Electronic Communications of the EASST
Volume 078 (2019)

8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation

-

Doctoral Symposium and Industry Day, 2018

Guidance in Model-based Compilations

Steven Smyth
and Alexander Schulz-Rosengarten
advised by Reinhard von Hanxleden

xvii pages

Guest Editors: Falk Howar, Anna-Lena Lamprecht
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Guidance in Model-based Compilations

Steven Smyth
and Alexander Schulz-Rosengarten
advised by Reinhard von Hanxleden

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

Abstract: Model-driven development promises to ease the overall development process of complex systems. Models provide abstract problem solutions often without the need to tackle concrete technical details. However, as complexity and safety requirements of modern systems grow, transforming text-based to graphical-based programs does not suffice any more to create a reasonable overview. The modeler should not be burdened with maintaining an overview over all potential conflicts within a particular model of computation or between components of a system. They should further be able to understand what is happening during transformation steps. Modern modeling tools and model-based compilers should not only be sophisticated graphical editors that produce code on demand. They should guide modelers during the modeling process so that they can refine their models interactively.

The KIELER Compiler constructs transformation snapshots and augmented models automatically during compilation. We demonstrate six different exemplary transient views that can help the modeler to refine their models and to solve modeling issues, such as causality problems in synchronous languages. While the compiler is agnostic towards the meta-models of the source and target languages, the synchronous language SCCharts serves as main example.

Keywords: Interactive Compilation, Model-based Design, Model-driven Development

1 Introduction

As part of the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[1] modeling tools, we developed the model-based KIELER Compiler (KiCo) to demonstrate the capabilities of our synchronous language SCCharts and transient view framework KLighD [23]. SCCharts, first presented in 2014 [32], are a dialect of statecharts [14] with a synchronous semantics. The used sequentially constructive semantics [33] reconciles deterministic concurrency with an imperative, sequential programming style. Even though SCCharts were the initial motivation for KiCo, it is a modular framework that is not tailored to a specific language. Hence, KiCo can be used to compile (or process) anything. Every intermediate result is a fully functional artefact that can be

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

inspected via state-of-the-art pragmatic modeling techniques and the compilation context itself may change during its life cycle [30]. As an example, Fig. 1 shows the compilation chain of the netlist-based C compilation of SCCharts within the KIELER modeling tools. The compilation system references two other systems, SCCharts Normalization and SCG Netlist-based, with the latter being expanded to explore the included steps. Each contained blue rectangle is an intermediate and fully functional result that can be inspected. Additional color coding can help to identify problems. KIELER uses the common set of colors, white for infos, yellow for warnings, and red for errors.

Figure 1: Example of a KiCo compilation chain: The Netlist-based compilation. Each contained blue rectangle is an intermediate and fully functional result that can be inspected.

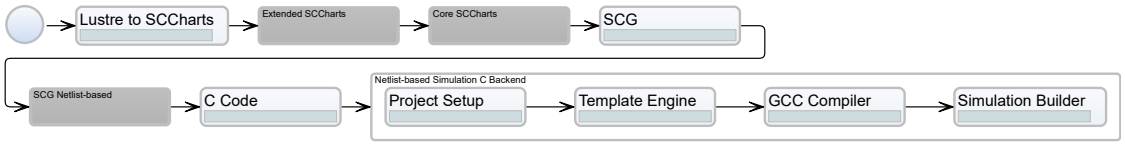


Figure 2: Extended compilation chain that compiles Lustre programs to SCCharts and then uses the pre-exisiting SCCharts compilation to compile to C. Afterwards, the simulation backend prepares an appropriate simulation.

While SCCharts serves as main example throughout this paper, several other source or target languages have been implemented into KIELER. The framework initially supports other synchronous languages, such as Esterel [20,25] and Lustre [12], as source. Experimentally, standard C code can be compiled to high-level model descriptions [16]. Besides standard compilation targets, such as C and Java, KIELER supports direct code generation for hardware circuits [22, 29] or different dialects, such as C for Arduino [17]. Since KiCo is a modular compiler, further languages can be added to the compiler as sources or targets easily. If desired, pre-existing compilation chains can then be used to proceed with the compilation. For example, the compilation chain in Fig. 2 adds a Lustre processor to the already existing SCCharts compilation chain. This chain can be used to compile Lustre programs to SCChart and then uses the pre-existing chain to compile to C code. This addition can be made on the fly without restarting the IDE, because compilation chains in KiCo are also models, which can be created and altered during run-time. However, in practice it is often preferable to develop compiler components in a separate IDE with defined target platform specifications and deploy KIELER releases. New compiler additions are then also available in a lean command-line tool variant of KIELER.

