Interactive Model-Based Compilation Continued

Incremental Hardware Synthesis for SCCharts

Francesca Rybicki, Steven Smyth, Christian Motika,
Alexander Schulz-Rosengarten, 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/
{fry,ssm,cmot,als,rvh}@informatik.uni-kiel.de

Abstract. The Single-Pass Language-Driven Incremental Compilation (SLIC) strategy uses a series of model-to-model (M2M) transformations to compile a model or program to a specified target. Tool developer and modeler can inspect the result of each transformation step, using a familiar, graphical syntax of the successively transformed model, which is made possible by harnessing automatic layout. Previous work (presented at ISoLA'14) introduced the basics of the SLIC approach and illustrated it with a compiler that translated SCCharts, a synchronous, deterministic statechart language developed for safety-critical systems, to software. The compiler is implemented in the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER), an open-source development framework based on Eclipse.
This paper proposes two extensions to SLIC. First, we extend the M2M transformation mechanism with a *tracing* capability that keeps track of model elements during transformations. Second, we make use of the tracing capability for an interactive *simulation*, where we not only observe a model's input/output behavior during execution, but can inspect the run-time behavior of each model component, at any transformation stage. We illustrate these concepts by new transformations in the KIELER SCCharts compiler, which allow to synthesize hardware circuits, and a simulator that executes an intermediate-level software model and visualizes the simulation at the high-level model as well as the low-level circuit.

1 Introduction

In an earlier case-study on interactive model-based compilation [12], we investigated possible compilation strategies for *Sequentially Constructive Statecharts* (*SCCharts*). SCCharts [21] is a synchronous statechart modeling language for reactive systems, designed with safety-critical systems in mind. Due to its sequentially constructive model of computation [22] it provides semantic rigor and determinism. At the same time, it permits sequential assignments within a reaction, which is forbidden in classical synchronous languages.

The case-study also introduced *Single-Pass Language-Driven Incremental Compilation* (*SLIC*). The user story is as follows: i) A user edits a textual model.

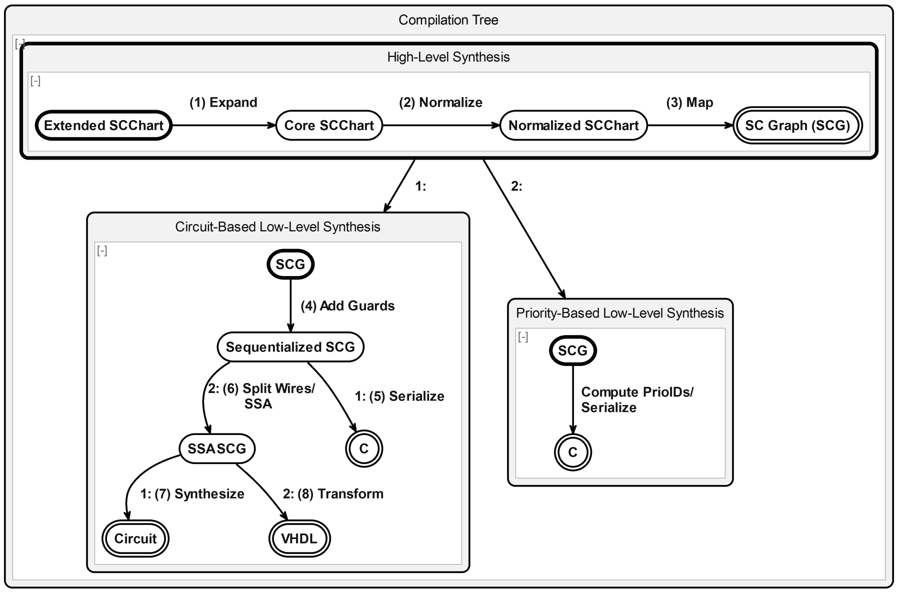


Fig. 1. Full compilation tree from Extended SCCharts to hardware (e. g., VHDL) or software (e. g., C code) splits into a high-level and low-level parts (adapted from [12]).

ii) The user selects a chain of M2M transformations to be applied to the source model. iii) The selected transformations are applied and the visual representation of the transformed model is updated.

Unlike in traditional compilers, each step of the transformation chain is fully transparent and each intermediate result can be inspected. As long as the M2M transformations are within the same meta model, the same graphical syntax that is already familiar to the user can be used to visualize the transformation results. The tool smith can validate and optimize each step of the compiler. There are no hidden (intermediate) data structures that carry additional information. This is particularly useful for safety-critical systems.

The original introduction of the SLIC approach [12] implemented the compilation tree depicted in Fig. 1, using Statecharts notation, and explained the *high-level synthesis* part in detail. The SCCharts language is split into two parts: Extended SCCharts, the syntactic sugar, and Core SCCharts, the minimal language set. 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) straight-forward M2M mapping of these constructs to a *Sequentially Constructive Graph* (*SCG*).

The low-level synthesis strategies involve the code generation for software (e. g., C code) and hardware (e. g., VHDL) as presented earlier [21], with the data-flow approach for software, (4) and (5), explained in detail elsewhere [18].

Modeling and Programming

Traditionally, one way to separate programming and modeling was based on whether the primary concrete syntax of the high-level artifact was graphical ("model") or textual ("program"). However, if that distinction was ever justified, it is now less and less so as textual modeling frameworks such as Xtext become more common place, and these frameworks provide standard compilation services such as parser generation. The SLIC approach and its extensions proposed here is related to the fields of program compilation and modeling alike. We address a classical compilation task, namely translating a high-level artifact developed by a human to a low-level, executable piece of hardware or software. While doing so, we make systematic use of classical modeling concepts and a widely used modeling ecosystem (Eclipse).

We thus see the work presented here as yet another step towards blurring the boundary between modeling and programming. This not only concerns the "technical" aspects of how programs/models are analyzed and synthesized, but also how the programmer/modeler is involved in the process. The choice of concrete syntax is just one example; other examples, which we invite the reader to consider in the remainder of this paper, include the way the compiler is controlled using a "model" of the compilation chain (as illustrated in Fig. 1), how intermediate compilation results are presented, or how a program/model is simulated.

Contributions

In this paper, we present two extensions to the SLIC approach. First, we extend the M2M transformation mechanism with a *tracing* capability that keeps track of model elements during transformations. This allows to map high-level model elements to their low-level counterparts and vice versa. Second, we make use of the tracing capability for an interactive *simulation*. As in source-level debugging familiar from high-level languages, the execution state is reflected in the original model, but here we can inspect the run-time behavior of each model component at any transformation stage as well.

To illustrate these concepts, we further explore the picture given in Fig. 1 by explaining how to create hardware circuits from models written in SCCharts within the SLIC approach. This includes (6) the transformation of the SCG into a Single Static Assignment (SSA) [15] form and (7) the generation of the circuits via M2M transformations as well as the visualization and simulation. The simulation is done based on an intermediate transformation result that determines a *tick function*, which is compiled to C code that is then executed. The bidirectional transformation tracing information not only allows to map simulation results to the original model, but also to the hardware circuit that is the final result of the transformation.

