Compiling SCCharts — A Case-Study on Interactive Model-Based Compilation *

Christian Motika, Steven Smyth, and Reinhard von Hanxleden

Real-Time and Embedded Systems Group, Department of Computer Science
Christian-Albrechts-Universität zu Kiel, Olshausenstr. 40, 24118 Kiel, Germany
www.informatik.uni-kiel.de/rtsys/
{cmot,ssm,rvh}@informatik.uni-kiel.de

Abstract. SCCharts is a recently proposed statechart language designed for specifying safety-critical reactive systems. We have developed an Eclipse-based compilation chain that synthesizes SCCharts into either hardware or software. The user edits a textual description which is visualized as SCChart and subsequently transformed into VHDL or C code via a series of model-to-model (M2M) transformation steps. An interactive environment gives the user control over which transformations are applied and allows the user to inspect intermediate transformation results. This *Single-Pass Language-Driven Incremental Compilation* (*SLIC*) approach should conceptually be applicable to other languages as well. Key benefits are: (1) a compact, light-weight definition of the core semantics, (2) intermediate transformation results open to inspection and support for certification, (3) high-level formulations of transformations that define advanced language constructs, (4) a divide-and-conquer validation strategy, (5) simplified language/compiler subsetting and DSL construction.

1 Introduction

Sequentially Constructive Statecharts (*SCCharts*) are a recently proposed statechart modeling language for reactive systems [13]. SCCharts have been designed with safety-critical applications in mind and are based on the sequentially constructive model of computation (SC MoC) [15]. The SC MoC follows a synchronous approach, which provides semantic rigor and determinism, but at the same time permits sequential assignments within a reaction as is standard in imperative languages. The basis of SCCharts is a minimal set of constructs, termed *Core SCCharts*, consisting of state machines plus fork/join concurrency. Building on these core constructs, *Extended SCCharts* add expressiveness with a rich set of advanced features, such as different abort types, signals, or history transitions. The safety-critical focus of SCCharts is reflected not only in the deterministic semantics, but also in the approach to defining the language, building up on Core SCCharts, which facilitate rigorous formal analysis and verification.

* This work was supported by the German Science Foundation (DFG HA 4407/6-1 and ME 1427/6-1) as part of the PRETSY project.

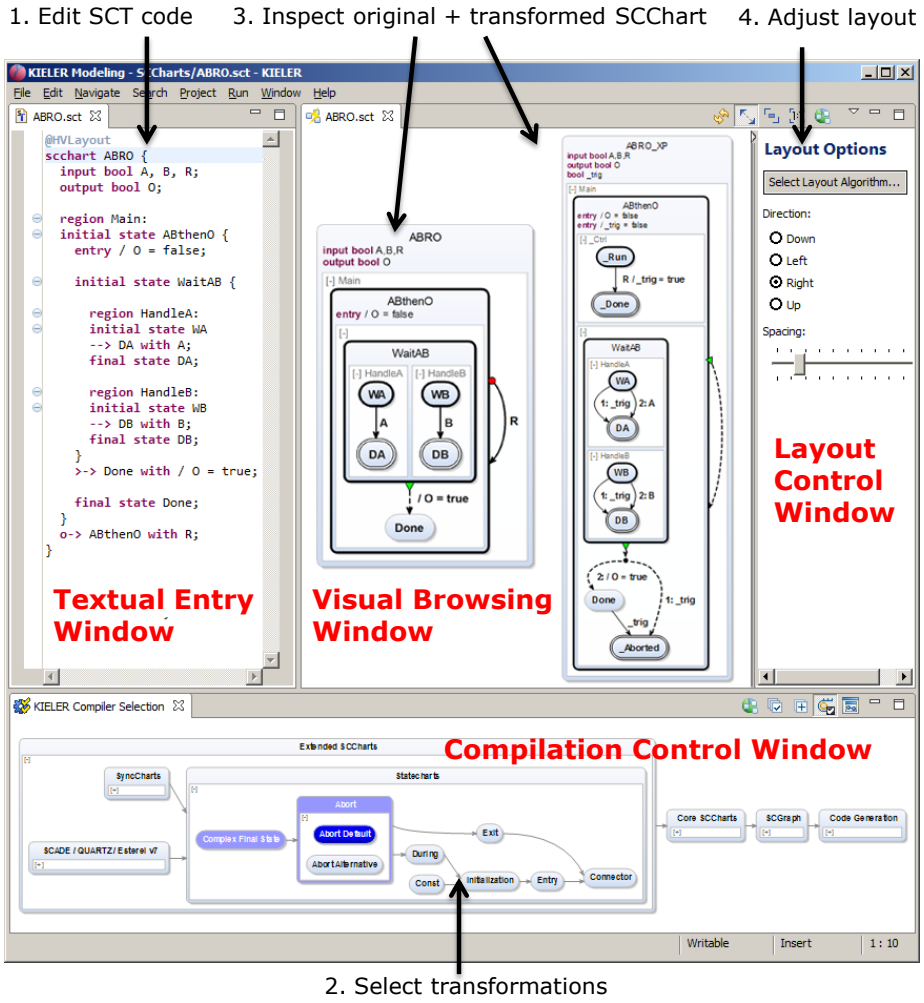


Fig. 1. Screen shot of KIELER SCCharts tool annotated with high-level user story for interactive model-based compilation

The original SCCharts language proposal [13] also presents possible compilation strategies for compiling SCCharts into software (e. g., C code) or hardware (e. g., VHDL). That presentation covers the abstract compilation concepts, largely specific to SCCharts. However, it gives only little detail and motivation on our *incremental, model-based* strategy for realizing these concepts, which is the focus of this paper now.

To get a first idea of this incremental model-based compilation approach and the possibilities it offers, consider the user story depicted in Fig. 1: (1) The user edits a model in a *textual entry window*. In our SCCharts prototype, this is

done with the SCCharts Textual Language (SCT). (2) The user selects model-to-model (M2M) transformations to be applied to the model in a *compilation control window*. In our prototype, these transformations are a series of incremental compilation steps from a textual SCChart (SCT) to C or VHDL. (3) The user inspects visual renderings, synthesized by modeling tool in the *visual browsing window*, of both (a) the original SCChart that directly corresponds to the SCT description, before applying the transformation, and (b) the transformed SCChart. (4) The user may fine-tune the graphical views of the SCChart in the *layout control window*. The visual browsing window is updated whenever any input in any of the other three windows changes. For a modeler, the possibility to view not only the original model, but also the effects that different transformation/compilation phases have on the model can help to understand the exact semantics of different language constructs and to fine-tune the original model to optimize the resulting code. Furthermore, the tool smith can validate the compiler one language feature at a time. This compiler validation support is desirable for any language and compiler; it is essential for safety-critical systems.

