INSTITUT FÜR INFORMATIK

Practical Causality Handling for Synchronous Languages

Steven Smyth and Alexander Schulz-Rosengarten
and Reinhard von Hanxleden

Bericht Nr. 1808
December 2018
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

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

Practical Causality Handling for Synchronous Languages

Steven Smyth and Alexander Schulz-Rosengarten and
Reinhard von Hanxleden

Report No. 1808
December 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
Design, Automation and Test in Europe (DATE'19)
Florence, Italy, March 2019.

***Abstract*—The synchronous principle is a well-established paradigm for reconciling concurrency with determinism. A key is to establish at compile time that a program or model is *causal*, which basically means that there exists a schedule that obeys the rules put down by the language. This rules out surprises at run time; however, in practice it can be rather cumbersome for the developer to cure causality problems, in particular as programs/models get more complex.**

We here propose to tackle this issue in two ways. Firstly, we propose to enrich the scheduling regime allowed by the language to not only consider data dependencies, but also explicit scheduling directives that operate on statements or coarser scheduling units. These directives may be used by the developer, or also by model-to-model transformations within the compiler. Secondly, we propose to enhance programming/modeling environments to guide the developer in finding causality issues. Specifically, we propose dedicated *causality views* that highlight data dependencies involved in scheduling conflicts, and *structure-based editing* to efficiently add scheduling directives.

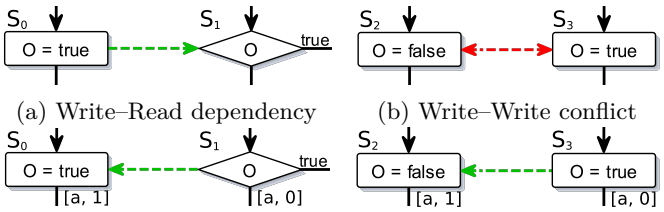
We illustrate our proposals for the SCCharts language. An Eclipse-based implementation based on the KIELER framework is available as open source.

***Index Terms*—model-based design, scheduling, synchronous languages, modeling pragmatics, SCCharts**

I. Introduction

To reconcile concurrency and determinism for programming reactive systems, synchronous languages follow strictly defined models of computation (MoCs). Execution is separated into discrete *ticks*, where (sensor) inputs are read and (actuator) outputs are produced. Within each tick, concurrent threads of execution progress according to certain scheduling rules, defined by the MoC, that guarantee determinism. For example, the prominent *write-before-read* principle, employed in languages such as Esterel [4] and in dataflow languages, such as Lustre [7], demands that a write to some variable x must be scheduled before a concurrent read of x.

The write-before-read principle clearly guarantees determinism, but like other scheduling rules comes at the price that a compiler may reject a program because it cannot find a viable schedule for it, e. g., because of cyclic write-read dependencies. We then say that the program is *not causal*, and it is the programmers job to fix the program. This, in practice, is often easier said than done, due to different reasons. 1) Some synchronous MoCs are restrictive in ways that the average programmer may not expect; 2) the compiler's analysis and scheduling abilities may be limited and conservatively reject programs that would indeed be schedulable; and 3), the feedback provided by the compiler may be too limited to be helpful to the programmer, in particular when the program gets complex. Issues 1) and 2) not only matter for the human developer, but also when transforming a program or model as part of a compilation; restrictive scheduling regimes defined by the MoC may make model-to-model transformations that compile advanced language features into simpler ones more complex than one might hope.



(a) Write–Read dependency (b) Write–Write conflict

(c) Read–Write SD dependency (d) Write–Write SD dep.

Figure 1: Dependencies induced by either a MoC or an SD

Contributions & Outline: To make causality handling more practical we present two proposals. First, we propose to add Scheduling Directives (SDs) that form Flexible Schedules (FSs) to synchronous languages (Sec. II). These should not replace existing scheduling regimes, but rather augment them, either to change the default scheduling or to make program schedulable (causal) in the first place. This approach should not replace existing solutions for solving causality issues, such as *pre*, but adds another tool to the repertoire of the modeler. We also illustrate how model-to-model (M2M) transformations can benefit, without the modeler having to interact (Sec. III) using the synchronous language SCCharts as a demonstrator. Second, we present three different ways to guide the user to causality problems using transient view technologies, namely *data dependency views*, the *causality dataflow view*, and *annotated compilation models* (Sec. IV). We discuss related work in Sec. V and conclude in Sec. VI.