Outline

The next section covers the SCCharts language, as far as required for the remainder of this paper. The section introduces AO, a subset example of ABRO presented in the previous paper [12]. AO will serve as ongoing example for the circuit synthesis.

Sec. 3 then discusses the two proposed extensions to the SLIC, tracing and simulation, at a general level. The interactive incremental hardware synthesis that illustrates these extensions follows in Sec. 4. That section explains the transformations that are necessary to create circuits and how to extend the existing toolchain to simulate and validate the generated circuits. The evaluation for the interactive hardware synthesis, showing the practicability of the approach, is discussed in Sec. 5.

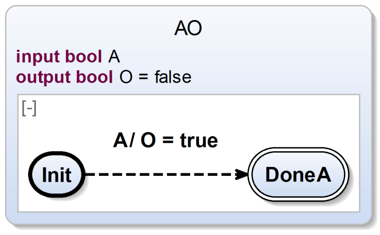
We summarize related work in Sec. 6, and conclude in Sec. 7.

2 SCCharts

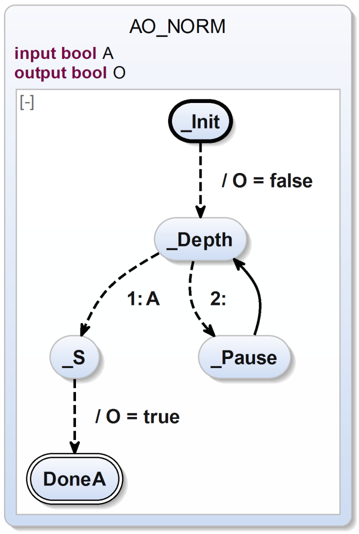
In this section, we will introduce the AO SCChart, see Fig. 2a, a tiny example of SCCharts. We choose AO because of space considerations. Nevertheless, the approach presented here applies any SCChart that is statically scheduable, e. g., ABRO, the "hello world" [1] of synchronous programming, included in the previous case-study [12]. We will explain all used features of SCCharts as far as they are necessary to understand the model. In depth details of the SCCharts language are described in the introductory SCCharts paper [21] and the technical report on the features of the SCCharts language [20].

In general, an SCChart starts with an *interface declaration* at the top 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. One may also declare *local variables.* In AO the interface declaration consists out of one input variable A and one output variable O, which will be fed back to the environment at the end of a reaction cycle.

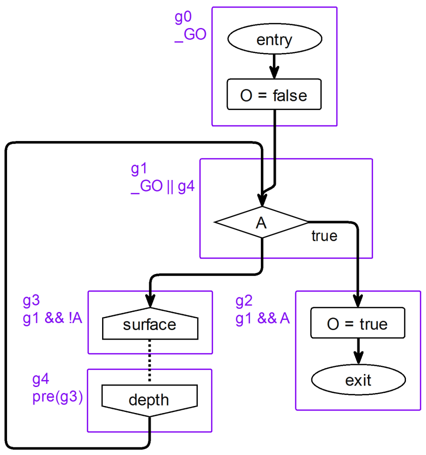
AO has only one (top) level of *hierarchy.* It includes its two *states* Init and DoneA which are connected via a *transition.* Since AO does not comprise any concurrency, only one of these states may be active. At the start of the program AO state Init, the *initial state* (thick border), is the active one. The program ends after DoneA, a *final state* (double border), is reached. If Init is active and the *trigger* of the transition, input A, is true, Init is left and DoneA becomes active. Simultaneously, the *effect* O = true is executed which sets O to true. If two or more transitions outgoing from the active state are eligible to run, the transition with the higher *priority* is taken (cf. _Depth in Fig. 2b). The transition in AO is an *immediate* transition, indicated by the dashed edge, which means that it is enabled as soon as its source state becomes active. Otherwise, it would have been



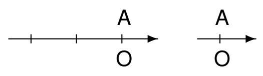
(a) Source model of the AO SCChart



(b) Equivalent Normalized SCChart after high-level compilation (expansion)



(c) SCG of AO depicting the control-flow of AO. The basic blocks (purple boxes) enclose the statements of the program. They are annotated with their name and the expression that determines when a block is active.



(d) Two possible execution traces with true-valued inputs above the tick time line and true-valued outputs below. The second trace emits O in the first tick because the transition in AO is taken immediately.

Fig. 2. The AO example, illustrating Extended and Normalized SCCharts features and its sequential control-flow

delayed by default, meaning it cannot trigger in the first tick of its source state. This convention prevents instantaneous cycles, which would be problematic for this synthesis.

AO_NORM (Fig. 2b) expresses the exact semantics of AO but uses only language elements of the Core SCCharts subset. The transformation from compact extended SCCharts to Normalized SCCharts and more key features of SCCharts, such as concurrency, hierarchy and preemption, are explained in detail elsewhere [12, 20].

According to the compilation strategies presented before [12, 18, 21] a normalized SCChart is mapped to its corresponding SCG (also see (3) in Fig. 1.). The SCG of the normalized version of AO (cf. Fig. 2b) is shown in Fig. 2c. The basic blocks (purple boxes) determine which part of the program is active in

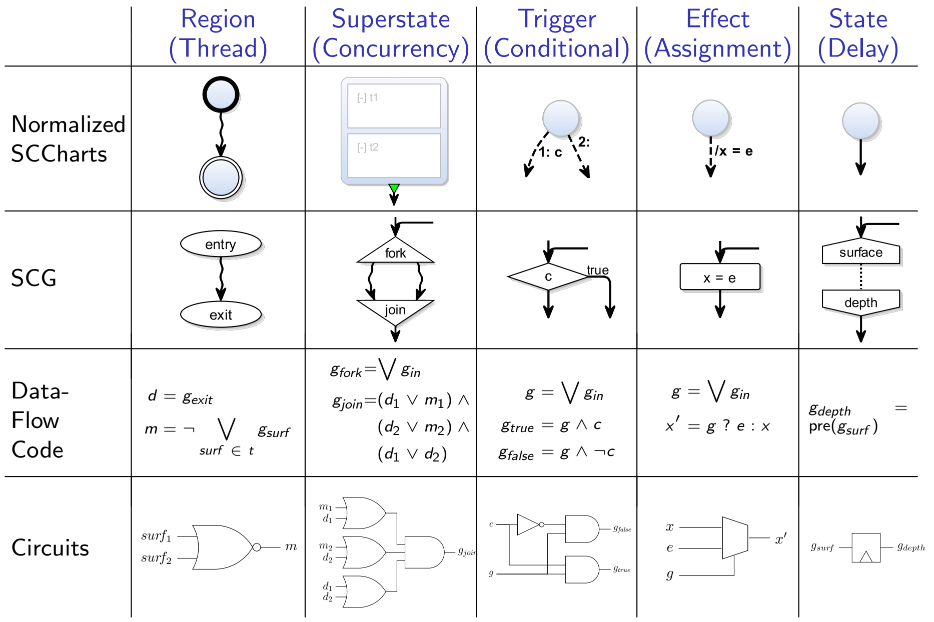


Fig. 3. Matrix showing the entire mapping throughout the transformation process from SCCharts to circuits (adapted from [21])

the actual tick. They are annotated with their activation expression, also called *guard*. The specific mapping from normalized SCCharts pattern to SCG elements is depicted in the upper part of Fig. 3. The lower part shows the direct mapping between SCG elements and data-flow code and their corresponding circuits.

