dead code. Considering this allows to statically determine the overall behavior of the program, which always outputs 21 for B and 42 for C.

In the compilation the input to the SSA processor is the SCG representation of the SCChart with basic blocks, displayed in Fig. 9b. Note that the additional dashed arrows indicate dependencies between concurrent variable accesses, which are defined by the SCCharts semantics and cause writers to be scheduled before their readers. The next step is the transformation into SSA form, which starts with a dominator analysis based on basic blocks. The internal tree structure resulting from this analysis can be inspected and is shown in Fig. 9c. The tree structure indicates which basic blocks are always

	SCG (9b)	SSA (10a)	DCE (10b)	SCCP (10c)
Nodes	16	20	8	4

Table 1: Results from the automatic evaluation of the processor's intermediate results from Fig. 9 and 10

executed before others and thus dominate them. This structure allows the SSA algorithm to efficiently place the SSA-functions and then rename the variables to create statically single assignments for each variable. The example SCG in CSSA form is presented in Fig. 10a, where Φ-functions handle the variable merge after conditionals, the Π-function incorporates the concurrent writer, and the Ψ-function joins the two writers to B when the threads join. Afterwards, the SCCP optimization can start to propagate constants while eliminating dead code. In this example we want to inspect an intermediate result of this optimization where only the dead code is eliminated but the constant propagation is not yet applied. The result is shown in Fig. 10b. As expected, the optimization detects that both conditionals can statically be evaluated due to the constant values in the conditions. Their dead branches are then removed. The left-hand thread then no longer has any effective content and is also removed, including the fork and join nodes integrating the right-hand thread into the main thread. The Ψ-function and Φ-functions were removed when the control-flow was remodeled after removing the nodes, and the Π-function is an ineffective relict which is removed in the upcoming constant propagation. The final result of the optimization is presented in Fig. 10c. There the constants are propagated, expressions are statically evaluated, assignments to local variables, with no readers or with no effect on the output are removed, and the renaming of the SSA form is reverted. As explained in Chap. 3, all models presented in Fig. 9 and 10 are observable during and after compilation. Inspecting these intermediate models and their visual representations helps users and compiler developers to gather detailed knowledge of the compilation process and its effects on the program. Furthermore, a processor developer can add additional snapshot positions in the compilation to inspect the intermediate results in even more detail as presented here. To debug the presented processor(s) one could inspect for example: each placement of an SSA-function, the renaming of variable names into SSA form, the propagation of each constant, or the elimination of each dead block. As this approach blends interactively into the developer's IDE, processors can also be used to gather experimental results about the models. A compilation system can also invoke analyzers after every processor automatically to gather these information during compilation. The results are stored inside the context. For example, the node results for the SCGs depicted in Fig. 9 and 10 can be seen in Tab. 1. Here, we can observe that the SCG of the source program increases in size at first due to the SSA form. However, afterwards the optimization can reduce its size to 0.25 of the original model size. Of course, it is also possible to present the information graphically in dedicated views in the developer's IDE during run-time.

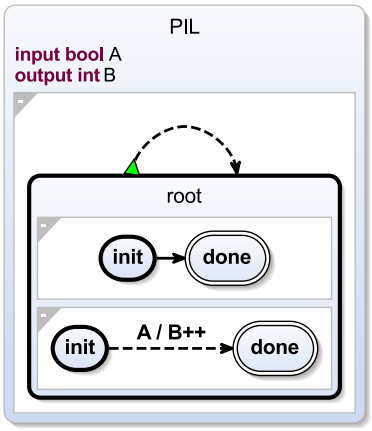
5 Curing Compilation Problems

When it comes to synchronous languages, instantaneous loops are a common issue. The synchronous hypothesis [12] divides times in discrete logical *ticks*. It states that all computation is done in zero time and the reaction of the program is present at the same time as its inputs. Thus, instantaneous loops must be broken to avoid an infinite amount of operations within a tick. Things get even more complicated when control-flow paths become *potentially instantaneous*. This can occur when a (concurrent) program has a feedback loop and the instantaneous path depends on an input.