Although KIELER is a Java project, developers do not have to implement all reasonable behavior again in Java. For example, KiCo processors can also invoke external tools, such as calling the GCC. The chain in Fig. 2 does not end on the C code generation. Once the C code has been

generated, it is send to the GCC[2], which creates a binary. Subsequently, the simulator integrated in KIELER loads the binary and simulates the program within the IDE. Further post-processing, such as deployment to specific hardware, is also possible. More details on the possibilities of the model-based compiler can be found elsewhere [31].

The core of a model-based compiler are its model-to-model transformations [18]. All models generated by these transformations can be inspected at run-time and saved. Issues, such as race conditions in classical programming or causality problems in synchronous languages are sometimes hard to spot. Therefore, we explored appropriate processing and representations/views of the available data. We argue that modern development tools, such as model-based compilers, besides producing the correct result, can and should also guide the modeler to potential issues and provide means to understand what is happening (during its transformations).

Contribution & Outline: After covering related work in Sec. 2, (1) we divide intermediate model-based compilation results, also called *snapshots*, into two categories, named *simple* and *augmented* snapshots in Sec. 3. With a focus on augmented snapshots in Sec. 3.2, (2) we give six examples of enriched model feedback to illustrate the possibilities of instantaneous interactive feedback while modeling. Additionally, (3) we show preliminary results gathered during the development of the model-based SCCharts compiler in Sec. 4 and conclude in Sec. 5.

2 Related Work

If a program is rejected by a compiler, it is important to guide the user towards the problem. Graphical languages have the advantage of intuitive visual problem reporting. However, regarding synchronous languages, such as SyncCharts [3] and SCADE [10], this potential is often only used for simulation. For SCCharts as a descendant of SyncCharts, KIELER relies on a visual representation close to the original Statecharts introduced by Harel [14]. This graphical representation has been proven to be intuitively understandable and is also the basis of the UML Statecharts dialect [9].

```
1 x = 3
2 if (x > 0) {
3     x = x * 7;
4 }
5 y = x * 2;
```

Listing 1: A small C program

```
1 x_3 = 3;
2 if (x_3 > 0)
3 goto <bb 3>; [0.00%]
4 else
5 goto <bb 4>; [0.00%]
6
7 <bb 3> [0.00%]:
8 x_4 = x_3 * 7;
9
10 <bb 4> [0.00%]:
11 # x_1 = PHI <x_3(2), x_4(3)>
12 y_5 = x_1 * 2;
```

Listing 2: Code snippet of the Single Static Assignment (SSA) intermediate representation in GNU Compiler Collection (GCC) generated with the `-fdump-tree-ssa` option. (Example taken from TR-1806 [31].)

For general purpose compilation, compilers also often allow to access intermediate representations. For example, the GCC includes intermediate representation for basic blocks [2] and optimizations in textual form. However, the accessibility and understandability is tailored to the needs of a compiler expert. List. 2 shows an extract of the SSA representation of the C program in List. 1. It illustrates the basic block separation (lines 7 and 10), renaming of variables (lines 1, 2, 8, 11 and 12), and placement of a Φ-function (line 11) in the partially translated code. While all

[2] https://gcc.gnu.org

information is present, without additional affiliated tools that further process these intermediate results, the representation is arguably rarely helpful for the modeler.