The execution of an SCChart is divided into a sequence of logical ticks. Two example traces for AO can be seen in Fig. 2d. The program is terminated as soon as the reaction that emits O occurred. This includes the first tick of the program, in which Init becomes active, because the transition between Init and DoneA is immediate.

3 SLIC Extensions

In this section, we first recall the general SLIC user story [12]. Afterwards, the new extensions, namely tracing and simulation, are introduced. An extended SLIC compiler is able to present every intermediate result and propagate information, such as runtime information about a running simulation, between all intermediate model instances. The models may be instances of different meta models.

SLIC User Story

Fig. 4 shows a screenshot of the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER[1]) modeling tools for SCCharts annotated with a schematic workflow:

(1) A modeler models their model textually.
(2) The model gets displayed graphically. This is done instantly and achieved by using automatic layout techniques, such as KIELER's layouting tools.
(3) At any point in time, the modeler may select one or more transformations in the compiler selection. Subsequently, the model gets transformed and will be displayed as intermediate result. The tooling will only present transformations that match the input model or any model format that is reachable from the input model. As described in the original introduction of SLIC [12], the transformation steps are executed in a statically determined order and each transformation produces an intermediate result. This is depicted as directed chain of arrows in the box at the bottom of Fig. 4. The added tracing technology, explained in Sec. 3.1, allows a bidirectional mapping between all intermediate results.
(4) The results may be simulated. Therefore, an (intermediate) model is compiled to executable code. A simulation engine then runs the program and feeds back runtime information that can be used, e. g., to visualize the model instances.

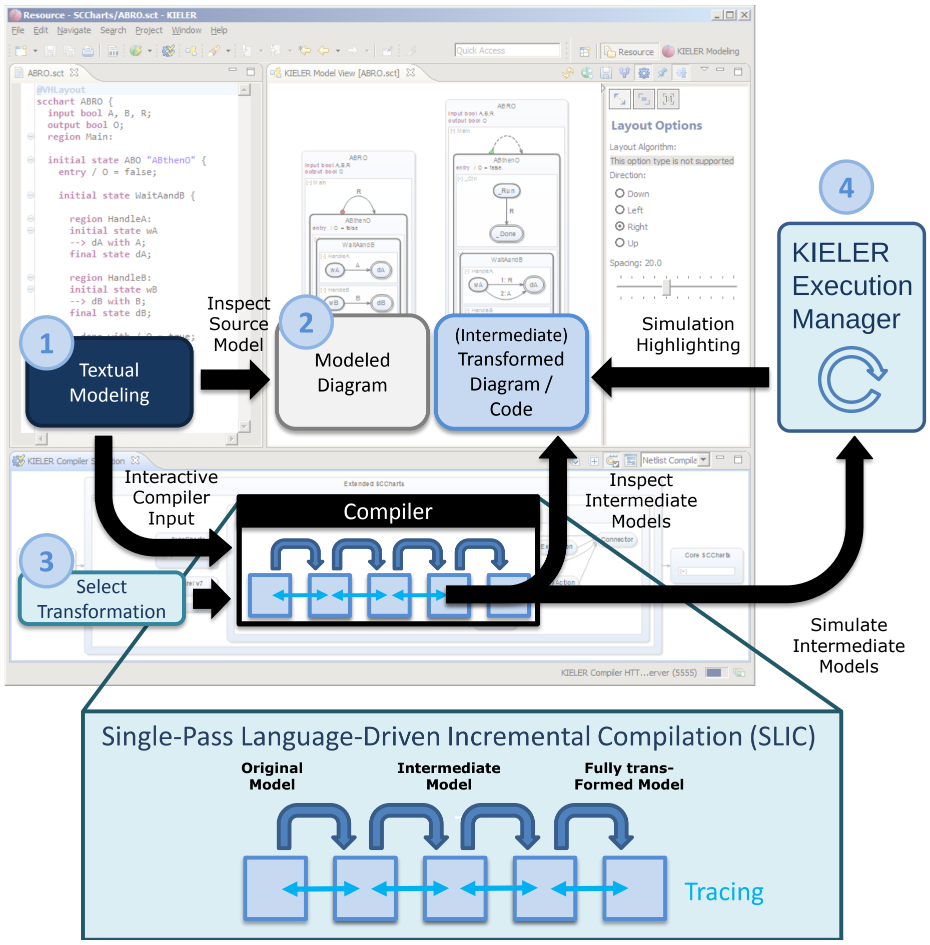
3.1 SLIC Extension: Tracing

The tracing of model elements creates a map that stores information about the relationships between different model elements w.r.t. intermediate transformation results. Transformation rules, introduced earlier [12], describe how model elements get transformed into different, new model elements. The newly generated model elements may be of the same or a different meta model. Fig. 5 shows a transformation step. Model I with nodes A, B, C, D, and E gets transformed to model II with nodes F, G, H, and I. We observe four kinds of element relations:

(1) Object A transforms to F and G. This depicts a 1:n relation.
(2) Node B translates to I in a 1:1 relation.
(3) Element D has no corresponding nodes in the target model.
(4) Nodes C and E both transform to node H, which depicts and n:1 relation.

Every transformation produces an intermediate compilation result, which is a fully functional model instance. It can be visualized and used as origin for further transformation steps. The changes that enable tracing capability to SLIC transformations are minimal because the developer must only add tracing information

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



```
@VHLayout
scchart ABRO {
  input bool A, B, R;
  output bool O;
  region Main:

  initial state ABO "ABthenO" {
    entry / O = false;

    initial state WaitAandB {

      region HandleA:
      initial state wA
      --> dA with A;
      final state dA;

      region HandleB:
      initial state wB
      --> dB with B;
      final state dB;
```

Resource - SCCharts/ABRO.sct - KIELER
File Edit Navigate Search Project Window Help
ABRO.sct
KIELER Model View [ABRO.sct]
Quick Access
Resource
KIELER Modeling
Layout Options
Layout Algorithm:
This option type is not supported
Direction:
Down
Left
Right
Up
Spacing: 20.0
1
Textual Modeling
Inspect Source Model
2
Modeled Diagram
(Intermediate) Transformed Diagram / Code
Simulation Highlighting
4
KIELER Execution Manager
Interactive Compiler Input
Compiler
Inspect Intermediate Models
3
Select Transformation
Simulate Intermediate Models
KIELER Compiler Selection
Extended SCCharts
Core SCCharts
Netlist Compile
KIELER Compiler HTT...erver (5555)
Single-Pass Language-Driven Incremental Compilation (SLIC)
Original Model
Intermediate Model
Fully trans-Formed Model
Tracing

Fig. 4. SLIC extended with tracing and an interactive simulation as implemented in the KIELER SCCharts tools

for newly created elements. Elements that are present in both, the source and the target model, are mapped to a 1:1 relation by default. Analogously, model elements without a target are also handled automatically.