II. Scheduling Directives and Flexible Schedules

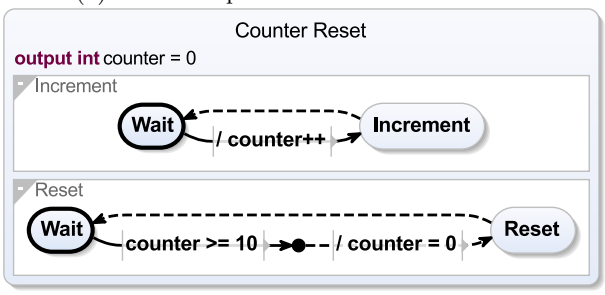
Accesses to variables are usually categorized into *writers* and *readers*. A possible control flow graph representation, as depicted in Fig. 1, shows assignment statements (rectangle nodes) and conditional statements (diamond nodes). A *schedule* is a static order of all nodes in a control flow graph, meaning the order is determined at compile-time and fixed during run-time. The particular ordering is governed by the used MoC. Usually, it is determined by the *control* and/or (concurrent) *data dependencies*. Fig. 1 shows four examples of concurrent data dependencies. In Fig. 1a a write-before-read dependency is depicted as green dashed arrow. The control flow is also visible as black solid edges. An exemplary relation for these statements is $s_0 \rightarrow_{moc} s_1$, with $\rightarrow_{moc}$ being an order relation that implements the rules of the underlying MoC (s_0 before s_1). Fig. 1b shows two conflicting write accesses. In the example, the dependency conflict is depicted as red dashed double arrow.

A *scheduling directive* (SD) associates a *scheduling unit* with a *named schedule* and an *index*. The scheduling unit may be for example a single statement, or a coarser unit of execution such as a thread. For a named schedule s, the scheduling units associated with s must be scheduled

```
1  scchart CounterReset {
2    output int counter = 0
3
4    region Increment:
5    initial state Wait
6      do counter++
7        go to Increment
8
9    state Increment
10     immediate go to Wait
```

```
10   region Reset:
11   initial state Wait
12     if counter >= 10 go to Do
13
14   connector state Do
15     immediate do counter = 0
16       go to Reset
17
18   state Reset
19     immediate go to Wait
20 }
```

(a) Textual representation of Counter Reset



(b) Automatically generated graphical representation of Counter Reset

Figure 2: Concurrent Counter Reset program in SCCharts

according to their index, lowest index first[1]. For example, considering Fig. 1c, we may add an SD to each of the scheduling units (statements) s_0 and s_1 that associates them with schedule a and indices 1 and 0, respectively. This induces a scheduling order $s_1 \rightarrow_{sd} s_0$. The value of O is now read from in s_1, before written to in s_0. Analogously, the write–write conflict is resolved in Fig. 1d by giving statement s_3 a lower index than statement s_2.

A *flexible schedule* (FS) is a schedule that takes all SDs of the model into account. If there exists an SD for two statements, the SD order ($\rightarrow_{sd}$) is used. Otherwise, the MoC determines the order ($\rightarrow_{moc}$).

For a model that contains scheduling conflicts, we propose to not consider it causally wrong per se, but merely incomplete. When a conflict occurs that leads to an incomplete model, the modeler can complete it with SDs. They can be used *directly* on different levels of detail as will be shown in this section, and *indirectly* via M2M transformations as will be explained in Sec. III.

A. Causality in SCCharts

We exemplify modeling with SDs in the SCCharts [19] language. Fig. 2b shows an diagram of an SCCharts model, named Counter Reset. The textual source program is shown in Fig. 2a. In the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) modeling environment, an SCChart diagram is automatically generated from the textual source, as depicted in Fig. 2b. The model has one integer *output* counter, which represents a counter

[1]We here avoid the term "priority" to avoid confusion with the priorities of priority-based scheduling [19], where the highest priority is executed first.

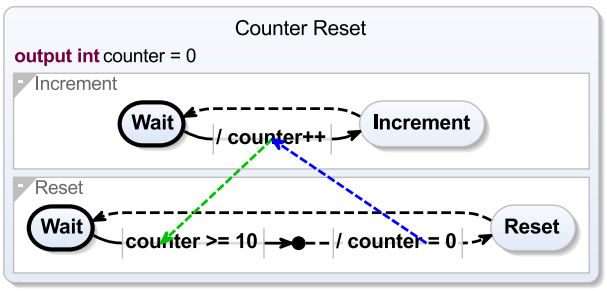


Figure 3: Data dependencies are visualized as colored, dashed edges between the regions.