In contrast, the traditional modeling and software synthesis user story is: (1) The user edits/draws one view of a model. (2) A compiler parses the model and synthesizes code. (3) The user may inspect the final artefacts, such as a C file. This is appropriate for advanced users who are very familiar with the modeling language. However, it offers little guidance for the beginner. Also, this hardly allows to fine-tune and optimize the intermediate and/or resulting artifacts. Furthermore, and perhaps even more importantly, the compiler developer has little support here.

Outline and Contributions

The next section covers the SCCharts language, as far as required for the remainder of this paper, introduces the ABRO example, and presents an overview of the compilation of SCCharts.

A main contribution of this paper, which should be applicable outside of SCCharts as well, is the Single-Pass Language-Driven Incremental Compilation (SLIC) approach presented in Sec. 3. We discuss how to determine whether features can be successively transformed in a single sequence, how to derive a transformation schedule, guiding principles for defining transformations and how to build feasible language subsets.

Another contribution of this paper, which is more specific to SCCharts and synchronous languages, is the transformation sequence from ABRO to an equivalent SCChart presented in Sec. 4.

We give some implementation notes in Sec. 5, summarize related work in Sec. 6 and conclude in Sec. 7.

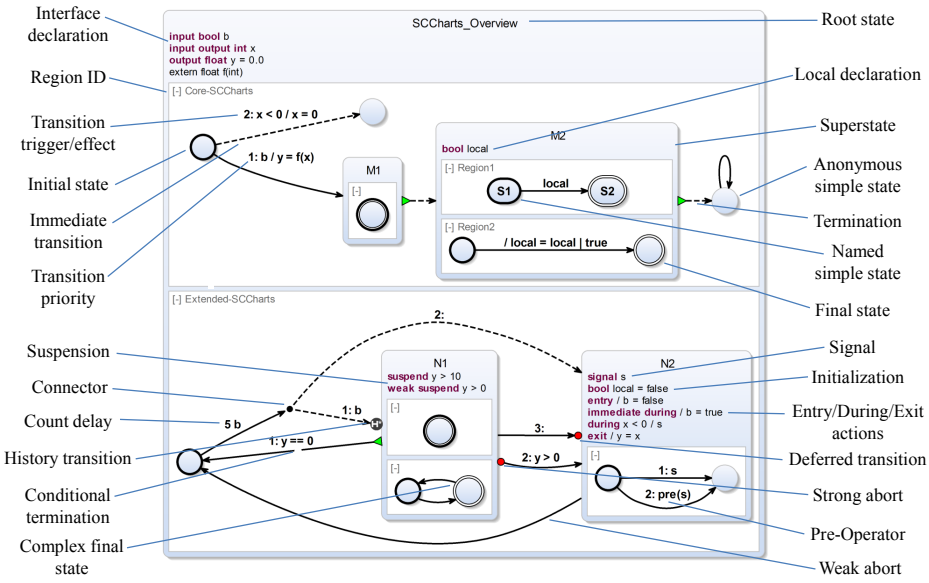


Fig. 2. Syntax overview. The upper region contains Core SCCharts elements only, the lower region illustrates Extended SCCharts.

2 SCCharts & Compilation Overview

An overview of the SCCharts visual language is shown in Fig. 2. The upper part illustrates Core SCCharts; the lower region contains elements from Extended SCCharts.

2.1 Core SCCharts Language Elements

Interface/Local Declarations. An SCChart starts at the top with an *interface declaration* that can declare variables and external functions. Variables can be *inputs*, which are read from the environment, and/or *outputs*, which are written to the environment. At the top level, this means that the environment initializes inputs at the beginning of the tick (stimulus), e. g., according to some sensor data, and that outputs are used at the end of a tick (response), e. g., to feed some actuators. The interface declaration also allows the declaration of local variables, which are neither input nor output. Declarations of local variables may also be attached to inner states as part of a *local declaration*.

States and Transitions. The basic ingredients of SCCharts are *states* and *transitions* that go from a *source state* to a *target state*. When an SCChart is in a certain state, we also say that this state is *active*. Transitions may carry a *transition label* consisting of a *trigger* and an *effect*, both of which are optional. When a transition trigger becomes true and the source state is active, the transition is

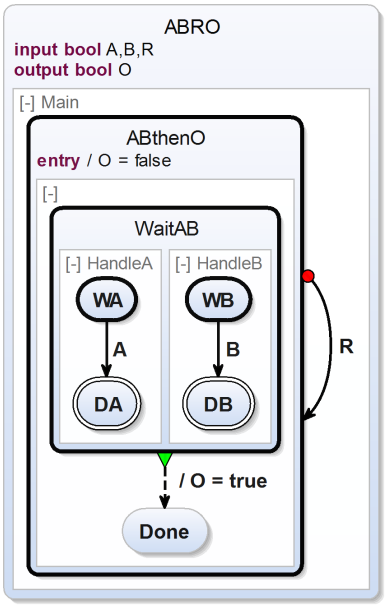
taken instantaneously, meaning that the source state is left and the target state is entered in the same tick. However, transition triggers are per default *delayed*, meaning that they are disabled in the tick in which the source state just got entered. This convention helps to avoid instantaneous loops, which can potentially result in causality problems. One can override this by making a transition *immediate*, which is indicated graphically by a dashed line. Multiple transitions originating from the same source state are disambiguated with a unique *priority*; first the transition with priority 1 gets tested, if that is not taken, priority 2 gets tested, and so on. If a state has an immediate outgoing transition without any trigger, we refer to this transition as *default transition* because it will always be taken. Furthermore, if additionally there are no incoming deferred transitions, we say that the state is *transient* because it will always be left in the same tick as it is entered. When taken, deferred transitions preempt all immediate behavior (including leaving) of the target state they are connected with.

Hierarchy and Concurrency. A state can be either a *simple state* or it can be refined into a *superstate*, which encloses one or several concurrent *regions* (separated region compartments). Conceptually, a region corresponds to a thread. A region gets entered through its *initial state* (thick border), which must be unique to each region. When a region enters a *final state* (double border), then the region *terminates*. A superstate may have an outgoing *termination* transition (green triangle), also called (unconditional) *termination transition*, which gets taken when all regions of this superstate have reached a final state. Termination transitions may be labeled with an action, but do not have an explicit trigger label; they are always immediate (indicated by the dashed line).