For example, the *initialization* transformation of the KIELER SCCharts implementation, see Fig. 6, creates a new *entry action* for every initialization part of a declared variable. Therefore, it retrieves an iterator for objects with initialization part and creates an entry action for each at index 0 in reverse order to preserve the initialization order. The initialValue of the valuedObject will be added to the assignment of the entry action in Line 5. Hence, the expression is removed from the valuedObject containment. As mentioned before, this is implicitly traced and must not be added to the transformation rule explicitly. For the

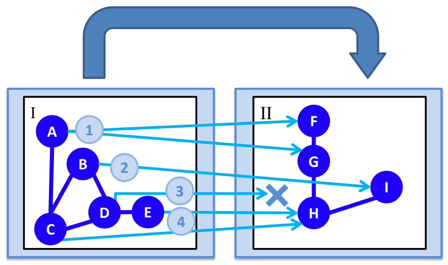


Fig. 5. Transformation step from model I to model II: Model element tracing depicted by arrows.

```
1 def transformInitialization(State state) {
2   val valuedObjects = state.valuedObjects.
      filter[initialValue != null].reverseView
3   for(valuedObject : valuedObjects) {
4     state.createEntryAction(0) => [
5       effects +=
          valuedObject.createAssignment(
          valuedObject.initialValue)
6       trace(valuedObject)
7     ]
8   }
9 }
```

Fig. 6. Xtend implementation of transforming variable initializations including tracing command

tracing, only Line 6 had to be added. This traces the newly created entry action back to valuedObject in a 1:1 relation.

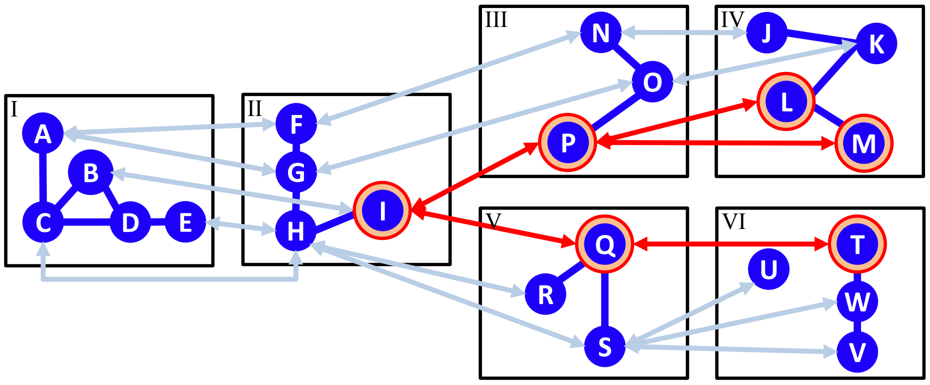
Applying the tracing to all transformations creates a *tracing tree*, which can be used to trace model information between arbitrary intermediate model instances of a complete compilation chain. Fig. 7a shows the application of the rules depicted in Fig. 5 for models I and II. Additionally, four subsequent transformations, creating the instances III, IV, V, and VI, form the model instance hierarchy seen in the figure. The tracing tree can be used to show the relationships between all model elements of a particular compilation. E. g., model element K in model IV corresponds to element A in model I via O in model III and G in model II.

Furthermore, it is not mandatory to always map from source to target or vice versa. The topology of the tracing tree w.r.t. the model instances can be seen in Fig. 7b. Since the tracing is transitive, the elements from, e. g., model instance IV can be used to trace relationships in, e. g., model instance VI as depicted in the figure. Here, model II serves as least common transformation result. For example, as depicted in red in Fig. 7a, elements M and L in model IV both relate to element T in model VI and vice versa.

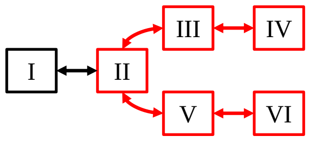
At the moment our tracing framework only allows the mapping from model instance to model instance and not, e. g., to pure text. Hence, the mapping of the last code generation step in the KIELER compiler must be done explicitly, if textual program code, such as C, is generated. However, this is only a tooling restriction of the current KIELER version. In principle, the tracing tree can also include the textual program data, e. g., represented as model.

3.2 SLIC Extension: Simulation

The user story told at the beginning of this section also depicts the possibility to simulate any intermediate model (cf. (4) in Fig. 4). If using a SLIC compiler equipped with transformation tracing as described in Sec. 3.1, each intermediate result can be simulated to gather runtime information about all other



(a) Utilizing the tracing tree to depict model element relationships. Nodes L and M in the model IV correspond to node T in model VI (shown in red). Both tracing paths originate from node I in model II.



(b) End-to-End mapping depicting the tracing path from model IV to model VI via model II. Information about model elements propagate transitively between the different model instances.

Fig. 7. Resolving model tracing

model instances. E. g., the model can be compiled down to C code and then be executed by a simulation engine which handles input/output communication. This technique is implemented in the *execution manager* [11] that is part of the KIELER modeling tools. Another way would be the execution of a model by an interpreter on model level. In both cases the runtime information could be propagated throughout the whole tracing tree. The modeler may choose, which model instances they want to inspect (cf. (3) in Fig. 4).

Hence, considering the example depicted in Fig. 7a in Sec. 3.1, any model of models I – VI may be executed to gather runtime information. E. g., if model VI is executed and element T is active, the simulation deduces that elements L and M in model IV would also be active. The granularity of these deductions depends on the structure of the tracing branches of the tracing tree. The simulation sets an element to active as soon as at least one corresponding tracing element is also active. Therefore, while executing model IV, element T would be marked as active if element L or element M is active. Of course, this simulation convention may be changed. Depending on the actual use-case and transformation setup, it is for example conceivable to set an element to active only if all corresponding tracing elements are also active.