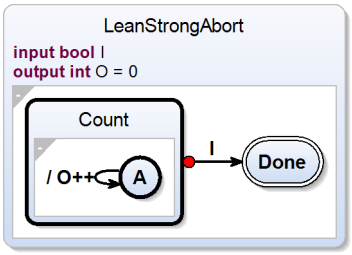
Over the years, a number of modeling compilation approaches have been developed, such as CINCO [19], a meta-level modeling tool generator, and MARAMA [13], which provides metatools for language specification and tool creation. While these tools provide sophisticated means to work on the artefacts in question, once a modeling step is done and the model is compiled, there is little information or interactivity that guides the developer on what happened. Same is true for other Statecharts modeling tools, such as Rhapsody and Simulink Stateflow. However, we currently investigate object-oriented modeling techniques [24] that can be found in, e. g., Rhapsody, and how to guide modelers in these contexts.

In our approach we provide the modeler with generic, interactive tools to orchestrate compilation processes. These are divided into atomic steps that aid the modeler to refine the process and to find errors without the need for long development cycles. The source, intermediate, target, and additional models are presented in well-readable graphical views using transient view and automatic layout technologies [23]. To guide the modeler, one goal is to provide meaningful model representations in the domain of the modeler. Hence, contrary to popular compiler infrastructures, such as LLVM [15], which try to reach a common intermediate language as soon as possible to maximize modularity, KiCo tries to stay in a domain meta-model as long as reasonable to facilitate understandability.

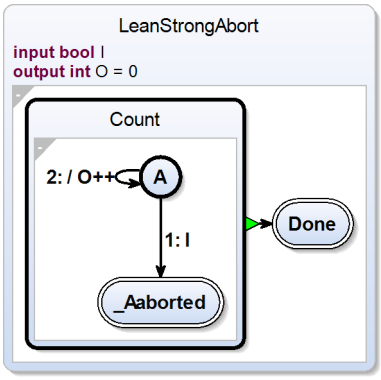
The process systems of the KiCo can be seen as a variant of *scientific workflows* [7] for model-to-model transformations combined with state-of-the-art pragmatic modeling techniques. There are similarities in the orchestration, presentation, and the overall life cycle of these workflows, even though KIELER is specialized in model-to-model transformations and interactivity. Equal to the experiences gathered during the development of KIELER, Curcin et al. [7] observed an iterative refinement process during the life cycle of a workflow. Developers review individual fragments of the workflows to modify the functionality or to improve the performance. After the development completes, the workflow is deployed for execution.

3 Interactive Guidance

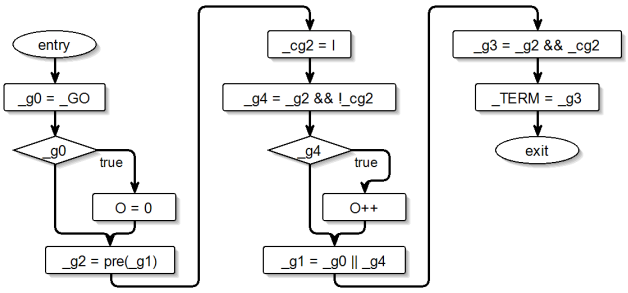
KiCo uses interactive process systems to perform compilations [30]. They can be used to generate transformation *snapshots* easily and present them with or without additional information to the modeler. A *simple snapshot* represents the state of the transformation chain at that particular moment, whereas an *augmented snapshot* is enriched with additional information, such as model element annotations or mappings to the original model, which basically come for free in the KiCo framework. Each of the colored rectangles in the compilation chain example in Fig. 1 is an inspectable intermediate model. We explored different ways of presenting the data that is gathered. In Sec. 3.1 we give an example for simple snapshots and then, focus on different guidance possibilities with augmented snapshots in Sec. 3.2. Both variants can be shown interactively by selecting the desired interactive result or as dedicated view.



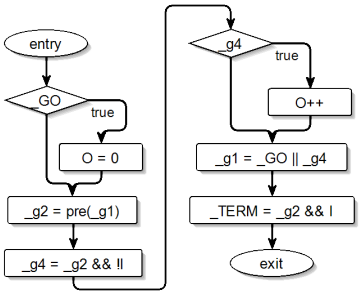
(a) Original model



(b) Model after the abort transformation



(c) Model in sequential controlflow form



(d) Optimized controlflow form

Figure 3: Example of different unmodified transformation snapshots: LeanStrongAbort

3.1 Simple Snapshots

While transformation steps are mandatory for the desired process system to fulfill its task, they can also guide the user without further annotations to understand a transformation stepwise. These steps can be shown as simple snapshots on arbitrary granularity. Fig. 3 shows four different steps of the KIELER netlist-based compilation, which overall consists of over 30 transformations. One can see the original SCCharts model in Fig. 3a. It serves as source for the compilation. Without going into too much details of the SCCharts language, after an initialization in the first cycle, the program increases an integer output O in every clock cycle. As soon as an input I is present, the counting stops immediately without increasing the counter in this cycle due to a so called *strong abort* (red circle transition) and the program is terminated.