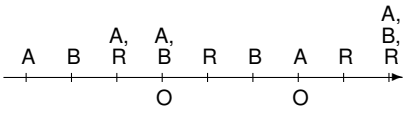
2.2 The ABRO Example

The ABRO SCChart (Fig. 3a and also Fig. 1), the "hello world" [1] of synchronous programming, compactly illustrates concurrency and preemption. The reset signal R triggers a *strong abort* (red circle) of the superstate ABthenO, which means that if R is present, ABthenO is instantaneously re-started.

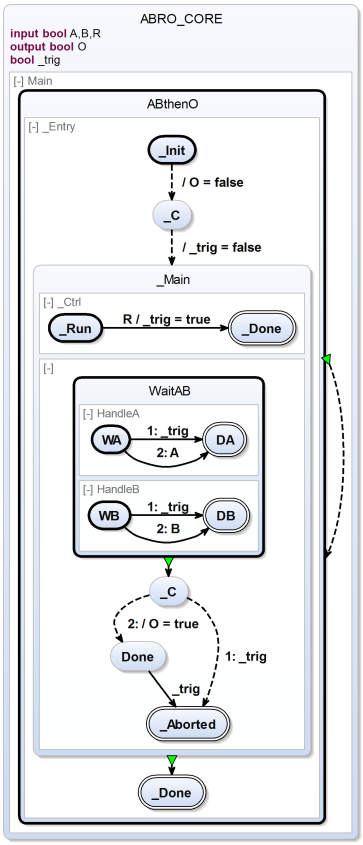
The execution of an SCChart is divided into a sequence of logical ticks. The interface declaration of ABRO states that A and B are Boolean inputs and O is a Boolean output. The execution of this SCChart is as follows. (1) The system enters initial state ABthenO as well as WaitAB. When entering ABthenO the entry action sets the output O to false. WaitAB consists of two regions (threads) HandleA and HandleB. Transitioning into a superstate does not trigger transitions nested within that state unless those transitions are immediate. The initial states WA and WB of both concurrent regions are also entered. (2) HandleA stays in its initial state WA, until the Boolean input A becomes true. Then it transitions to the final state DA. Similary, HandleB stays in its initial state WB, until the Boolean input B becomes true. Then it transitions to the final state DB. (3) When both threads



(a) Original ABRO Extended SCChart



(b) Possible execution trace with true-valued inputs above the tick time line and true-valued outputs below



(c) Equivalent Core SCChart after high-level compilation (expansion)

Fig. 3. ABRO, illustrating Extended and Core SCCharts features and the result of consecutive transformations from an Extended SCChart into an equivalent Core SCChart.

eventually are in their final states DA and DB, immediately the termination transition from WaitAB to Done is taken which is setting the output O to true. (4) The behavior can be reset by setting the input R to true. Then the self-loop transition from and to ABthenO is triggered causing a strong preemption and a re-entering of that state. This causes the entry action to reset the output O to false. The strong preemption means that the output O will not be true in case R is true in the same tick when the termination transition from WaitAB to Done is taken.

The exact semantics of ABRO is expressed by the equivalent ABRO_CORE (Fig. 3c), which only uses Core SCCharts language elements.

The ABRO example (Fig. 3a) illustrates some significant concepts of Core and Extended SCCharts. Core features are tick-boundaries (delayed transitions),

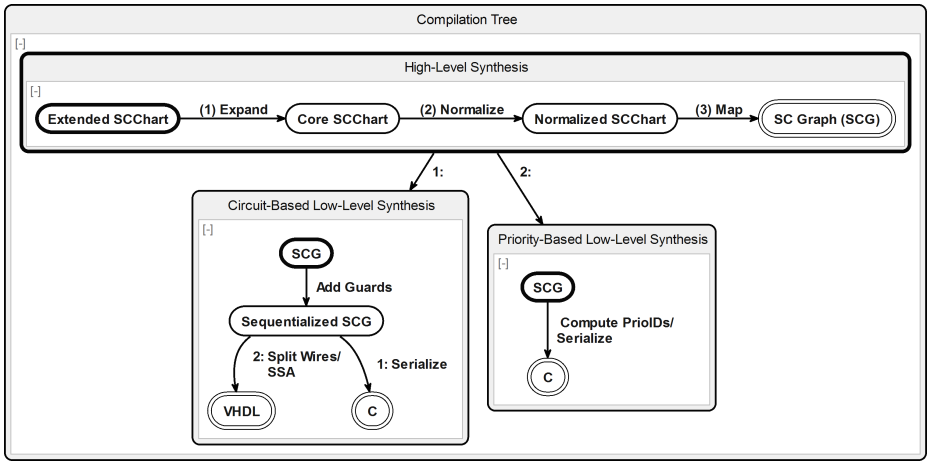


Fig. 4. Full compilation tree from Extended SCCharts to VHDL or C code splits into a high-level part and two different low-level parts.

concurrency (with forking and joining), and deterministic scheduling of shared variable accesses. Extended features are the concept of preemption by using a strong abort transition type for the self-loop transition triggered by R and the entry action for initializing or resetting the output O to false.

2.3 Compilation Overview

The full compilation tree is illustrated in Fig. 4, using Statecharts notation. In a way, this compilation tree, where incremental compilation steps correspond to the edges, is the dual to the compilation control window (Fig. 1), where the compilation steps correspond to the nodes.

The compilation splits into a high-level and a low-level part. The high-level compilation involves (1) expanding extended features by performing consecutive M2M transformations on Extended SCCharts, (2) normalizing Core SCCharts by using only a small number of allowed Core SCCharts patterns, and (3) straightforward (M2M) mapping of these constructs to an *SC Graph* (*SCG*).

An SCG is a pair (N, E), where N is a set of statement nodes and E is a set of control flow edges. The node types are *entry* and *exit* connectors, *assignments*, *conditionals*, *forks* and *joins*, and *surface* and *depth* nodes that jointly constitute tick-boundaries. The edge types are *flow* edges (solid edges), which denote instantaneous control flow, *pause* tick-boundary edges (dotted lines), and *dependency* edges (dashed edges), added for scheduling purposes. The SCG of ABRO that results after applying (2) normalization and (3) mapping to the core version (cf. Fig. 3c) is shown in Fig. 5.