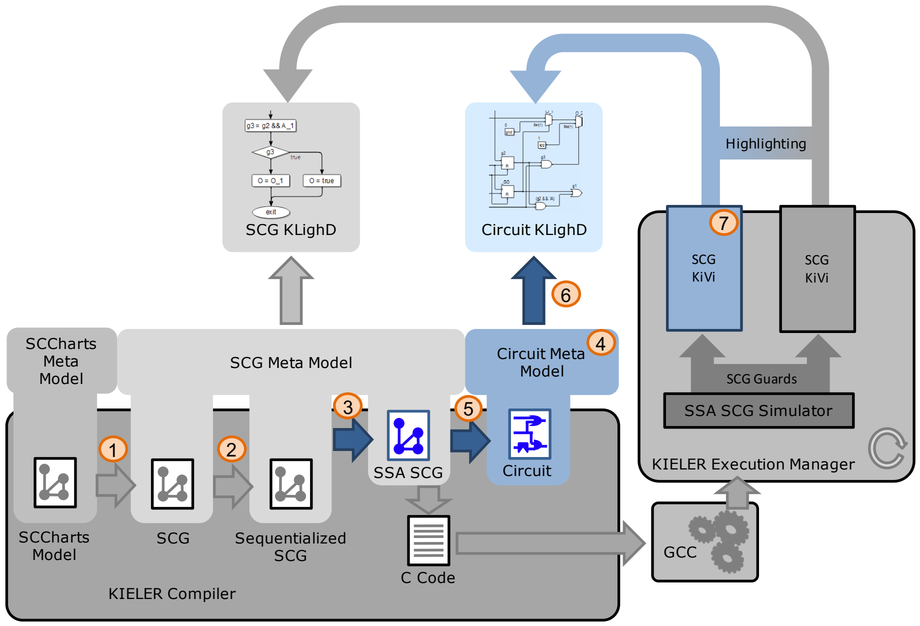
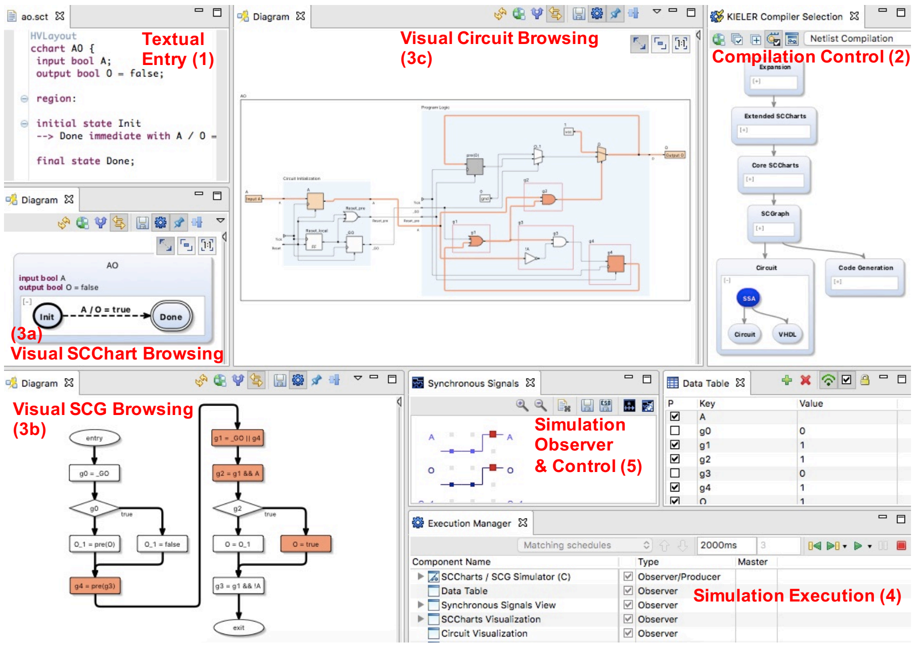


Fig. 8. Interactive Incremental Hardware Synthesis workflow overview. The new steps (3 – 7, marked in blue) fully integrate into the existing SLIC toolchain.

4 Interactive Incremental Hardware Synthesis

We used the SLIC approach, including the aforementioned extensions, to implement an interactive incremental hardware synthesis. Fig. 8 presents an overview of all necessary steps from 1 – 7 to realize the transformation from SCCharts into circuits. The incremental synthesis steps marked in blue (3 – 7) are the steps presented in this section. The overview shows how the hardware synthesis steps are integrated into the existing SLIC toolchain and how they depend on each other.

As indicated in Fig. 1 the transformation from sequentialized SCGs into circuits needs one intermediate step to generate an SSA form of the SCG (3). The SSA transformation (cf. Sec. 4.1) resolves data-flow dependencies appearing in the SCGs. Eventually, the actual transformation into circuits takes place (5) (cf. Sec. 4.2). The visualization of the circuits is another M2M transformation (6) and is done via KLighD [17]. Hence, the circuits get layouted fully automatically without the need to manually rearrange individual elements. Sec. 4.4 explains the mechanics of the simulation. The visualization of the SCG simulation is used to visualize the dynamic behavior of the circuit (7).



```
HVLayout
cchart AO {
 input bool A;
 output bool O = false;

 region:

 initial state Init
 --> Done immediate with A / O =

 final state Done;
```

P	Key	Value
✓	A	
☐	g0	0
✓	g1	1
✓	g2	1
☐	g3	0
✓	g4	1
✓	O	1

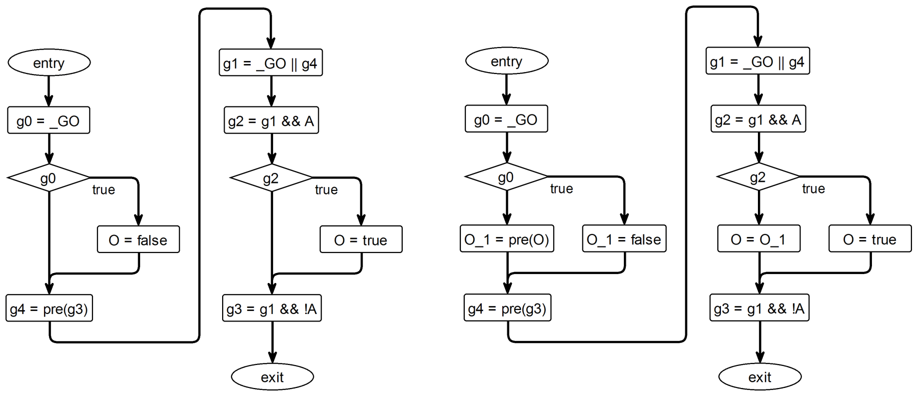
Component Name		Type	Master
SCCharts / SCG Simulator (C)	✓	Observer/Producer	
Data Table	✓	Observer	
Synchronous Signals View	✓	Observer	
SCCharts Visualization	✓	Observer	
Circuit Visualization	✓	Observer	

Fig. 9. Screenshot of KIELER SCCharts tool annotated with high-level user story for incremental interactive model-based hardware synthesis

User Story for Interactive Hardware Synthesis

Referring to the user story introduced in Sec. 3, the new incremental transformation for hardware synthesis and interaction is depicted in Fig. 9:

(1) The textual representation of the model is written in SCT, the textual language of SCCharts. In the screenshot, AO is shown.
(2) The interactive compilation control view allows the user to select different M2M transformations. Transformations may depend on each other. The new *feature group* Circuit (lower left part of (2) in Fig. 9) contains the interactive incremental hardware synthesis presented in this paper.
(3) The user may inspect the source model and the results of the transformations, selected in step (2), in the visual browsing windows. (3a) shows the source model, which is AO in this case. Since the user selected the appropriate transformations in (2), (3b) illustrates the SSA version of the SCG corresponding to the source model. (3c) depicts the resulting hardware circuit. If the source model gets modified in the editor (1), all views get updated.
(4) The user may add simulation components in the execution manager view to configure a simulation. The selected transformation (2) serves as input for the simulation. While executing a simulation, for each tick, the active components of the visualized model will be highlighted in all model views. This



(a) Sequential SCG of AO (b) SSA SCG of AO

Fig. 10. Transformation of the sequentialized SCG of AO into an SSA form

improves the dynamic comprehensibility of the model instances, and hence, of the circuit.

(5) During the execution of a simulation, the user may set input variables and observe the reaction of the system. This can also be done automatically by loading traces of previous simulations.

4.1 SSA SCG Transformation

The target of the SCCharts data-flow approach is a sequentialized form of an SCG which only consists out of *assignments* and *conditionals* [18]. In sequentialized SCGs multiple writes to one and the same variable are possible. To solve data-flow dependencies if such a variable is read, the SCG is transformed into an SSA form. As defined elsewhere [15] a program is in SSA form if each variable is the target of exactly one assignment in the program text. In the transformation of sequentialized SCGs into SSA, each variable which is the target of multiple assignments becomes indexed. If at some point of execution a variable is read from, the value of the assignment with the highest index at this point of execution is used.