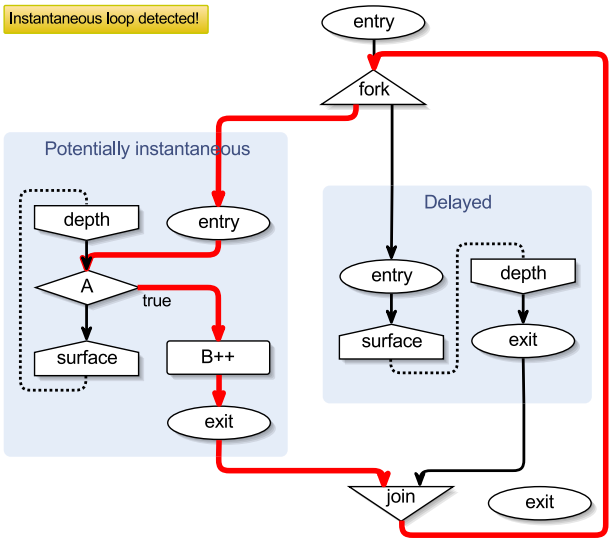
An example can be seen in Fig. 11a. The control-flow graph of the program in Fig. 11b shows the instantaneous path in red. A conservative, netlist-based compiler would reject such a program, because it would recognize the potential instantaneous path. However, semantically the program is fine. The pause in the concurrent thread breaks the cycle, and hence the program will never end up in an instantaneous loop. However, depending on the position of the control-flow at the beginning of the tick, a statement may be executed more than once within a tick. This problem is called *schizophrenia* [21]. In this example, the conditional A and the assignment B++ can be active twice within the same tick, if the control-flow starts in the depth. If A is true, the assignment will be executed and both threads will be exited. The control-flow then loops back to the fork and restarts both threads. A is still true and both, the conditional and the subsequent assignment in the true branch, will be executed again. However, this time, the second thread stops at the pause and the cycle is broken. Eventually, if B was 0 at the start of the tick, then it will be 2 at the end of it.

As soon as one compiles the PIL program with the interactive compilation chain depicted before in List. 2, KiCo will report warnings (yellow) and compilation errors (red). The excerpt in Fig. 12a shows that the compiler stored differed models of the compilation for inspection. The loop analyzer inside of the dependency analysis has detected the instantaneous cycle. The analyzer annotates the path inside the SCG and saves the information in the environment. The result was seen in Fig. 11b. In large models these cycles can be long, and spotting all relevant information on a single screen can become difficult. As described in 3 processors can stores arbitrary additional data in their environments. To help the modeler, the analyzer stores a second model in the environment that strips all other nodes and only shows the instantaneous path. The excerpt model for the example is depicted in Fig. 12b.

Eventually, the scheduler throws an error, because we invoked a netlist-based compilation. In the netlist-based compilation all basic blocks are ordered topologically. However, if we have a cycle, the ordering is impossible. The modeler can inspect the erroneous



(a) Model PIL with potentially instantaneous loop

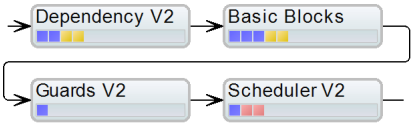


(b) SCG of PIL; the instantaneous path is colored in red

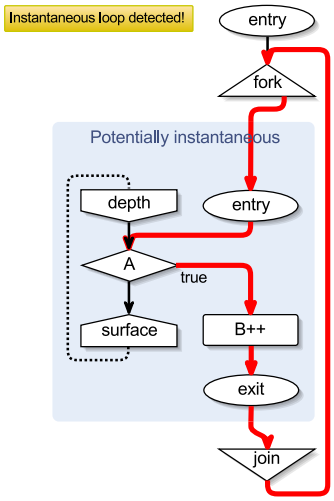
Figure 11: An exmaple for potentially instantaneous loops: PIL

netlist in the first error model the scheduler has stored. An excerpt is shown in Fig. 12c. It shows a representation of the netlist with annotations that show which basic block guards could not have been scheduled. This view might be helpful for developers of the compiler to spot errors, but on the other side it might overload a model with too much too detailed information. However, the scheduler also stored a second model. It used the netlist information to propagate the error back to original model, which is also still present in the environment. In the original model, the processor can annotate the error in a more user-friendly form. Fig. 12d shows the *causal loop* as an annotation inside the original model. The netlist-model mapping can be done heuristically or with more sophisticated *model tracing* [15] if supported by the framework.