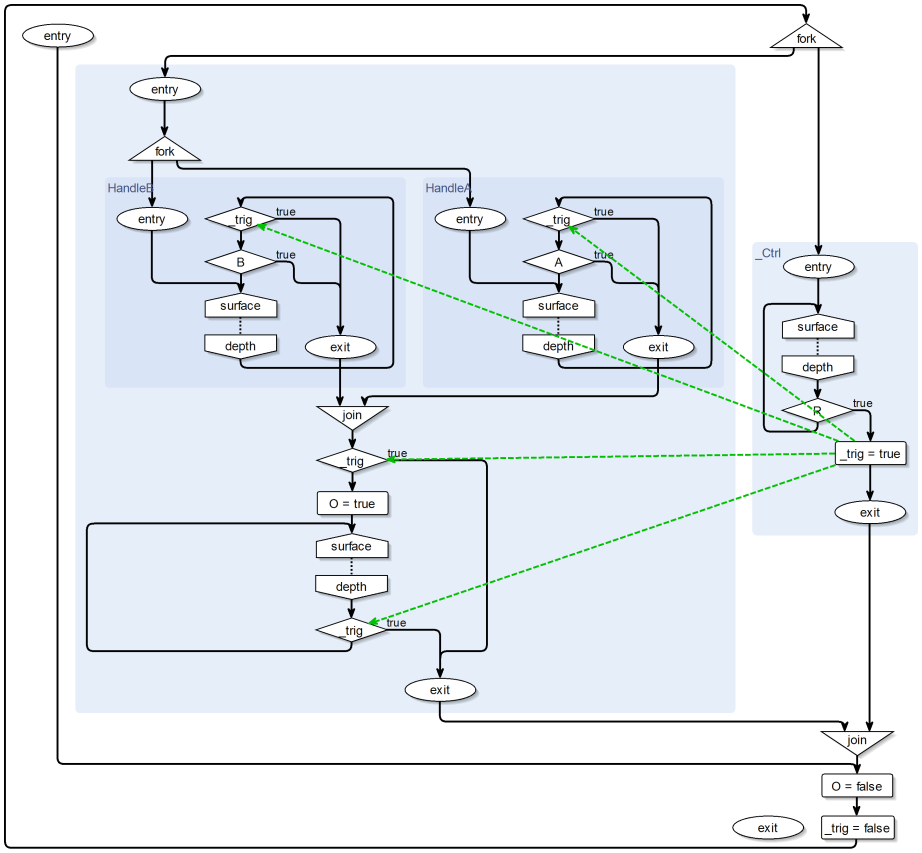


Fig. 5. The ABRO SC Graph (SCG). Dependencies (dashed edges) are used to sequentialize the SCG in further low-level compilation steps.

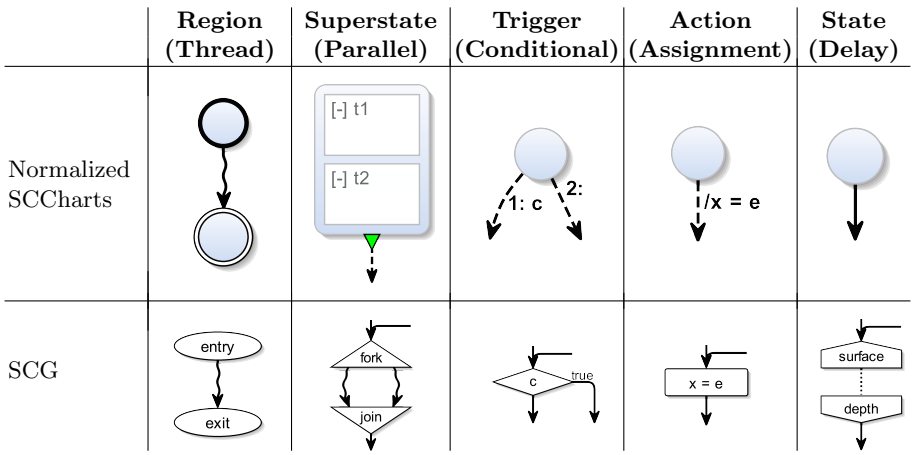
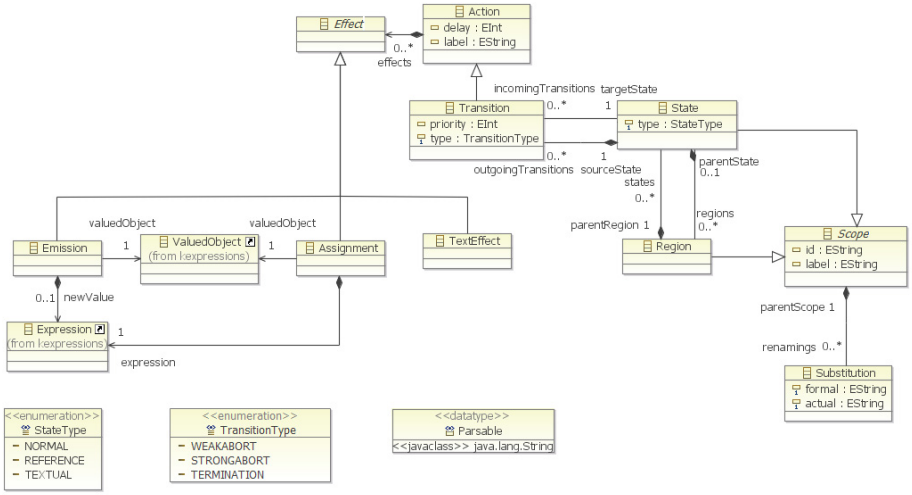
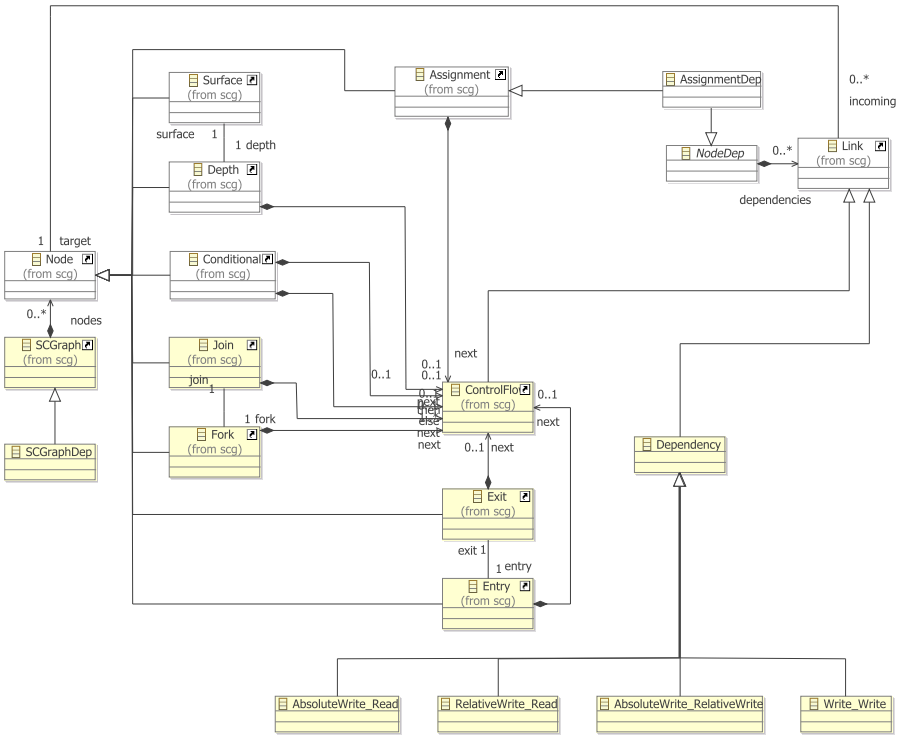