For the SSA SCG transformation only variables defined by the user are relevant w.r.t. SSA because automatically created guard variables are unique by definition. In sequentialized SCGs, assignments to variables depend on conditionals. The assignments are executed if the corresponding guard is true. Hence, the *else branch* of these conditionals does not have any nodes. Fig. 10a depicts the sequential SCG of AO. As the first conditional node in this SCG shows, O shall not be modified if g0 is evaluated to false. Instead, it is desired that O still stores the unmodified value. The SSA transformation therefore adds assignment nodes on the *else branches*. If a condition is evaluated to false, those nodes

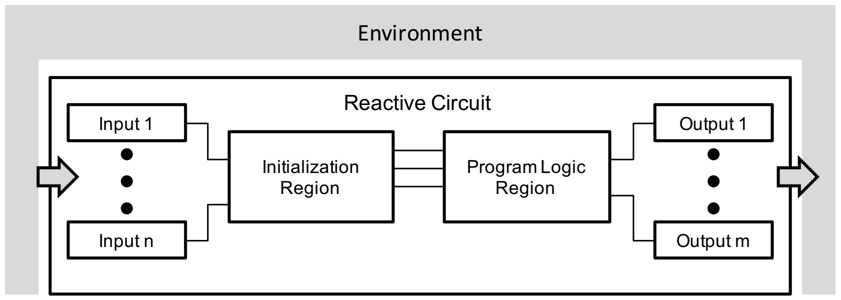


Fig. 11. The circuit and its regions in the context of the controlled environment

assign the latest version of a variable. This is shown in Fig. 10b. If guard g0 in the first conditional node evaluates to false, O_1 is target of the assignment pre(O). This means the value of O from the previous tick is applied to O_1, which reflects the fact that in SCCharts, variables are static and hence persist across tick boundaries. Otherwise, O_1 is set to false. Analogously, if guard g2 in the second conditional node is evaluated to false, O is set to O_1 which is the latest unmodified instance of O at this point of execution. As observable, the depicted SSA SCG is not in classical SSA form. In general, a ϕ-function decides which version of an SSA variable is used after possible modifications on different instances. However, the ϕ-function that decides which instance of a variable is chosen can directly be resolved in the conditional branches because from both branches always only one is executed exclusively [9, 16].

4.2 Circuit Transformation

According to Fig. 8, Step (5), the SSA SCG gets transformed into a circuit representation. SCCharts models are designed for reactive systems. The corresponding circuit is usually meant to be embedded in a reactive environment. Hence, sensor inputs are read from the environment, outputs are computed and then fed back to control the environment. Therefore, for each input (output) in the source model, a corresponding input (output) port is created in the circuit. The structure of the circuit is depicted in Fig 11. It is divided in two parts. The first part, the Initialization Region, provides the *reset* and *tick* logic. The second part, the Program Logic Region, contains the transformation of the SSA SCG and represents the logic of the program.

The exact translation rules are depicted in Fig. 3. Each SCG node corresponds to data-flow equation which can be translated directly into hardware circuits. For example, an assignment of the form $x = e$ in the SCG gets translated into a Multiplexer (MUX) element. The responsible guard g, which is the composition of its predecessor guards g_{in}, decides whether or not e is assigned to x. Therefore, g is connected to the select pin of the MUX and x and e serve as inputs. x' then becomes the new actual instance of x.

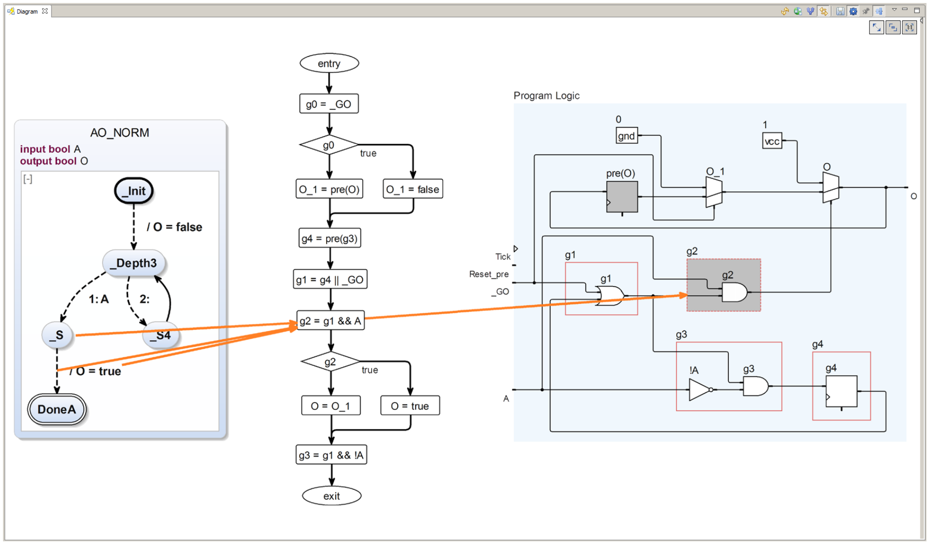


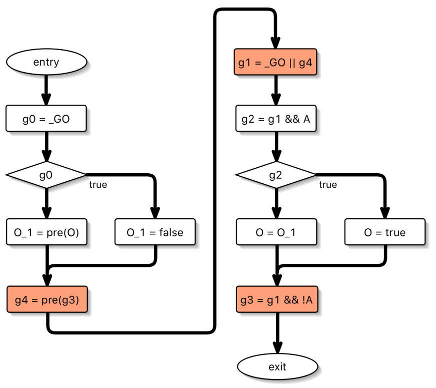
Fig. 12. Tracing of M2M transformations: Selecting the circuit block g2 in the circuit shows the origin in the intermediate and the source model.

4.3 Traceability

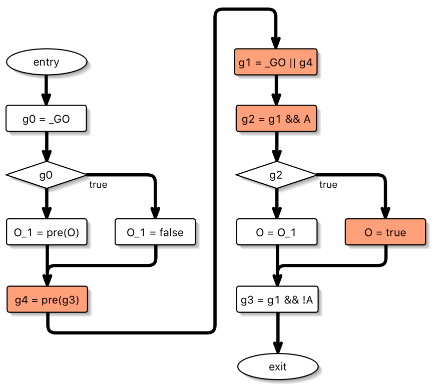
As depicted in the user story at the beginning of Sec. 4, the modeler may inspect the result of every transformation, including the final circuit. The elements of the graphical representation of a model are interactive. Each may be selected to trace its individual transformation history as explained in Sec. 3.1.

Fig. 12 illustrates a side-by-side view of selected transformation steps. The modeler may select an arbitrary number of transformations. In the figure, the source model, the SSA SCG, and the final circuit are selected. By selecting the g2 block in the circuit, the modeler sees the origin and transformation history of the g2 block. In this case, the block was created because of the assignment to g2 in the SSA SCG. The guard g2 guards the basic block (cf. Fig. 2c) and hence is indirectly created from the elements which are guarded by g2, namely state _S, the outgoing transition of _S and the assignment O = true.