Since strong aborts are an extended language feature, the feature has to be transformed into more simpler language constructs. Fig. 3b shows the semantically equivalent model after the application of the abort transformation. The strong abort transition is now resolved into a normal termination transition (green triangle transition), which triggers if the control of its source state reaches a final state (double border). Transition priorities (with lower integers having higher priorities) now manage the counter. If I becomes true, the control switches to the _Aborted state. Otherwise, O is incremented.

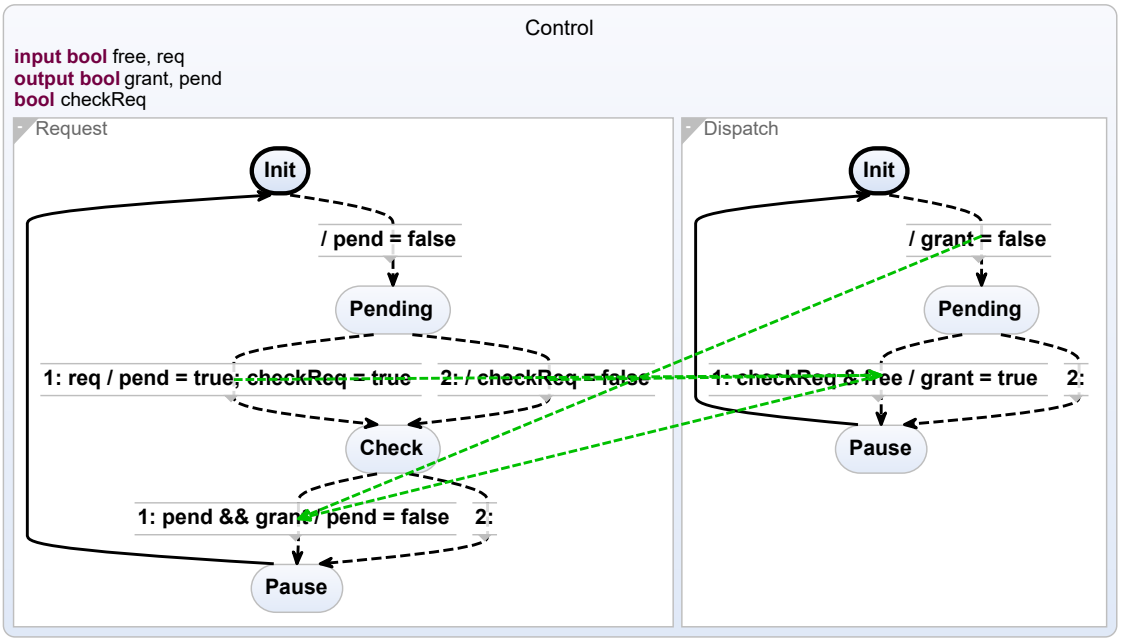


Figure 4: Data dependencies

Eventually, the program is sequentialized into a controlflow graph that only contains *assignments* (rectangles) and *conditionals* (diamonds) in the netlist-based compilation as can be seen in Fig. 3c. The whole netlist logic is executed in every cycle, with the _GO variable signaling the (re-)start of the program. _TERM is set to true if the program terminates. Values from previous cycles can be obtained with a *pre* operator. The sequentialized code can further be optimized. Fig. 3d shows the results of the copy propagation [1]. Even if not familiar with the netlist-based compilation approach, the behaviour of the program is observable. O is set to 0 in the beginning. It is increased in subsequent cycles managed by the *guard* _g2 and if the input I is false. As soon as I becomes true, the program terminates without counting O.