Fig. 6. Direct mapping from Normalized (Core) SCCharts constructs to SCG elements



(a) SCCharts Meta Model



(b) SCG Meta Model

Fig. 7. EMF Meta Models used in SCCharts compilation

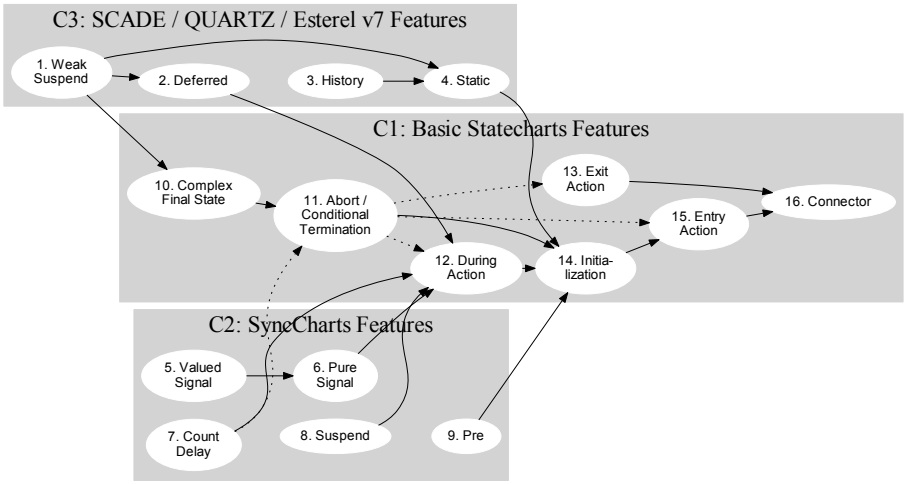


Fig. 8. Extended SCCharts features with their SLIC schedule index and their interdependencies

The normalization of Core SCCharts restricts the patterns to be one of the constructs shown in the upper part of Fig. 6, which also illustrates how Normalized SCCharts can be mapped directly to SCG elements.

As illustrated in Fig. 7, the meta models of SCCharts and SCGs are both fairly light-weight, but quite different. Technically, SCGs are just another representation of Normalized SCCharts to facilitate further compilation steps. The low-level transformation steps (cf. Fig. 4) also involve semantics-preserving M2M transformations. Then the resulting sequentialized SCG, e. g., is used directly to derive executable VHDL or C code [13].

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

We propose to break down rather complex compilation/synthesis tasks, such as the transformation of arbitrary SCCharts that may contain extended features into Core SCCharts, into a sequence of smaller transformation steps. Each transformation step should handle just one language feature at a time. We call this *single-pass language-driven incremental compilation* (SLIC). This approach is not fundamentally new, the concepts of syntactic sugar and language preprocessors are quite related. We here advocate to exploit this paradigm specifically for purposes of user feedback and tool validation.

The SLIC approach has several advantages:

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

When developing a SLIC transformation sequence, two non-trivial questions arise:

Q1 Does a linear, single-pass transformation sequence suffice?
Q2 If so, how must we order the individual transformation steps?

These questions are answered by the transformation relations presented next.

3.1 Transformation Relations

Given a set of language features F, we propose to define each feature $f \in F$ in terms of a *transformation rule* T_f that expands a model (program) that uses f into another, semantically equivalent model that does not use f. More precisely, T_f produces a model not containing f, but possibly containing features in $Prod_f \subseteq F$. Also, T_f can handle/preserve a certain set of features $Handle_f \subseteq F$. Note that $Handle_f$ must include f.

Based on *Prod* and *Handle*, we define the following relations on F:

Production order: $f \rightarrow_p g$ iff $g \in Prod_f$. We say that "T_f produces g."
Handling order: $f \rightarrow_{nhb} g$ iff $f \notin Handle_g$ ("f is not handled by T_g").
SLIC order: $f \rightarrow g$ iff $f \rightarrow_p g$ or $f \rightarrow_{nhb} g$ ("T_f must precede T_g").

Now we can answer the two questions from above. On Q1: A linear, single-pass transformation sequence suffices iff the SLIC order is acyclic. On Q2: We must order the individual transformation steps according to the SLIC order.

If the SLIC order is acyclic, we can implement a static *SLIC schedule*, which assigns to each $f \in F$ a *schedule index* $i(f)$ such that $f \rightarrow g$ implies $i(f) < i(g)$.

3.2 A SLIC Order for SCCharts Compilation

We now discuss the SLIC order for compiling SCCharts. We focus on the "Expand Extensions" part (see compilation overview, (1) Expand in Fig. 4), but the same principles apply to the other compilation steps as well.

Extended SCCharts provide a set F of *extended features*, listed in Fig. 8. The Extended SCCharts features are grouped into three categories:

C1: Basic Statecharts features. Common features of various statecharts dialects as known from Harel statecharts [5], e. g., entry actions, exit actions or strong and weak preemption.

C2: SyncCharts features. Extended SCCharts are quite rich and include, for example, all of the language features proposed for SyncCharts [1], e. g., synchronous signals or suspension.

C3: Further features. Extended SCCharts include additional features adopted from other synchronous languages such as weak suspension from Quartz [10] or deferred transitions from SCADE. We also categorize History transitions here for language subsetting purposes (cf. Sec. 3.4), even though they were part of the original Harel statecharts.

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

That the SLIC order for SCCharts is acyclic can be validated by visual inspection of Fig. 8, where all features $f \in F$ are ordered left-to-right according to $\rightarrow_p$ (solid arrows) and $\rightarrow_{nhb}$ (dotted arrows). We can also see the SLIC schedule, as each $f \in F$ is prefixed with a "$i(f)$." label that shows its schedule index.

Concerning the feature categories C1, C2, and C3, we observe that inter-category precedence constraints are only of type C2 $\rightarrow$ C1 or C3 $\rightarrow$ C1. Thus we can modularize our schedule according to categories: First transform away all features from C3, then all features from C2, and finally all features from C1.