value, and two concurrent *regions*, Increment and Reset. In the region Increment, there are two *states*, Wait and Increment, which are connected via *transitions*. The *initial* state is depicted with a bold border. A solid transition is *delayed*, meaning it will at the earliest trigger one tick after the originating state was entered, whereas a dashed transition is *immediate*, which means that it can trigger as soon as the state is entered. Hence, in every tick, counter gets incremented in Increment, which in the SCCharts MoC is considered an *update*. In the Reset region, the state Wait waits for the counter to reach the value 10. Afterwards, it should be reset to 0. However, this results in a conflict, because the scheduling protocol, in this case SCCharts' *initialize-update-read* protocol, states that concurrent accesses within one tick can only set, update, and read variables in this particular order as indicated by the colored, dashed dependencies in Fig. 3. Thus we have a scheduling cycle counter = 0 $\rightarrow_{moc}$ counter++ $\rightarrow_{moc}$ counter >= 10 $\rightarrow_{moc}$ counter = 0. Therefore, under the SCCharts MoC, similar to other synchronous MoCs, this model would be considered not causal and would not compile.

B. Scheduling Directives on Statement-Level

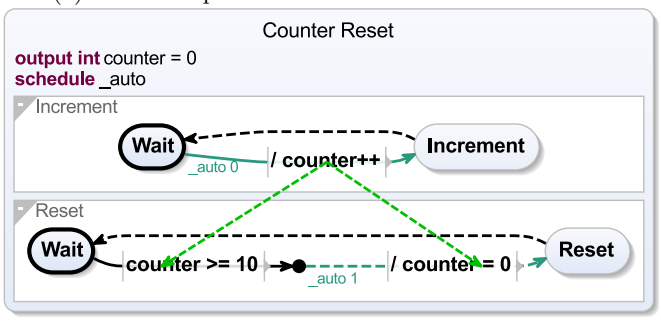
We extended SCCharts with the possibility to add SDs to a model using named schedules. To illustrate, consider Fig. 4a, which is the Counter Reset example from Fig. 2 enriched with SDs. First, a named schedule _auto is declared, in line 3. Named schedules can be used in SDs, which are of the form ⟨*scheduling unit*⟩ schedule ⟨*schedule name*⟩ ⟨*index*⟩. In Fig. 4a, the SDs in lines 8 and 19 resolve the cycle (recall Sec. II-A) by incrementing the counter before the test and reset.

It may be difficult for a modeler to obtain an overview over all conflicts and subsequent potential cures for these conflicts. Thus, instead of letting the modeler add SDs, a perhaps more efficient approach is to define SDs by interacting with the diagram, which will be discussed in Sec. IV-A.

Scheduling Transitivity: Since it is possible to declare arbitrary many schedules, it is also possible to declare a schedule for any number of concurrent expressions.

```
1  scchart CounterReset {
2    output int counter = 0
3    schedule _auto
4
5    region Increment:
6    initial state Wait
7      do counter++
8        schedule _auto 0
9        go to Increment
10
11   state Increment
12     immediate go to Wait
13   region Reset:
14   initial state Wait
15     if counter >= 10 go to Do
16
17   connector state Do
18     immediate do counter = 0
19       schedule _auto 1
20       go to Reset
21
22   state Reset
23     immediate go to Wait
24 }
```

(a) Textual representation of Counter Reset with SDs



(b) Automatically generated graphical representation of Counter Reset; the dependency edges are now influenced by the SDs.

Figure 4: Counter Reset example with SDs

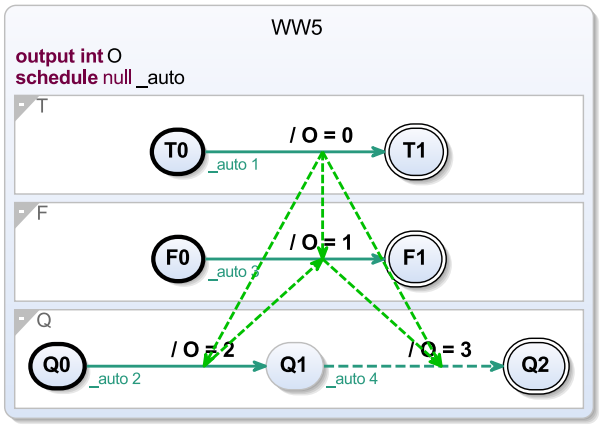


Figure 5: Example WW5 depicts a strict order across three regions.

However, when considering user-defined schedules, it is sufficient to use exactly one schedule per superstate due to the transitive nature of the scheduling.

Fig. 5 shows a more complex model with the three concurrent regions T, F, and Q. Let _autoTF, _autoTQ, and _autoFQ be different, named schedules between the regions T, F, and Q. The scheduling constraints are then given by

$$\begin{aligned} \text{\_autoTQ} &: T0 \rightarrow Q0 \rightarrow Q1 \\ \text{\_autoTF} &: T0 \rightarrow F0 \\ \text{\_autoFQ} &: Q0 \rightarrow F0 \rightarrow Q1. \end{aligned} \tag{1}$$

However, following the transitive principle, the modeler can enforce the schedule $T0 \rightarrow Q0 \rightarrow F0 \rightarrow Q1$ with one scheduling definition, which is named _auto in the example.