3.2 Augmented Snapshots

We demonstrate six different views that may guide the developer in the following subsections. Other possibilities of data processing and visualization are imaginable, especially when tailored to specific use-cases. There is no hard limit for the compiler framework. However, the usefulness of the processed data is tightly connected to its presentation. The guidance may be more effective if supported by transient view syntheses [23]. Nonetheless, even plain text or unmodified artefacts may help the modeler as long as they are interactively accessible and easy to understand.

All figures show different variations of the data gathered from the compilation of the same model with slight modifications w.l.o.g. to show the potential of the different views. The first three views, described in Sec. 3.2.1 – 3.2.3, follow a more pro-active approach in notifying the modeler about potential conflicts, also called *causality guidance*. The last three views, explained

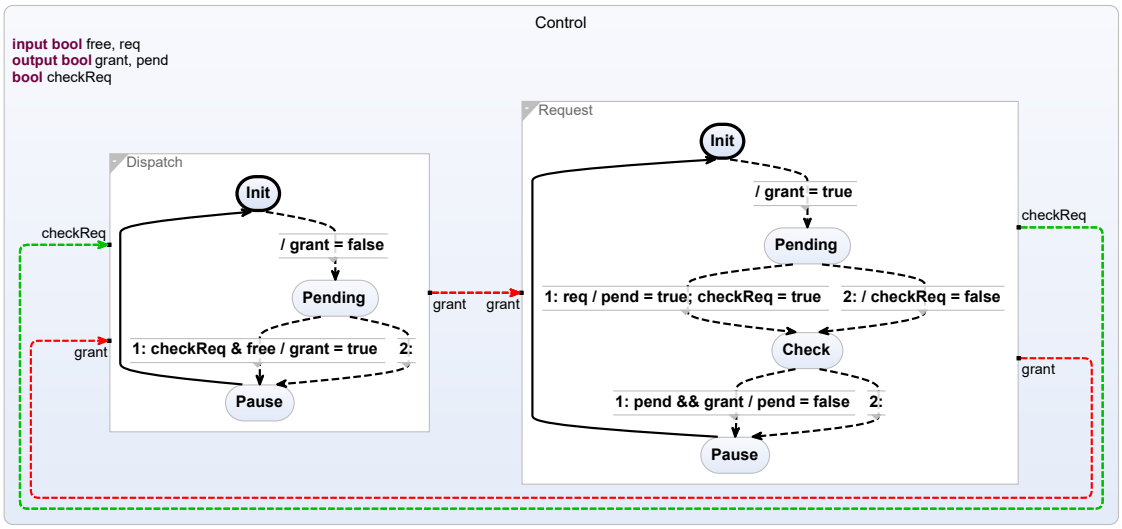


Figure 5: Causal dataflow

in Sec. 3.2.4 – 3.2.6, help the modeler to understand what is happening during the transformations, which we call *transformation observation*. Unmodified Snapshots also fall in this category.

