Practical Causality Handling for Synchronous Languages

Steven Smyth, Alexander Schulz-Rosengarten, Reinhard von Hanxleden
Department of Computer Science, Kiel University, Kiel, Germany, {ssm, als, rvh}@informatik.uni-kiel.de

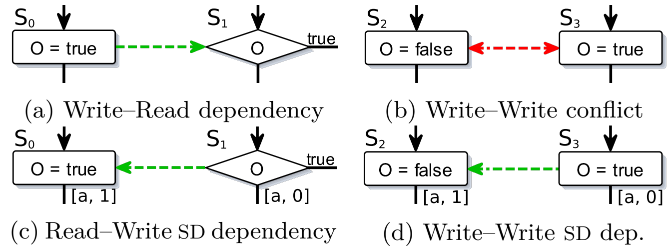
***Abstract*—A key to the synchronous principle of reconciling concurrency with determinism is to establish at compile time that a program is *causal*, which means that there exists a schedule that obeys the rules put down by the language. In practice it can be rather cumbersome for the developer to cure causality problems. To facilitate causality handling, we propose, first, to enrich the scheduling regime of the language to also consider explicit scheduling directives that can be used by either the modeler or model-to-model transformations. Secondly, we propose to enhance programming environments with dedicated *causality views* to guide the developer in finding causality issues. Our proposals should be applicable for synchronous languages; we here illustrate them for the SCCharts language and its open source development platform KIELER.**

***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). The write-before-read principle, employed in languages such as Esterel [3], 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. Issues 1) and 2) not only matter for the human developer, but also when transforming a program as part of a compilation.

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. 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. Please consult the associated technical report [8] for more details.



(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

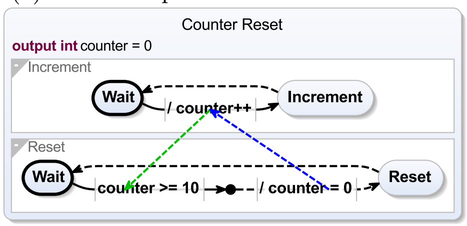
II. Scheduling Directives and Flexible Schedules

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*. 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 according to their index, lowest index first. For example, considering Fig. 1c, we may add an SD to each of the

```
scchart CounterReset {
  output int counter = 0

  region Increment:
  initial state Wait
    do counter++
      go to Increment

  state Increment
    immediate go to Wait

  region Reset:
  initial state Wait
    if counter >= 10 go to Do

  connector state Do
    immediate do counter = 0
      go to Reset

  state Reset
    immediate go to Wait
}
```

(a) Textual representation of Counter Reset

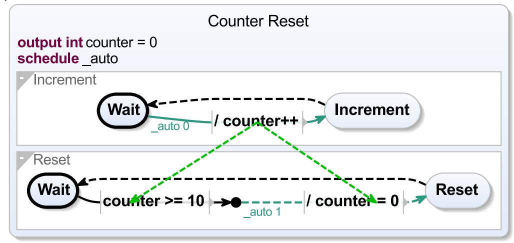


(b) Automatically generated graphical representation of Counter Reset

Figure 2: Concurrent Counter Reset program in SCCharts

```
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 3: Counter Reset example with SDs

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 [10] language. Fig. 2b shows an diagram of an SCCharts model, named Counter Reset. The textual source program is shown in Fig. 2a. The model has one integer *output* counter, which represents a counter 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 protocolstates 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. 2b. 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. 3a, 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. 3a, the SDs in lines 8 and 19 resolve the cycle 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, the modeler can interactive with the diagram to add SDs as detailed further in Sec. IV-A.

C. Scheduling Directives on Coarser Granularities

It is often sufficient to define SDs on a coarser granularity than the statement level. If statement-level SDs are available in the core language, coarse granularity SDs can be implemented as extended features, which can be

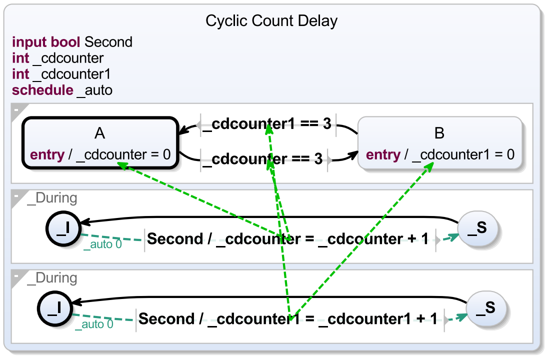


Figure 5: Cured expanded Cyclic Count delay

transformed automatically to statement-level SDs via M2M transformations. For example, 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 [7]. Even if the modeler does not use SDs directly, they can improve these transformations w.r.t. complexity and efficiency.

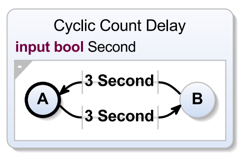


Figure 4: Cyclic count delay

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. 4.

A straightforward transformation which simply counts the occurrences as implemented by Motika [6] 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. 4, this simple approach fails, because of cyclic dependencies that are introduced by the M2M transformation 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. 5. Additionally, an arguably unintuitive

	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

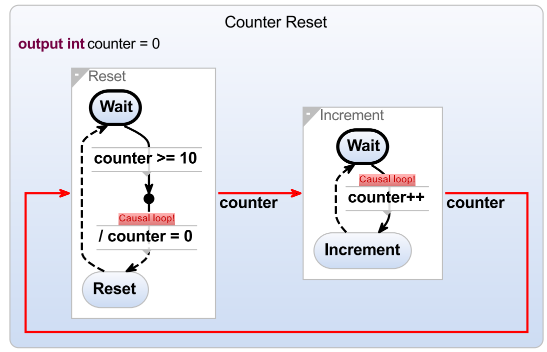


Figure 6: Counter Reset model shown with causality dataflow and annotations

reset to -1, 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. 4. 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 statements are executed more than once within one tick. Handling schizophrenia does not come trivially [9]. The SD solution avoids schizophrenia.

More practical examples can be found in the associated technical report [8].

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.

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 Fig. 2b 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. 2b), 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. 3b) is modified. A new schedule, named _auto, is declared as shown in Fig. 3a. 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

Fig. 6 shows the same model as Fig. 2b in a *causality dataflow view*, which shows a dependency cycle in red. 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 SCCharts compiler framework allows to create annotated models during compilation to hint at potential problems. The compilation error of the Counter Reset example will be detected during the scheduling phase. However, the issue is propagated back automatically and the causality loop warning is also displayed in the diagram as also depicted in Fig. 6.

V. Related Work

Many of the established synchronous languages, such as Esterel [3] and Lustre [4], use strict write-before-read MoCs. In this paradigm, even if not in a concurrent context, it is forbidden to change a value after it has been read from.

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 [11]. These languages can only access concurrent data from previous ticks, which cannot be modified any more.

Simulink/Stateflow [5] define the scheduling order depending on the graphical ordering of elements. 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.

If a program is rejected by the compiler, it is important to guide the user towards the problem. 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. 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 languages. 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.

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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.