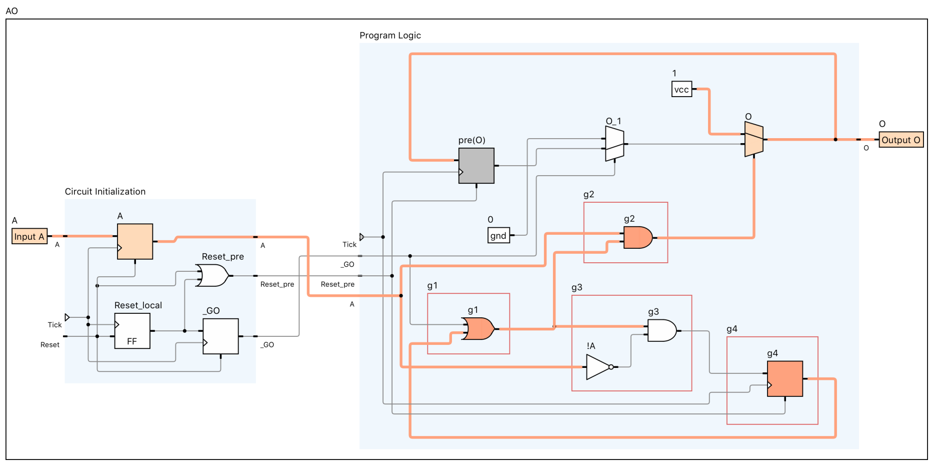
As can be seen in the middle part of Fig. 12, g2 determines whether or not O is set to true, which is the case when the outgoing transformation from _S is taken (cf. Fig. 12). Otherwise, O is set to O_1, the previous instance of O, meaning the state of O stays unchanged. Inspecting the circuit in Fig. 12 reveals that guard g2 controls the MUX O. Therefore, the aforementioned selection of the O instance is directly visible in the circuit.



(a) Simulation step of AO with input A set to false



(b) Simulation step of AO with input A set to true



(c) Simulation step of the AO circuit with A set to true: All active elements, analogous to the SSA SCG, are highlighted in orange.

Fig. 13. Simulation visualization of AO

4.4 Simulation

Step (7) in Fig. 8 marks the simulation component of the synthesized circuits. Following the simulation approach depicted in Sec. 3.2, C code is generated from the SSA SCGs. However, as explained before, the simulation is not restricted to C code. Every system that feeds back runtime information about the running program can be used. Since the circuit synthesis translates assignment nodes with guards and expressions as described in Sec. 4.2, the highlighting information gained from the generated C code can be used for the highlighting of circuit components. In each tick, the C code delivers runtime informations of active

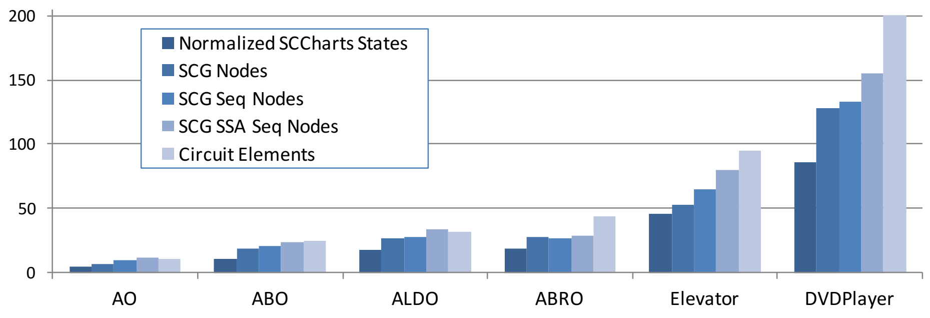


Fig. 14. Scaling of synthesized circuits compared to the corresponding SSA SCG and SCCharts depending on the number of nodes

guards and, hence, their basic blocks. The elements in the SSA SCGs and the circuits are highlighted according to this information. This corresponds to the end-to-end mapping of the tracing tree depicted in Fig. 7b in Sec. 3.1.

Fig. 13 shows parts of the simulation of the AO example program. The highlighted nodes in the depicted SCG mark the active guards in two consecutive ticks in Fig. 13a and Fig. 13b. An assignment with orange colored background indicates that this guard is active. Fig. 13a shows a non-initial tick in which A is set to false. Therefore, the final state has not been reached yet which is indicated by the highlighting of g3. Furthermore, the highlighting of g4 shows that this tick is non-initial because g4 stores the value of g3 from the last tick. Node g1 is active in every tick. This node describes the guard for the basic block which contains the evaluation of input A. In Fig. 13b, the subsequent tick, A is set to true. Hence, the assignment g2 = g1 && A now evaluates to true. The highlighting of assignment O = true shows that the program reacts as desired.

The simulation of the second tick is also depicted in the circuit in Fig. 13c. All live wires and components are highlighted orange. As described before, g2 is the composition of g1 and A. g1 is either true at the beginning of the program, indicated by the _GO signal, or if register g4 is active. As A is set to true in this tick, guard g2 also becomes true and sets the selection input of MUX O which applies the voltage to the output. Hence, O is set to true.

5 Evaluation

Fig. 14 shows the size in terms of the number of model elements of each intermediate result between the normalized source mode and the resulting circuit. The number of nodes of the normalized SCCharts reach from 5 for the AO model to 86 for a slightly bigger model, the DVDPlayer. Because the SSA transformation only adds assignments in the *else branches* of the conditional nodes, the number of nodes between sequentialized SCGs and SSA SCGs stay almost the same. As for the circuits, it is observable that in no case the number of nodes exceed twice the number of nodes in the SSA SCGs (omitting multiple occurrences of vcc and gnd). The AO circuit has 11 nodes and the DVDPlayer has 200 nodes. The number of nodes in the circuit for ABRO, the "hello world" example discussed in the case study [12], does not exceed 50. The complete ABRO circuit can be inspected in the thesis regarding interactive incremental hardware synthesis [16].

There are three different aspects which influence the scaling of circuits depending on the nodes in sequentialized SCGs:

1. Expressions like gX = gY are simply translated as one wire with two different names and therefore do not increase the number of nodes in circuits.
2. Guard expressions like gX = gY || gZ produce as many logic gates as nested operator expressions exist. Notice that the pre operator results in the creation of a register that stores the value of the previous reaction. Depending on the number of concurrent regions in the SCChart, a *complex guard* [18], which is used for joining threads, may be more complex. For each pause statement per thread, another logic gate is required.
3. New assignment nodes on *else branches* in the SSA SCGs (cf. Sec. 4.1) do not increase the number of logic gates. The conditional still only needs one MUX as each MUX summarizes the assignment nodes from each conditional branch. This is the reason why, e. g., the ALDO circuit has fewer nodes than its sequentialized SCG.