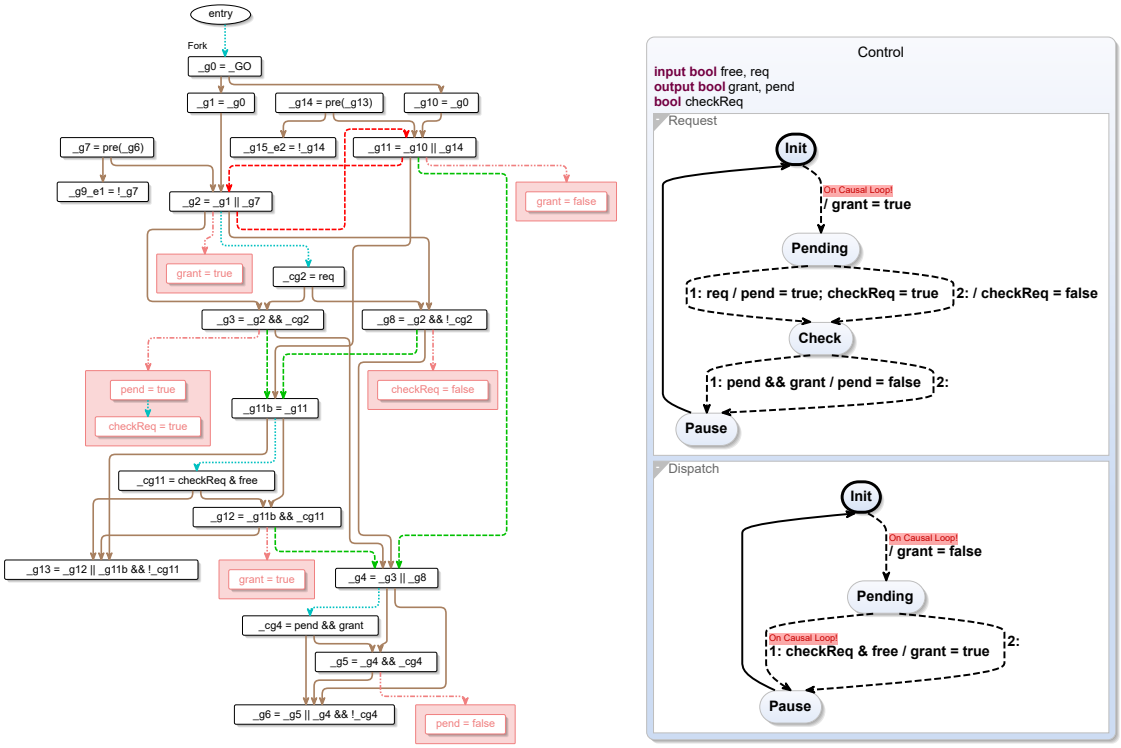
3.2.1 Data Dependencies

Data dependencies usually govern the scheduling order of the program. Fig. 4 shows an example SCChart with two concurrent regions. Dependencies between concurrent variable accesses can be visualized directly in the diagram of the model, instead of showing a corresponding control or dataflow flow graph. The data dependencies here are depicted as green dashed edges. They can also be visualized on different granularity levels within the model or only if specific states are selected by the user if desired. It is also possible to display all, i. e. non-concurrent, variable dependencies and mark ineffective accesses, which can help to refine the model further.

3.2.2 Causal & Induced Dataflow

The induced dataflow view [34] shows communication between concurrent regions. It visualizes the dataflow of the program even if the underlying model uses the control flow paradigm, such as, e. g., SCCharts. A variant thereof, the causality dataflow view, focusses on identifying data dependency cycles. If the source model is changed, such that conflicting values are written to the variable grant, the dependency cycle is depicted in red, as can be seen in Fig. 5. The modeler is now informed that a dependency cycle between concurrent regions is present and that the model is not constructive in the sense of the underlying Model of Computation (MoC). It becomes clear that the compiler is going to reject the program without the need to actually run the compilation chain.

3.2.3 Scheduling Propagation



(a) Sequentially constructive program dependency graph

(b) Propagated scheduling data

Figure 6: Scheduling propagation

Nonetheless, if desired or if constructivity is still not determined, programs can be compiled. Information gathered during compilation can be propagated back to the original model to provide reasonable feedback. On this level, complex scheduling relationships are simply displayed as annotations [30]. The problematic cyclic variable access, which was introduced in Sec. 3.2.2, is shown to the modeler within the model.

The compiler (or expert thereof) can detect the dependency cycle in the program dependency graph of the program, which is depicted in Fig. 6a. The complete scheduling information, which is needed during compilation anyway, is accessible interactively in a readable way. It can be used by experts to solve complex scheduling issues. The dependency cycle is depicted as cyclic dashed edge in the example. However, the modeler is informed via propagation in the original model on the right side in Fig. 6b. Once notified, the issue can be fixed easily without the need to dive deep into the compilation chain.