When looking at individual dependencies, it is still possible to create a *dependency cycle*, which is not schedulable, e. g., $T0 \rightarrow Q0 \rightarrow F0 \rightarrow T0$. However, as concurrent conflicts are solved by prioritizing conflicting expressions within a single schedule, the aforementioned scenario cannot be expressed in the SCCharts language. It is still possible to create an unschedulable program if the control flow disallows a dependency schedule. For example, the schedule $Q1 \rightarrow T0 \rightarrow F0 \rightarrow Q0$ is expressible, but not schedulable, because the control flow from Q0 to Q1 makes the schedule infeasible.

C. Scheduling Directives on Coarser Granularities

It is often sufficient to define SDs on a coarser granularity than the statement level. Especially when the language supports a distinction between core (resp. kernel) and extended features, coarser granularities are implemented easily. If statement-level SDs are available in the core language, coarse granularity SDs can be implemented as extended features, which can be transformed automatically to statement-level SDs via M2M transformations.

Directives on transitions and actions: Actions come in various forms in statecharts dialects. An example is the *entry action* of a superstate, which is executed as soon as the superstate is entered. Using the schedule keyword on a transition or an action assigns the SD to all statements of the transition/action. Note that this precludes interleaved execution of the transitions.

Directives on region: Using schedule on regions sets the directive for all statements in that region.

III. Scheduling Directives in Transformations

Consecutively executed M2M transformations are the core of a model-based compiler [10]. Even if the modeler does not use SDs directly, they can improve these transformations w.r.t. complexity and efficiency.

Count Delay: One M2M transformation in the SCCharts compiler transforms the *count delay* feature into simpler constructs. In a graphical syntax, count delay is depicted as an integer n in front of a transition trigger. Such a transition is only taken if it would have been eligible to run n times without the count delay. An example of two alternating count delays can be seen in Fig. 6.

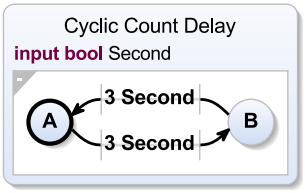


Figure 6: Cyclic count delay

A straightforward transformation which simply counts the occurrences as implemented by Motika [9] adds a counter per count delay and waits until n is reached. This works for simple count delays. However, if two count delays are called in a cyclic manner as in Fig. 6, this simple approach fails, because of cyclic dependencies that are

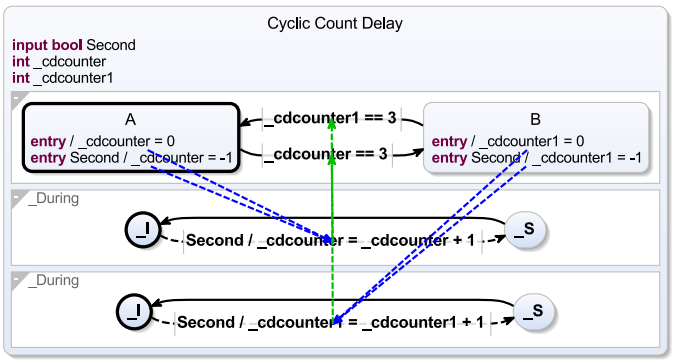


Figure 7: Expanded cyclic count delay

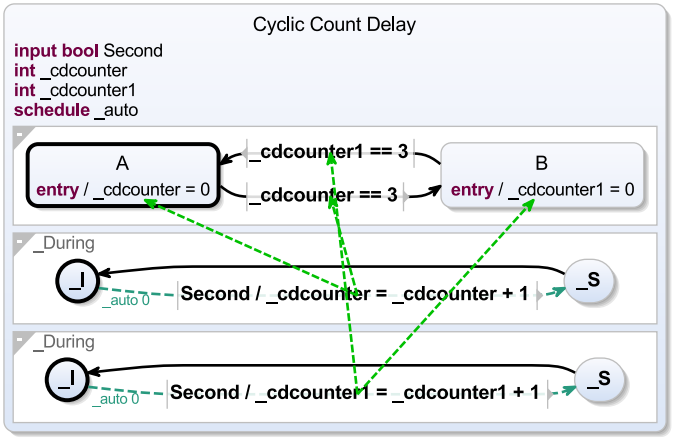


Figure 8: Cured expanded Cyclic Count delay

introduced by the M2M transformation, see Fig. 7, similar to the pattern shown in Sec. II-A.