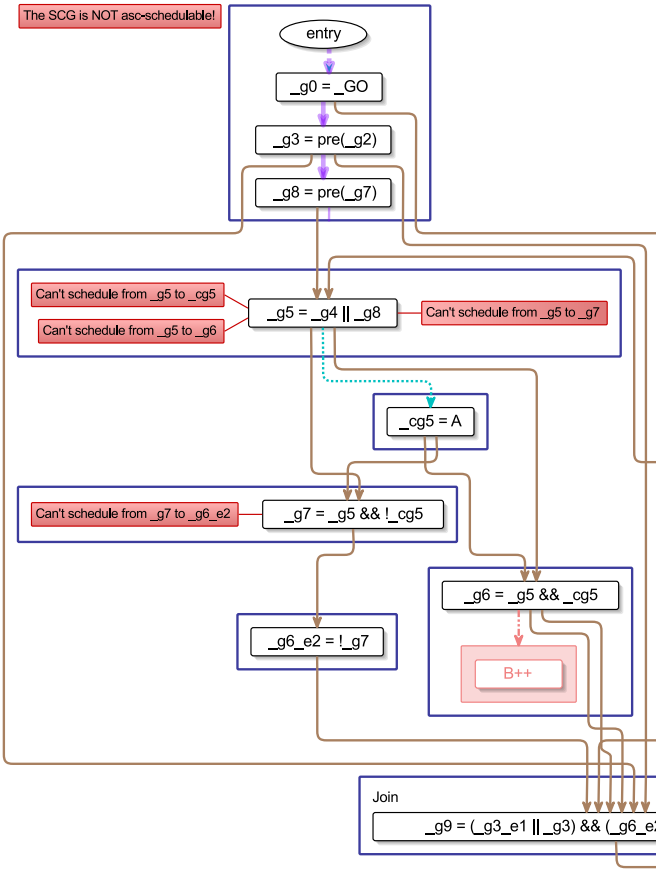
All these views and annotations are available interactively, which should help the modeler to spot potential errors and to fix them in the original model. Furthermore, in some cases, a solution can be deployed to the compiler to accept more programs. In this case, we can also do this interactively, because a processor exists that solves the schizophrenia problem for the netlist-based compilation approach, the *structural depth join* processor. Even if this processor did not exist, one could develop it within the IDE without re-compiling or re-starting the compiler framework. Here, it is sufficient to edit the compiler model (List. 2) and to add the identifier of the structural depth join processor to the basic block transformation (e. g., pre process de.cau.cs.kieler.scg.processors.structuralDepthJoin). The result can be seen in Fig. 12e. The schizophrenic statements got duplicated (displayed in orange) and the potentially instantaneous path is broken. From here, the compilation can proceed as before.



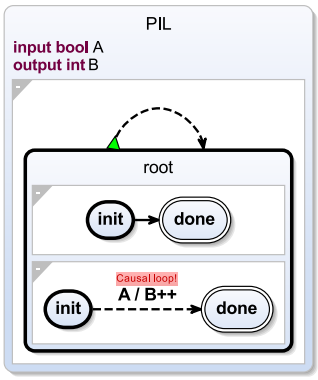
(a) SCG downstream compilation with warnings and errors