Referring back to the interactive user story depicted in Fig. 1, note that the compilation control window presents a customizable, slightly abstract view of the transformations and their dependencies depicted in Fig. 8. The user can customize this view by collapsing/expanding parts of the compilation chain. In Fig. 1, the user has chosen to expand the Statecharts node, corresponding to C1, and has selected the Abort Default transformation to be applied (thus shown in dark blue). The tool automatically selects all "upstream" *required* transformations (light blue) as well, as such that the Abort Default transformation is not confronted with any language constructs it cannot handle.

3.3 Designing the Transformation Rules

Whether the SLIC order is acyclic or not is not an inherent property of the language features themselves, but depends on how exactly the transformations for the features are defined. For example, we might have defined our transformation rules T_f such that each extended feature f would be transformed directly into Core SCCharts by T_f alone ($Prod_f = \emptyset$), while preserving all other features

($Handle_f = F$). This would have resulted in an empty SLIC order that would be trivially acyclic. However, this would have defeated the purpose of modularizing the compilation, as at least some of the transformation rules would have to be unnecessary complex.

Instead, we wish the transformation for each f to be rather lean. For that purpose T_f may make use of other features, as reflected by a non-empty $Prod_f$. Furthermore, in defining T_f, we may restrict the models to be transformed to not contain all features in F, meaning that $Handle_f$ may be small. However, care must be taken to not introduce cycles this way. This implies that the more "primitive" a feature f is, the more features T_f it must be able to handle. Furthermore, there is often a trade-off between on the one hand lean transformation rules where some features undergo a long sequence of transformations and on the other hand compact, efficient transformation results.

3.4 Language Subsetting / Constructing DSLs

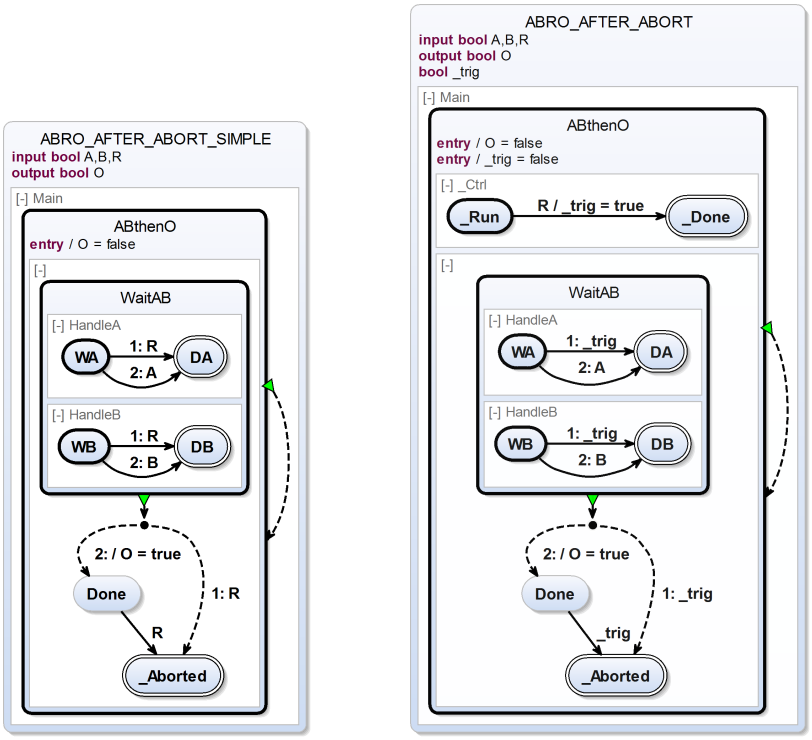
Given a language L with a set of language features F, a tool smith may wish to offer a derived language L' that offers only a subset $F' \subseteq F$ of language features to the user. For example, the SCCharts language proposal is very rich, which nicely illustrates how a wide range of different features proposed in SyncCharts, SCADE etc. can be grounded in a small set of Core SCCharts features. However, this variety of features may be overwhelming for the user. Also, some features might be rarely used in practice or not be appropriate for certain domains (such as, in our experience, suspension), or might be considered non-desirable for some reasons (such as history transitions, which increase the state space drastically).

Given a feature set F and a production order $\rightarrow_p$, we say that $F' \subseteq F$ is a *feasible subset* iff for all $f \in F'$ and $g \in F$, $f \rightarrow_p g$ implies $g \in F'$. In other words, the transformations of the features in F' do not produce any features outside of F'.

A conservative approach to ensure subset feasibility would include in F' all features whose SLIC schedule index is above a certain value. E. g., for SCCharts, if we define F' such that it includes all features with schedule index 10 and higher, we would obtain all features in category C1, which would be a feasible language subset. However, the definition of subset feasibility permits other subsets as well. E. g., the subset of SCCharts features with indices 11, 14, 15, 16, which includes Aborts (index 11) and all subsequently produced features, would also be feasible.

4 Example: An M2M Transformation Sequence from Extended ABRO to Core ABRO

This section uses the ABRO example to illustrate how selected Extended SCCharts features are incrementally transformed into Core SCCharts. Further details for these transformations and generalizations of the presented transformations are given elsewhere [12].



(a) Simple approach (b) WTO approach

Fig. 9. Transformation for Abort

4.1 Aborts

A hierarchical state can be *aborted* upon some trigger. The ability to specify high-level aborts is one of the most common motivations for introducing hierarchy into statecharts. Aborts are thus a powerful means to specify behavior in a compact manner, but handling them faithfully in simulation and code synthesis is not trivial. There are two cases to consider, *strong aborts*, which get tested before the contents of the aborted superstate gets executed, and *weak aborts*, which get tested after the contents of the aborted state get executed.

Consider the original ABRO Extended SCChart as shown in Fig. 3a. ABthenO is left by a self-loop strong abort transition triggered by R. This abort takes place regardless of which internal state of ABthenO is active at the time (tick) of an abort. In case of nested superstates with aborts, this transformation must be applied from the outside in, so that inner aborts can also be triggered by outside abort triggers.