The current version of the SCCharts compiler solves this problem by using a more sophisticated transformation that uses *pre* operators to look at values of from the previous ticks, which is a common way for solving causality problems in synchronous languages. However, since the increments should always be performed before the test and reset, this transformation can be done more efficiently with SDs similar to the counter example presented in Sec. II-B. It is sufficient to set the scheduling index of the counting regions to a lower value than the index of the main region. As a result, the SDs make sure that the increments are happening before the checks and potential resets of the counters, see Fig. 8. Additionally, an arguably unintuitive reset to -1 in Fig. 7, which was necessary previously to handle the case of a reset and a subsequent increment in the same tick, can be omitted.

Tab. I compares the three different implementations of SCCharts' count delay transformation when compiling the Cyclic Count Delay model in Fig. 6. While the simple approach is not able to handle two cyclic count delays, the pre variant needs more variables, states, and regions than the SDs approach. Furthermore, the pre transformation also creates schizophrenic models, that is, models where

	Simple	with Pre	with SDs
Schedulable	No	Yes	Yes
Schizophrenia	–	Yes	No
Variables	2	8	2
States	22	44	18
Regions	5	10	5
Binary Size (b)	–	2702	1337

TABLE I: Results of the different count delay approaches in SCCharts when compiling the Cyclic Count Delay model in Fig. 6

statements are executed more than once within one tick. Handling schizophrenia does not come trivially [18]. The SD solution avoids schizophrenia.

Timed Automata: A similar use-case is present in the way SCCharts supports timed automata [16]. Currently, such models are quite restrictive when it comes to parallel and hierarchical compositions, because timed automata allow a reset of a clock, for example, when a specific amount of time passed. Fig. 10 illustrates an SCChart representing a trafficlight controller. In each state a during action increments the local clock `x` by the passed time in `deltaT`. Without SDs it is not possible to write a single during action on root state level, since the increment must be scheduled before the test and reset of the clock in an outgoing transitions. This is the same case as the count delay example. Here, the compilation can benefit form SDs and simplify the program and reduce duplicate code.

Enforcer: An example for synchronous enforcers is the pacemaker enforcer [11]. It ensures that the signals from a pacemaker to the heart stay in defined parameter ranges and corrects them if necessary. Originally, the scheduling of the regions inside the enforcer was hard-coded to allow compact models with fixed-scheduled regions. With FSs, there is no need for special handling of this case as region-level SDs are sufficient. Setting the regions input to index 0, tick to 1, and output to 2 generates an enforcer with the desired properties.

IV. Guidance to Causality Conflicts

The modeler should not be burdened with maintaining an overview over all potential conflicts, but should be assisted with finding solutions to these. The KIELER SCCharts tools provide automatically generated graphical *transient views* while the modeler works on their model [14]. To guide the modeler to potential conflicts, we extend the standard transient diagram of the model with additional views, namely *data dependency visualization*, *causality dataflow view*, and *annotated compilation models*, which are described in the following.

A. Data Dependency Visualization

The data dependency visualization is used to identify individual conflicting data dependencies. This view is used to display the dependencies in the counter reset example in