6 Related Work

The close relation between compilation and modeling techniques has been observed quite early, e. g., by Steffen, who proposes to make use of *consistency models* to detect inconsistencies between different model descriptions, and relates this to giving a semantics to a programming language by translation into an intermediate language [19]. Since then, a number of modeling approaches have been developed that also address model compilation. For example, Cinco can automatically construct code generators from a given meta model [13]. Grundy et al. give a good overview of the current state and present Marama, which provides a set of mostly visual metatools for language specification and tool building, including synthesis [7]. One difference of these approaches and our proposal here is that we aim to 1) divide the synthesis of the human-authored artifact into the low-level result into rather small, in themselves conceptually simple steps, applied in a single, sequential pass, and to 2) make the intermediate transformation results accessible to the user, by automatically deriving well-readable graphical views of the model stages. Moreover, the separation of model and graphical view starts at the very beginning, as the human works on a textual model description, using all efficiency advantages of a textual editor and frameworks such as Xtext. This is in line with *pragmatic modeling*, which aims to free the user of tedious layout tasks involving a palette and manual place and route [6]. Lopes [10] studies the general specification of mappings and inspired the tracing SLIC extension. However, Lopes's approach considers the mapping specification between two meta models, where we also consider mappings between models of the same meta model.

Concerning our case study, the synthesis of hardware from SCCharts, the hardware synthesis from Statecharts [8] introduced by Drusinsky and Harel [4] uses Statecharts as behavioral HDL. The idea is to use single machines implementing finite state machines (FSMs). Since Statecharts allow non-deterministic behavior this approach is not taken into consideration for the hardware synthesis from SCCharts. Esterel [3] is a synchronous language tailored for the development of embedded reactive applications in hardware and software. Esterel programs can directly be translated into circuits [2]. Since SCCharts' SC MoC is a conservative extension of the classical synchronous MoC [21], the ideas of concurrent regions and the usage of registers to store the system state is adapted from Esterel's hardware synthesis. However, Esterel's hardware synthesis approach is a bit more involved than the synthesis from the SCG we propose here in that preemptions have already been transformed away when the SCG level is reached. Sequentially Constructive Esterel (SCEst), studied by Rathlev et al. [14], can be used to generate hardware circuits with our approach because the hardware synthesis can be directly applied to the SCG. Hence, we also provide a new route for circuit creation for Esterel. Johannsen [9] studied hardware synthesis for SCCharts before and translated SCCharts to VHDL. The ISE tool[2] then visualizes and simulates the circuit. This approach can also be pursued by other tools that are capable of describing FSMs. However, the interactive and incremental approach proposed here has no breaks in the toolchain and is integrated into the KIELER framework and thus uses KIELER layout and visualization. Therefore, no external tool is necessary. The traceability of the circuits behavior is supported since all intermediate transformations are visible. Additionally, SCCharts models can be compiled to software. Nevertheless, a comparison of the synthesis quality of classical hardware design tools with the approach presented here would be interesting future work.

Edwards [5] provides an overview of different approaches for hardware synthesis from C like languages and their limitations. Since SCCharts can be compiled to C or Java code, it is also possible to pursue these routes to generate hardware circuits. This would be particularly useful for *hostcode calls*, which are currently not included in our compiler.

7 Conclusions

The incremental interactive hardware synthesis integrates into the SCCharts SLIC approach. By adding two M2M transformations to the compiler chain, a modeler is able to generate hardware circuits for SCCharts models conveniently. Each transformation step, including the final circuit, can be simulated within the toolchain. During simulation, runtime information is visualized in all selected intermediate transformation results. Additionally, each model element can be

[2] http://www.xilinx.com

traced back to the source model; there are no breaks in the toolchain. Summarized, the convenient creation of source models, automatic generation of hardware circuits, and fully integrated simulation and traceability of model elements are powerful tools for developing integrated circuits. The interactivity between these key component is crucial.

As mentioned in Sec. 3.1, our tracing framework only considers actual model instances. In the future we are going to extend the framework, so that it can handle textual results as well. Additionally, a dedicated simulation interpreter, as proposed in Sec. 3.2, could exemplify the advantages of the combination of SLIC and a tracing framework even further. Concerning synthesis, we plan to reintroduce support for VHDL and hostcode calls. Simulation and visualization of dedicated tools, such as the ISE tool, could be compared to the KIELER results and hence represent a new resource for validation.

References

1. C. André. Semantics of SyncCharts. Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.
2. G. Berry. Esterel on Hardware. *Philosophical Transactions of the Royal Society of London*, 339:87–104, 1992.
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. D. Drusinsky and D. Harel. Using Statecharts for hardware description and synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(7):798–807, July 1989.
5. S. A. Edwards. The challenges of hardware synthesis from c-like languages. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, pages 66–67, Washington, DC, USA, 2005. IEEE Computer Society.
6. H. Fuhrmann and R. von Hanxleden. On the pragmatics of model-based design. In *Foundations of Computer Software. Future Trends and Techniques for Development—15th Monterey Workshop 2008, Budapest, Hungary, September 24–26, 2008, Revised Selected Papers*, volume 6028 of *LNCS*, pages 116–140, 2010.
7. J. C. Grundy, J. Hosking, K. N. Li, N. M. Ali, J. Huh, and R. L. Li. Generating Domain-Specific Visual Language Tools from Abstract Visual Specifications. *IEEE Transactions on Software Engineering*, 39(4):487–515, Apr. 2013.
8. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
9. G. Johannsen. Hardwaresynthese aus SCCharts. Master thesis, Kiel University, Department of Computer Science, Oct. 2013. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/gjo-mt.pdf.
10. D. Lopes, S. Hammoudi, J. Bézivin, and F. Jouault. Mapping specification in mda: From theory to practice. In *In: Konstantas, D., Bourrières, J.-P., Léonard, M., Boudjlida, N. (Eds). Interoperability of Enterprise Software and Applications - INTEROP-ESA*, pages 253–264. Springer, 2006.
11. C. Motika, H. Fuhrmann, and R. von Hanxleden. Semantics and execution of domain specific models. In *2nd Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2010) INFORMATIK 2010*, GI-Edition – Lecture

Notes in Informatics (LNI), pages 891–896, Leipzig, Germany, Sept. 2010. Bonner Köllen Verlag.

12. 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.
13. S. Naujokat, L.-M. Traonouez, M. Isberner, B. Steffen, and A. Legay. *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, chapter Domain-Specific Code Generator Modeling: A Case Study for Multi-faceted Concurrent Systems, pages 481–498. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.
14. K. Rathlev, S. Smyth, C. Motika, R. von Hanxleden, and M. Mendler. SCEst: Sequentially Constructive Esterel. In *Proceedings of the 13th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE'15)*, Austin, TX, USA, Sept. 2015.
15. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM.
16. F. Rybicki. Interactive incremental hardware synthesis for SCCharts. Bachelor thesis, Kiel University, Department of Computer Science, Mar. 2016. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/fry-bt.pdf.
17. 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 Sept. 2013.
18. S. Smyth, C. Motika, and R. von Hanxleden. A data-flow approach for compiling the sequentially constructive language (SCL). In *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2015)*, Pörtschach, Austria, 5-7 Oct. 2015.
19. B. Steffen. Unifying models. In *STACS 97, 14th Annual Symposium on Theoretical Aspects of Computer Science, Lübeck, Germany*, pages 1–20, Mar. 1997.
20. 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, Dec. 2013. ISSN 2192-6247.
21. 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.
22. 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.