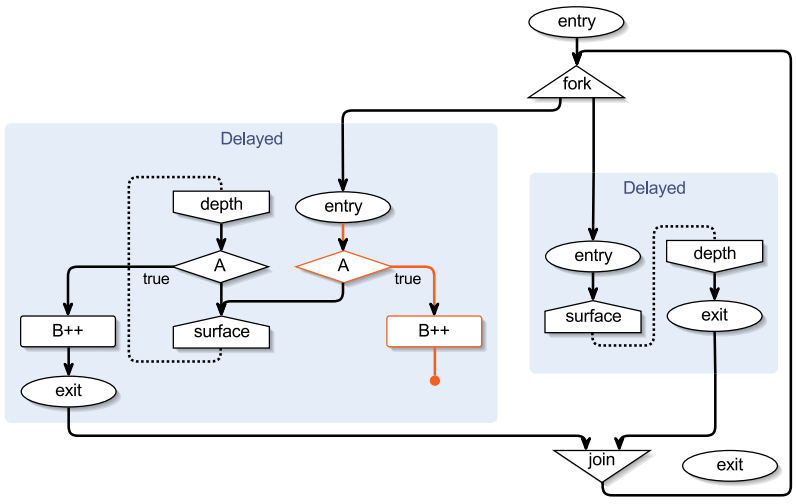
(b) Excerpt of the SCG with focus on the potential problem



(c) Low-level error messages inside the generated netlist for developers



(d) Original model annotated with potential problems



(e) SCG including the solution for the potential loop deployed by the structural depth join processor

Figure 12: Different information layers with annotated models

6 Related Work

Steffen already showed a close relation between compilation and modeling back in 1997 [19]. He proposed to use *consistency models* to detect inconsistencies between different model descriptions. This relates to giving a semantics to a programming language by translation into an intermediate language. Over the years, a number of modeling compilation approaches have been developed such as CINCO [11], a meta-level modeling tool generator, and MARAMA [7], which provides metatools for language specification and tool creation. KiCo's process categorization into specialized work units is in line with ETI's process system [20]. While targeting a slightly different group of experts, such an even more generic process synthesis approach could also be implemented in KiCo. We discuss further possible future routes in the conclusion in Chap. 7. In our approach we provide the modeler with generic, interactive tools to orchestrate compilation processes. These are divided into atomic steps that aid the modeler to refine the process and to find errors without the need for long development cycles. The source, intermediate, target, and additional models are presented in well-readable graphical views using transient view and automatic layout technologies [16].

When it comes to compilation challenges and approaches for concurrent languages, Edwards [5] and Potop-Butucaru al. [13] provide good overviews. Many concepts of the netlist-based compilation of Esterel can be found at the core of the SCCharts compilation. In contrast, we use the presented model-based compilation approach to guide both the compiler developer and the model as shown in Chap. 4 and Chap. 5.

When it comes to general compilation techniques, numerous well-understood approaches (e. g. Copy Propagation [1] and Register Allocation [3]) can be applied to our compiler to improve the results. However, as classical compilers are more or less a *blackbox*, working with intermediate results becomes difficult. For example, as depicted in Chap. 4, the gcc[1] possesses settings to toggle different optimizations or to print out intermediate representation of the basic blocks [2] of a source program. The interplay between the different modules and the textually representation of data seems to only target compiler experts and is arguably rarely useful for the common user. Also, when working on modern paradigms (e. g. deterministic concurrent scheduling in synchronous languages), even standard techniques sometimes become difficult to apply and reveal interesting research questions. Our pragmatic model-based approach provides us with flexible tools to pursue these.

[1] https://gcc.gnu.org

7 Conclusions

We presented how the compiler framework works that was used to create the reference implementation for the synchronous language SCCharts. We showed that process systems, such as compilation or simulation, themselves self are also models and how this can help both, the domain expert and the tool developer, with the goal to get better results faster. While both may have different foci during a project's lifetime, both can use a similar framework to drive their development and to help each other.

For us, in Model-driven Software Development (MDSD), programming with models does not only mean to model a program with, e. g., a sophisticated IDE that provides us with new tools to construct the program. For example, modern programming IDEs provide features such as syntax highlighting, code completion, reference counting, refactoring, etc. Many of these features focus on creating the model and then they are done. In our approach, the whole process is modeled. The user can inspect and change every part of it interactively. They can influence the compilation improving the final result or add new processors that provide new models and views to give better feedback. We thus argue that MDSD is not solely about modeling a particular artifact. It is also about the way to get to the final model.