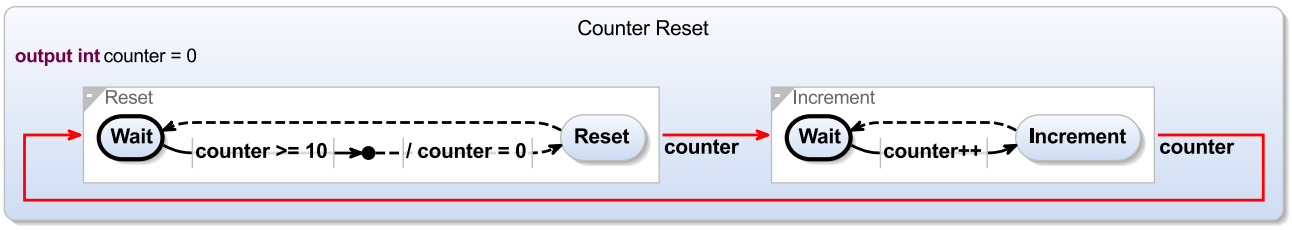


Figure 9: Counter Reset model shown with causality dataflow

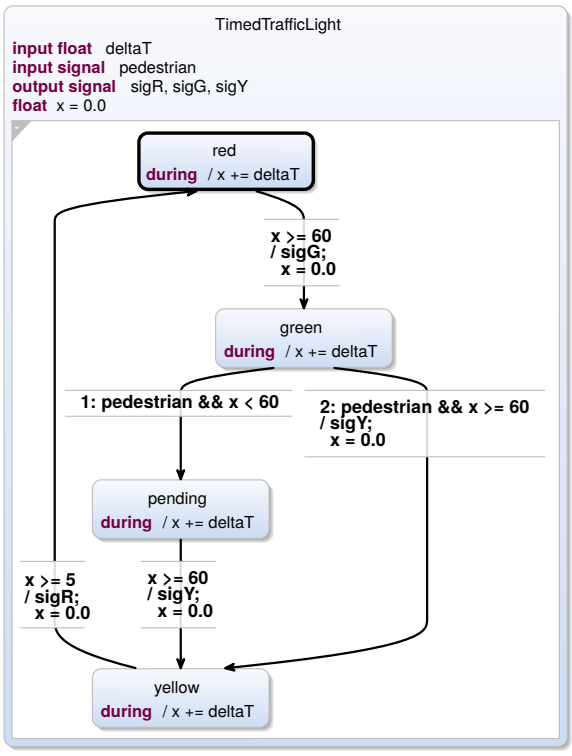


Figure 10: Trafficlight controller modeled as timed automaton in SCCharts [16].

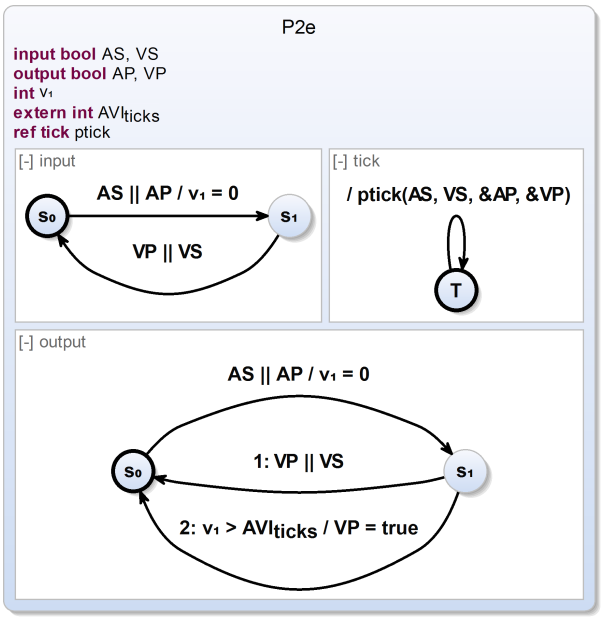


Figure 11: Example of an pacemaker enforcer [12]

Fig. 3 and others. The view augments the diagram with data dependencies that originate from variables accesses in the model. Furthermore, the modeler directly interacts with the diagram to add SDs in a user-friendly way.

For example, in the counter reset model (Fig. 3), the dependency from counter = 0 to counter++ can be reversed with an appropriate SD. When the user clicks on the dependency edge, the model diagram (Fig. 4b) is modified. A new schedule, named _auto, is declared as shown in Fig. 4a. Two directives assign this schedule and the indices 0 and 1 to the appropriate statements in the underlying model to reverse the dependency direction. The textual and graphical views adapt to the new model: Lines 3, 8, and 19 are added automatically.

B. Causality Dataflow View

The *induced dataflow view* [21] shows communication between regions. We propose a variant thereof, the *causality dataflow view*, which focusses on identifying data dependency cycles. Fig. 9 shows the same model as Fig. 2b in a *causality dataflow view*, which shows a dependency cycle in red.

Here, a natural way to solve cyclic behavior is to delay the dataflow by adding a register (resp. a pre function to access the value of the previous tick) to the loop, which is an often used technique in synchronous languages and was also used in the count delay transformation, as explained in Sec. III. The view shows the general dataflow even in state-based languages and hence is similar to the data dependency visualization view, but differs in granularity and arrangement of elements in the diagram.

C. Annotated Compilation Models

The usual method of error reporting in tools are messages, for example, as console output. These messages usually contain additional information about the location of the error to guide the user towards it. In graphical languages, such a message can be embedded directly into the (original) model by annotating it. The KIELER framework allows to create annotated models during compilation to hint at potential problems.

When compiling the Counter Reset example (Fig. 2), the compilation will fail due to the causality cycle. However, the cycle is detected on a low level of the compilation.