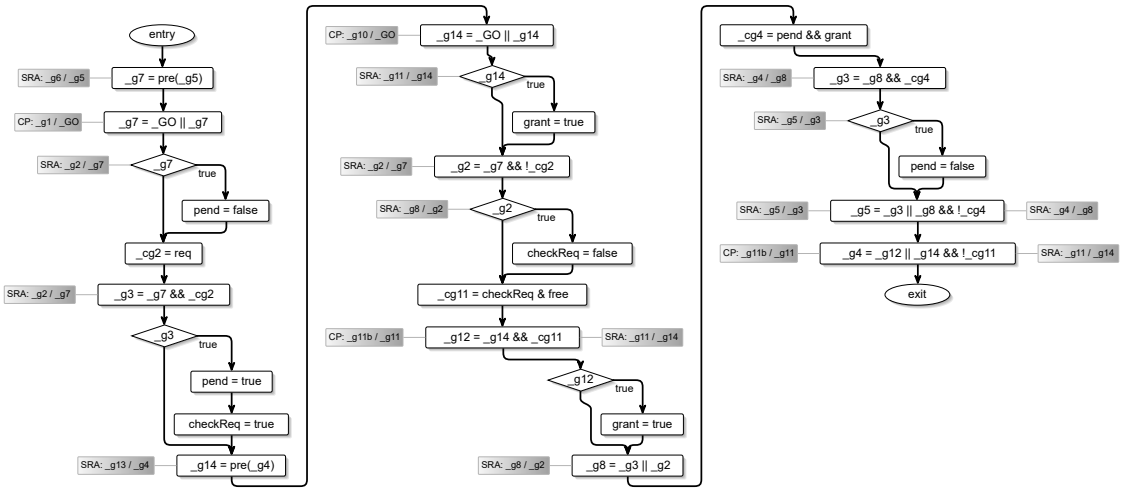


Figure 7: Transformation snapshots

3.2.4 Transformation Snapshots

Every intermediate step of the transformation chain can be preserved as intermediate snapshot model by the transformations and enriched with individual annotations [18, 30]. In the example shown in Fig. 7, different classical optimizations, such as copy propagation (CP) [1] and smart register allocation (SRA) [6], annotate which nodes were modified by their processes. The framework handles the mapping between model elements automatically. Hence, it is not necessary for the compiler developer to keep track of these changes manually. They can simply add annotations to specific model elements, e. g., graph nodes in the example, programmatically in their optimization. The annotations will be attached automatically to the appropriate visualization of the selected nodes.

3.2.5 Automatic Element Tracing

The compiler framework keeps track of the transitive model element relations [22]. Hence, the relation between the elements of arbitrary intermediate models of the compilation chain can be made visible. In the example shown in Fig. 8, the nodes of the final model of the netlist-based compilation [18] are mapped back to the original model.

Displaying all relationships at once can be confusing. Hence, the user can select points of interest by clicking on the elements. Fig. 9 shows a smaller program with a transitive element tracing over three transformation steps. Here, the final model represents a hardware circuit. The paths that lead to the creation of single gates can be inspected easily.