Fig. 9a illustrates how expanding ABRO results in an equivalent SCChart that does not use the extended feature Abort anymore. The underlying idea is to make the internal regions of WaitAB terminate explicitly whenever ABthenO

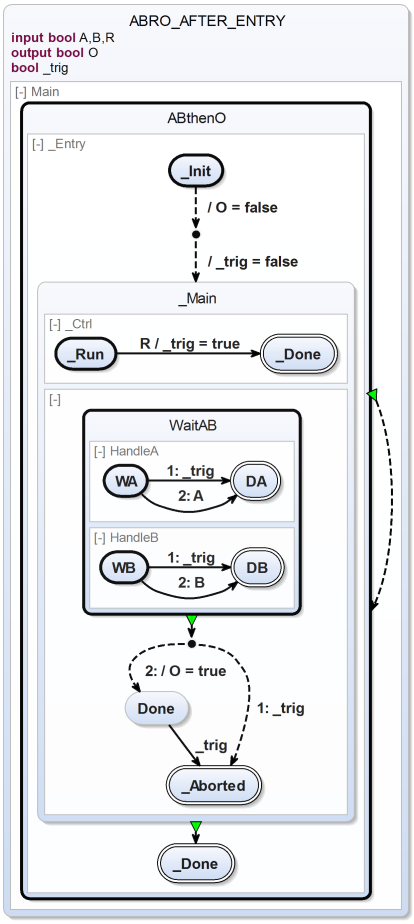


Fig. 10. Transformation for Entry Action

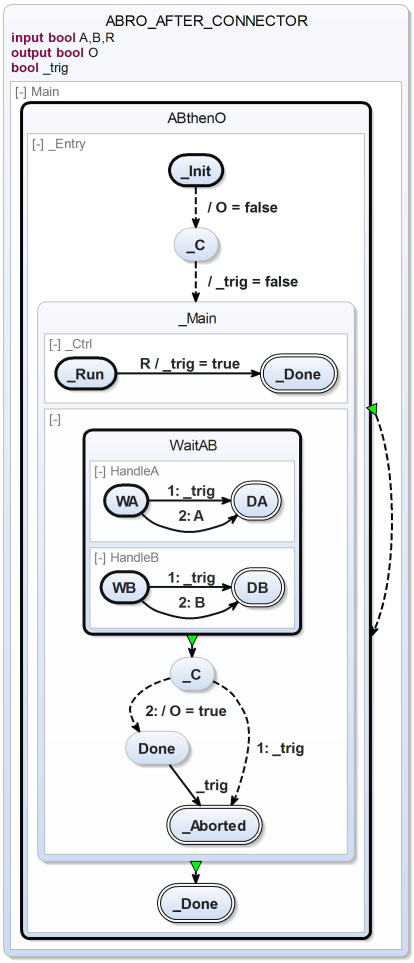


Fig. 11. Transformation for Connector

is aborted, and then use a termination transition to leave WaitAB. Note that strong abortion has the highest priority and thus the transitions triggered by R have the highest transition priority 1. Also note that in Fig. 9a the condition R was duplicated 4 times. This may result in multiple evaluations of R and thus violates the *Write-Things-Once* (WTO) principle. This may not be problematic if the condition consists of a single flag (in this case R), but can be an issue if the condition consists of a costly expression; down-stream synthesis may or may not be able to optimize this again by applying, e. g., common subexpression elimination. Fig. 9b shows an alternative transformation that meets the WTO principle by concurrently evaluating R just once and triggering the abort using an auxiliary variable _trig.

4.2 Entry Actions

When eliminating the extended abort feature from ABRO, the WTO-variant of the Aborts-transformation produced an auxiliary variable _trig together with an entry action for resetting it to false. Entry actions also are extended SCCharts features and hence need to be eliminated during compilation to Core SCCharts. As indicated in Fig. 8, entry actions must be transformed after aborts. Note that entry actions do not get moved outside of the state that they are attached to, hence entry actions can also make use of locally declared variables.

When a state S has an associated entry action A, then A should be performed whenever S is entered, before any internals of S are executed. If multiple entry actions are present, they are performed in sequential order. A non-trivial issue when defining this transformation is that we would like to allow entry actions to still refer to locally declared variables. Hence we cannot simply attach entry actions to incoming transitions, as these would then be outside of the scope of local variables. Our transformation handles this issue by handling all entry actions within the state they are attached to. This also handles naturally the case of initial states, which do not have to be entered through an incoming transition.

The transformation result after further transforming ABRO using the entry action transformation is shown in Fig. 10. The entry actions were inserted before the original initial state inside ABthenO. A new auxiliary initial state _Init and connectors for sequential ordering of all auxiliary transitions (one for each entry action) are used. Entry actions are executed instantaneously, hence all transitions are immediate.

4.3 Connectors

The last feature to eliminate in order to transform the ABRO Extended SCChart (cf. Fig. 3a) into a Core SCChart (cf. Fig. 3c) are connectors.

Connector nodes, sometimes also referenced as *conditional nodes*, link multiple *transition segments* together to form a *compound transition*. Connectors typically serve to make a model more compact, and to facilitate the WTO principle, without the introduction of further (transient) states.

Our approach to transform connectors is simply to replace each connector by a state which must be a transient state that is entered and immediately left again as part of a transition. Therefore, all outgoing transitions must explicitly be made immediate. This can be seen in Fig. 11.

5 Implementation

The SCCharts tool prototype[1] (cf. Fig 1) is part of KIELER[2] and uses the KLighD diagram synthesis framework [9] for graphical visualization of textually

[1] http://www.sccharts.com
[2] http://www.rt.informatik.uni-kiel.de/kieler

```
1  def void transformConnector(State state) {
2      // If a state is of type connector, then apply the transformation
3      if (state.type == StateType::CONNECTOR) {
4          // Set the state type to normal
5          state.setTypeNormal
6          // Explicitly set all outgoing transitions to be immediate transitions
7          for (transition : state.outgoingTransitions) {
8              transition.setImmediate(true)
9          }
10     }
11 }
```

Fig. 12. Xtend implementation of transforming connector states

modeled SCCharts. We implemented all transformations from Extended SCCharts to Core SCCharts, the normalization, the SCG mapping and all SCG transformations (cf. Sec. 2.3) as M2M transformations with Xtend[3]. To illustrate the compact, modular nature of the M2M transformations, Fig. 12 shows the Connector transformation described in Sec. 4.3. Xtend keywords and Xtend extension functions are highlighted. The precondition is checked in line 3, i. e., whether the considered state is a connector state. Line 5 sets the type of this state from connector to normal. Finally, lines 7-9 ensure that all outgoing and previously implicit immediate transitions of this state are now being set explicitly to be immediate transitions. As can be seen, the transformation description is straight-forward and of limited complexity.

For modeling SCCharts the textual editor shown in Fig 1 is used. We generated it using the Eclipse based Xtext framework which produces a full-featured textual editor for the SCCharts Textual Language (SCT) with syntax highlighting, code completion and built-in validation. More specifically, this editor is generated from an SCT Xtext grammar description declaring the actual concrete textual syntax for the SCCharts meta model elements (cf. Fig. 7a).