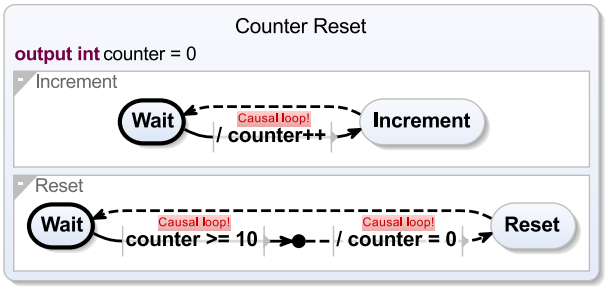


Figure 12: Annotated model of Counter Reset created by the compiler

Since it is rarely helpful for the modeler to refer to this low-level error message, the KIELER framework propagates the issue information back to the original model. The result is shown in Fig. 12. Besides scheduling conflicts, any data that is gathered during the compilation process can be reported this way if corresponding elements can be found in the original model.

V. Related Work

Many of the established MoCs of synchronous languages enforce a strict write-before-read semantics. For control flow-orientated languages, Esterel [4] (resp. SyncCharts [3]) and for dataflow-orientated languages, Lustre [7] and SCADE [6] are common examples. In this paradigm, even if not in a concurrent context, it is forbidden to change a value after it has been read from. Also, all writers must conclude to a deterministic value, hence the order of multiple writes must not influence the resulting value. This is managed by *confluent* writes, e. g., by writing the same value or using combinatorial functions. While the semantics are sound and concise, they make things difficult for imperative programmers, because common pattern such as `if (!x) { ...; x = true; }` cannot be expressed in these languages. Also, the semantics of signals, which mimic wires in a circuit, may not come as intuitive for traditional programmers, who are used to variables, which represent memory.

To handle causality in languages with write-before-read semantics might be more of an issue in control flow orientated languages, because of the sequential thinking of the programmer. In dataflow languages, actors are often inherently ordered sequentially due to the flow of data. However, even in dataflow languages (e. g. when programming textually in Lustre), causality between equations is not trivial. Especially if a sequential ordering of variables (resp. streams) is indicated textually, but interpreted as parallel execution, which is unintuitive for an imperative programmer.

SCCharts [19] uses the sequentially constructive (SC) MoC, which adheres to the sequential order in the program and only for concurrent variables accesses applies the *initialize-update-read protocol* (a refined version of write-before-read). This allows to accept more common imperative programming pattern, e. g., `if (!x) { ...; x = true; }`. Nonetheless, it still prescribes a fixed scheduling regime for concurrent communication. The protocol states that in a concurrent context initializations (writes) come before updates (combinatorial writes) which come before reads. Basically, this maps to three scheduling priorities, where the variable accesses inside these priority classes must be confluent towards each other. Nevertheless, depending on the chosen SCCharts semantics [20], to determine whether or not a program is *SC* is not trivial; particularly in the concurrent context. The most recent, non-speculative concise circuit semantics for SCCharts [15] also comes with strong restrictions for concurrency and is not considered practical in the sense of this contribution.

A generalization of dependency-based scheduling regimes are *policy interfaces*, proposed by Aguado et al. [1]. These also provide very flexible scheduling regimes, but are based on types, rather than scheduling units.

Another form of synchronous concurrency forbids direct communication within the same tick, as deployed in languages such as ForeC [22]. These languages can only access concurrent data from previous ticks, which cannot be modified any more. In some sense, this also resembles a write-before-read semantics.

Simulink/Stateflow [8] define the scheduling order depending on the graphical ordering of elements. This concept is not robust towards changes of the graphical representation, as slight modifications of the diagram can change the scheduling order. In PRET-C [2] the textual order defines the scheduling. This reflects a semantics where all scheduling decision are made explicit, even if this is not necessary.

Statecharts without strict distinct notion of time often employ a run-to-completion semantics, such as the UML statecharts [13] or the Yakindu Statechart Tools[2]. These semantics cannot be seen as direct communication in the sense of synchronous languages, where there exists some kind of back-and-forth communication within a tick.

If a program is rejected by the compiler, it is important to guide the user towards the problem. Textual languages often support common error messages or even textual highlighting, such as the X Esterel Simulator (XES) shown in Fig. 13, which can be seen as textual form of the annotated compilation models shown in Sec. IV-C. Graphical languages have the advantage of intuitive visual problem reporting. However, regarding synchronous languages, such as SyncCharts and SCADE, this potential is often only used for simulation, e. g., in the SCADE Suite[3]. To our knowledge there are no specific views or dedicated model augmentation for detecting and solving scheduling problems, such as we present them in this paper.

