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 compilers 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.

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
A. Causality in SCCharts

We exemplify modeling with SDs in the SCCharts language. Figure 2 shows a SCCharts model, 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 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 Figure 1d. Thus we have a scheduling cycle \( \text{counter} = 0 \rightarrow \text{moc} \rightarrow \text{counter} = 10 \rightarrow \text{moc} \rightarrow \text{counter} = 0 \). 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 Figure 3a, which is the Counter Reset example from Figure 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 \( \langle \text{scheduling unit} \rangle \text{ schedule name} \) (schedule index). In Figure 3b, 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 Section 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
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.

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 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.

<table>
<thead>
<tr>
<th></th>
<th>Simple</th>
<th>with Pre</th>
<th>with SDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedulable</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Schizophrenia</td>
<td>–</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Variables</td>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>States</td>
<td>22</td>
<td>44</td>
<td>18</td>
</tr>
<tr>
<td>Regions</td>
<td>5</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Binary Size (b)</td>
<td>–</td>
<td>2702</td>
<td>1337</td>
</tr>
</tbody>
</table>

**TABLE I: Results of the different count delay approaches in SCCharts**

Figure 4: Cyclic count delay

Figure 5: Cured expanded Cyclic Count delay

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

**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. 3d. 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. [7]. 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 [9]. 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 PRETC [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 SCADEx, 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