We defined the SCCharts and the SCG transformations on the EMF meta models. The extended and the normalization transformations of the high-level synthesis are so-called "inplace" model transformations because they modify the SCChart model that conforms to the SCCharts meta model shown in Fig. 7a. The SCG mapping transformation is defined on both the SCCharts meta model (cf. Fig. 7a) and the SCG meta model (cf. Fig. 7b). The low-level synthesis, e. g., the sequentialization of SCGs is again defined as several consecutive inplace model transformations all only based on the SCG meta model.

6 Related Work

Statecharts, introduced by Harel in the late 1980s [5], have become a popular means for specifying the behavior of embedded, reactive systems. The visual syntax is intuitively understandable for application experts from different domains and the statechart concepts of hierarchy and concurrency allow the expression

[3] http://www.eclipse.org/xtend/

of complex behavior in a much more compact fashion than standard, flat finite state machines. However, defining a suitable semantics for the statechart syntax is by no means trivial, as evinced by the multitude of different statechart interpretations. In the 1990s, von der Beeck identified a list of 19 different non-trivial semantical issues, and compared 24 different semantics proposals [11], which did not even include the "official" semantics of the original Harel statecharts (clarified later by Harel [6]) nor the many statechart variants developed since then, including, e. g., UML statecharts with its run-to-completion semantics. One critical issue in defining a statecharts semantics is the proper handling of concurrency, which has a long tradition in computer science, yet, as argued succinctly by Lee [7], has still not found its way into mainstream programming languages such as Java. Synchronous languages were largely motivated by the desire to bring determinism to reactive control flow, which covers concurrency and aborts [2]. SCCharts have taken much inspiration from André's SyncCharts [1], introduced as Safe State Machines (SSMs) in Esterel Studio. However, SCCharts are more liberal that SyncCharts in that they permit multiple variable values per reaction as long as the SC MoC can guarantee determinism.

Edwards [3] and Potop-Butucaru et al. [8] provide good overviews of compilation challenges and approaches for concurrent languages, including synchronous languages. We present an alternative compilation approach that handles most constructs that are challenging for a synchronous languages compiler by a sequence of model-to-model (M2M) transformations, until only a small set of Core SCCharts constructs remains. This applies in particular to aborts in combination with concurrency, which we reduce to terminations.

The incremental, model-based compilation approach using a high-level transformation language (Xtend) allowed us to build a compiler in a matter of weeks and to validate it in a divide-and-conquer manner. Furthermore, the ability to synthesize graphical models, with a high-quality automatic layout, lets the user fully participate in this incremental transformation, as illustrated in the interactive model-based compilation user story in the introduction. This fits very well with the *pragmatics-aware* modeling approach [14], which advocates to separate models from their view and to let the modeling tool generate customized views that highlight certain model aspects. In this light, we might say that the interactive model-based transformation provides the user with different views of one and the same model that differ in abstraction level, from the possibly very abstract model designed by the user all the way down to the implementation level.

7 Conclusions

The incremental, model-based compilation approach presented here did not originate from a desire to develop a new, general approach to synthesis, but rather was the outcome of building a compiler for a specific language, SCCharts. In fact, when building this compiler we did intend to re-use existing approaches and technologies as much as possible. Furthermore, the main purpose of M2M

transformation rules that constitute the compiler was originally to unambiguously define the various extended SCCharts features; we were positively surprised to find that they also produce fairly compact, efficient code as well [13]. In the end, the desire to quickly prototype a modular compiler, easy to validate and to customize, prompted us to follow the SLIC approach presented here; and Xtext, Xtend, KIELER and KLighD, all part of Eclipse, were the key enabling technologies for the implementation.

When asking what exactly is "model-based" about the SCCharts compilation approach, one notices that indeed there are many similarities to traditional compilation approaches. For example, the SCCharts with their hierarchical structure might also be considered a form of abstract syntax tree (AST), and the SCG is related to other intermediate formats used in compiling synchronous languages. However, the SLIC approach is model-driven in the following aspects:

- The compilation steps are M2M transformations where the resulting model contains all information. There are no other, hidden data structures.
- For the most part, the intermediate transformation steps are in the same language as the original model. We just apply a sequence of language sub-setting operations, transforming away one feature at a time. There is a change of language when going from normalized SCCharts to the SCG, but that is mainly for convenience, for example, to be able to separate the surface of a pause from its depth. However, even that step would not have been strictly necessary, we could have stayed with the SCCharts meta model all the way to the final C/VHDL code. In fact, our first implementation had only one meta model.

We see numerous directions for future work. For example, we want to explore further the best ways on how to let the user interact with the compiler and how to manage the model views. Especially for larger models we want to employ techniques like reference states to gain modularization and preserve scalability. Regarding scalability and practicability we hope to report on an ongoing larger case study soon. In this case study our approach is used for designing and implementing a complex model railway controller. The SLIC order for SCCharts, depicted in Fig. 8, has evolved over time, and we expect it to evolve further. For example we currently explore tool support for consistent choices of selected transformations, statically from the SLIC order and dynamically from the features used in concrete models. Also, we are experimenting with alternative transformation rules for one and the same feature, where the choice of the best rule may depend on the original model and overall constraints/priorities. Another active area is that of interactive timing analysis [4], where we investigate how to best preserve timing-information across M2M transformations. The main advantage of our approach is its interactivity. Nonetheless we envision a fully automatic compilation process including the possibility to include our compiler in scripts (e. g., a Makefile) or using it online in the Web.

References

1. C. André. Semantics of SyncCharts. Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.
2. A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages twelve years later. In *Proc. IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, Piscataway, NJ, USA, January 2003. IEEE.
3. S. A. Edwards. Tutorial: Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems*, 8(2):141–187, April 2003.
4. I. Fuhrmann, D. Broman, S. Smyth, and R. von Hanxleden. Towards interactive timing analysis for designing reactive systems. Technical Report UCB/EECS-2014-26, EECS Department, University of California, Berkeley, April 2014.
5. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
6. D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.
7. E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.
8. D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*. Springer, May 2007.
9. 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, 15–19 September 2013.
10. K. Schneider. The synchronous programming language Quartz. Internal report, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2010. http://es.cs.uni-kl.de/publications/datarsg/Schn09.pdf.
11. M. von der Beeck. A comparison of Statecharts variants. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 128–148. Springer-Verlag, 1994.
12. 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. Technical Report 1311, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2013. ISSN 2192-6247.
13. 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.
14. 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, December 2012.
15. R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, and O. O'Brien. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. In *Proc. Design, Automation and Test in Europe Conference (DATE'13)*, pages 581–586, Grenoble, France, March 2013. IEEE.