VI. Conclusion

We showed how to add Scheduling Directives (SDs), which form Flexible Schedules (FSs), to synchronous lan-

[2]https://www.itemis.com/en/yakindu/state-machine
[3]http://www.esterel-technologies.com/products/scade-suite/verification-validation/scade-suite-simulator/

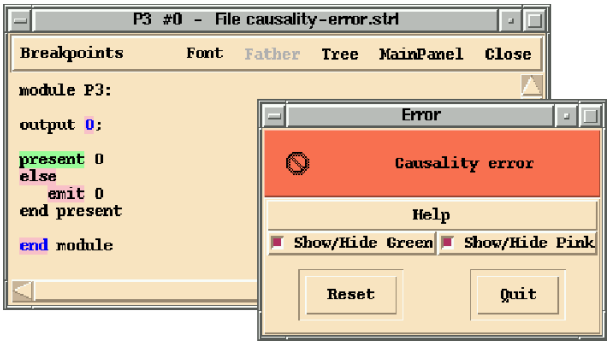


Figure 13: Causality error reporting in the X Esterel Simulator (excerpt [5])

guages. A modeler can use these SDs to explicitly alter the scheduling of the underlying MoC on modeling level to solve causality issues. It also enables M2M transformation developers to write simpler and more efficient transformations, as demonstrated in Sec. III.

To guide the user to potential conflicts, we proposed different views to spot causality issues. We argue that the data needed for these views often already exist in most compilation approaches, but must be presented to the modeler in a useful way. These enriched views make the aforementioned SD approach practical.

For future work, we want to investigate which M2M transformations can also profit from SDs and which forms of SDs benefit the modeler most. Using SDs on state-level, for example, would provide the modeler with a SD granularity between regions and transitions.

References

[1] J. Aguado, M. Mendler, M. Pouzet, P. S. Roop, and R. von Hanxleden. Deterministic concurrency: A clock-synchronised shared memory approach. In *27th European Symposium on Programming, ESOP'18*, pages 86–113, Thessaloniki, Greece, Apr. 2018.

[2] S. Andalam, P. S. Roop, and A. Girault. Deterministic, predictable and light-weight multithreading using PRET-C. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'10)*, pages 1653–1656, Dresden, Germany, 2010.

[3] C. André. Semantics of SyncCharts. Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.

[4] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 425–454, Cambridge, MA, USA, 2000. MIT Press.

[5] G. Berry and the Esterel Team. *The Esterel v5_91 System Manual*. INRIA, June 2000. http://www-sop.inria.fr/esterel.org/.

[6] Esterel Technologies. *SCADE Technical Manual*, 5.1 edition, Feb. 2006.

[7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[8] G. Hamon. A denotational semantics for Stateflow. In *EMSOFT'05: Proceedings of the 5th ACM International Conference on Embedded Software*, pages 164–172, New York, NY, USA, 2005. ACM Press.

[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. Pinisetty, P. Roop, S. Smyth, S. Tripakis, and R. von Hanxleden. Runtime enforcement of reactive systems using synchronous enforcers. In *Proc. International SPIN Symposium on Model Checking of Software (SPIN '17)*, Santa Barbara, CA, USA, July13–14 2017.

[12] S. Pinisetty, P. S. Roop, S. Smyth, S. Tripakis, and R. von Hanxleden. Runtime enforcement of cyber-physical systems. *ACM Transactions on Embedded Computing Systems, Special Issue for ESWEEK/EMSOFT '17*, 2017. In press.

[13] M. Samek. *Practical UML Statecharts in C/C++Event-Driven Programming for Embedded Systems*. Newnes, 2008.

[14] 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.

[15] A. Schulz-Rosengarten, S. Smyth, R. von Hanxleden, and M. Mendler. On reconciling concurrency, sequentiality and determinacy for reactive systems — a sequentially constructive circuit semantics for Esterel. In *2018 18th International Conference on Application of Concurrency to System Design (ACSD)*, pages 95–104, June 2018.

[16] A. Schulz-Rosengarten, R. von Hanxleden, F. Mallet, R. de Simone, and J. Deantoni. Time in SCCharts. In *Proc. Forum on Specification and Design Languages (FDL '18)*, Munich, Germany, Sept. 2018.

[17] S. Smyth, A. Schulz-Rosengarten, and R. von Hanxleden. Practical causality handling for synchronous languages. Technical Report 1808, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2018. ISSN 2192-6247.

[18] 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.

[19] 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)*, pages 372–383, Edinburgh, UK, June 2014. ACM.

[20] R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, O. O'Brien, and P. 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.

[21] N. Wechselberg, A. Schulz-Rosengarten, S. Smyth, and R. von Hanxleden. Augmenting state models with data flow. In M. Lohstroh, P. Derler, and M. Sirjani, editors, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, pages 504–523. Springer International Publishing, 2018.

[22] E. Yip, A. Girault, P. S. Roop, and M. Biglari-Abhari. The forec synchronous deterministic parallel programming language for multicores. In *10th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip, MCSOC 2016, Lyon, France, September 21-23, 2016*, pages 297–304, 2016.