Besides further improvements for the SCCharts compiler and streamlining the MDSD user experience, we see further future work. For example, the KIELER project includes several modules that still use dedicated components that perform dedicated model transformations to prepare the models for specific tasks. As illustrated in Chap. 2, we are currently working on the compilation and simulation systems. However, the KiCo framework could also be used to generalize even more of these processes, e. g., deployment tasks. This would also facilitate the re-usability of the approach beyond the classical compilation task. Furthermore, we want to combine our approach with the continuing trends of mobile location-independent technologies, such as mixed web/desktop applications using tools such as electron[1]. We are optimistic that this will further increase the possibilities for and flexibility of prototyping and team-driven software development.

[1] https://electronjs.org

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[2] F. E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[3] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Comput. Lang.*, 6(1):47–57, Jan. 1981.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[5] S. A. Edwards. Tutorial: Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems*, 8(2):141–187, Apr. 2003.

[6] H. Fuhrmann and R. von Hanxleden. On the pragmatics of model-based design. In *Foundations of Computer Software. Future Trends and Techniques for Development—15th Monterey Workshop 2008, Budapest, Hungary, September 24–26, 2008, Revised Selected Papers*, volume 6028 of *LNCS*, pages 116–140, 2010.

[7] J. C. Grundy, J. Hosking, K. N. Li, N. M. Ali, J. Huh, and R. L. Li. Generating domain-specific visual language tools from abstract visual specifications. *IEEE Transactions on Software Engineering*, 39(4):487–515, Apr. 2013.

[8] J. Lee, S. P. Midkiff, and D. A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '97, pages 114–130. Springer-Verlag, 1998.

[9] C. Motika. *SCCharts—Language and Interactive Incremental Implementation*. Number 2017/2 in Kiel Computer Science Series. Department of Computer Science, 2017. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel.

[10] C. Motika, S. Smyth, and R. 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, Oct. 2014.

[11] S. Naujokat, M. Lybecait, D. Kopetzki, and B. Steffen. Cinco: a simplicity-driven approach to full generation of domain-specific graphical modeling tools. *International Journal on Software Tools for Technology Transfer*, 20(3):327–354, Jun 2018.

[12] D. Potop-Butucaru, R. de Simone, and J.-P. Talpin. The synchronous hypothesis and synchronous languages. In R. Zurawski, editor, *Embedded Systems Handbook*. CRC Press, 2005.

[13] D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*. Springer, May 2007.

[14] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM.

[15] F. Rybicki, S. Smyth, C. Motika, A. Schulz-Rosengarten, and R. 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)*, volume 8802 of *LNCS*, pages 443–462, Corfu, Greece, Oct. 2016.

[16] C. Schneider, M. Spönemann, and R. 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, Sept. 2013.

[17] A. Schulz-Rosengarten. Strict sequential constructiveness. Master thesis, Kiel University, Department of Computer Science, Sept. 2016. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/als-mt.pdf.

[18] A. Schulz-Rosengarten, S. Smyth, R. von Hanxleden, and M. Mendler. A sequentially constructive circuit semantics for Esterel. Technical Report 1801, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Feb. 2018. ISSN 2192-6247.

[19] B. Steffen. Unifying models. In *STACS 97, 14th Annual Symposium on Theoretical Aspects of Computer Science, Lübeck, Germany*, pages 1–20, Mar. 1997.

[20] B. Steffen, T. Margaria, and V. Braun. The Electronic Tool Integration platform: concepts and design. *International Journal on Software Tools for Technology Transfer*, 1(1):9–30, Dec 1997.

[21] O. Tardieu and R. de Simone. Curing schizophrenia by program rewriting in Esterel. In *Proceedings of the Second ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'04)*, San Diego, CA, USA, 2004.

[22] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. 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. ACM.

[23] R. von Hanxleden, E. A. Lee, C. Motika, and H. 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, Dec. 2012.

[24] M. N. Wegman and F. K. Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.