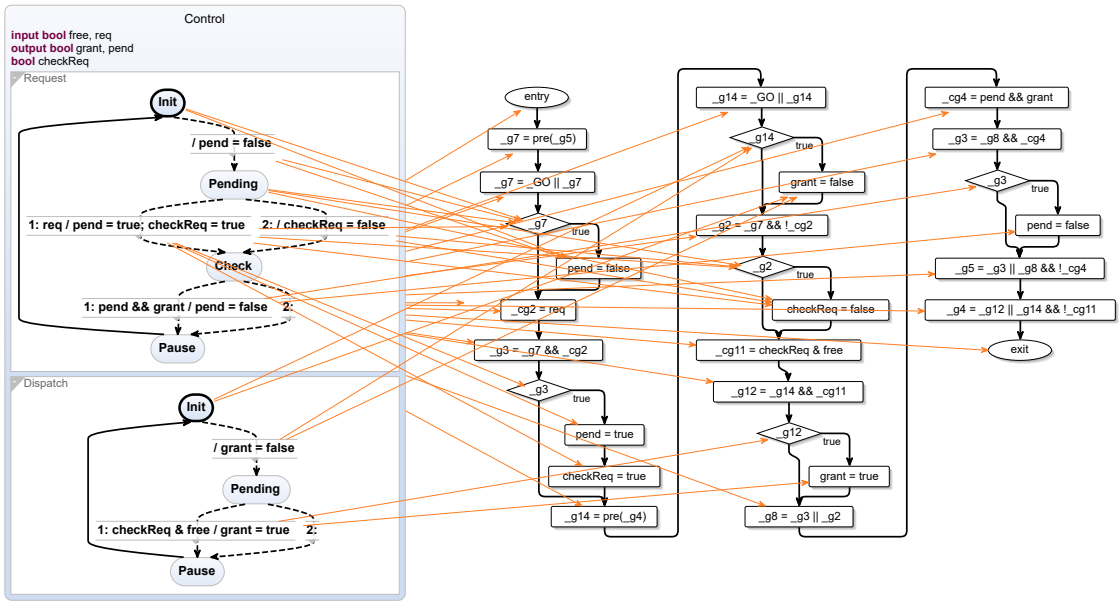


Figure 8: Automatic element tracing

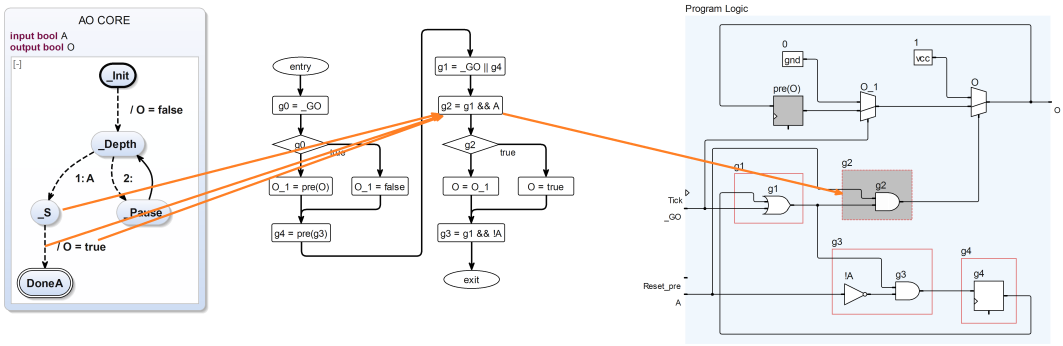


Figure 9: Automatic element tracing of selected elements

3.2.6 Built-in Code Mapping

Generated target code can be annotated directly in the original model if supported by the chosen code generator. For example (see Fig. 10), SCCharts' state-based code generation creates one function for each state besides other functions. These final code fragments can be shown directly in the original model. As shown in the figure, the generated source codes of each state function are attached as comment nodes to the state in the original model.

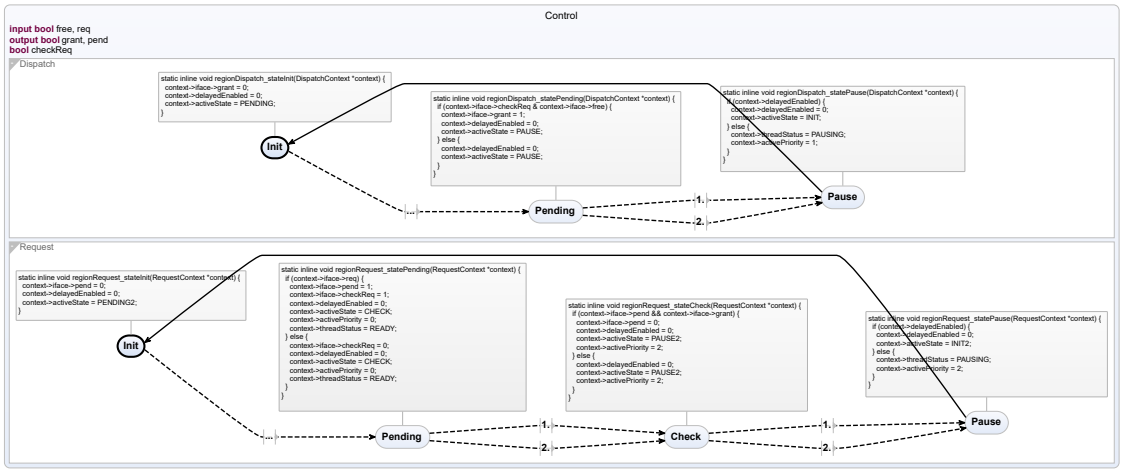


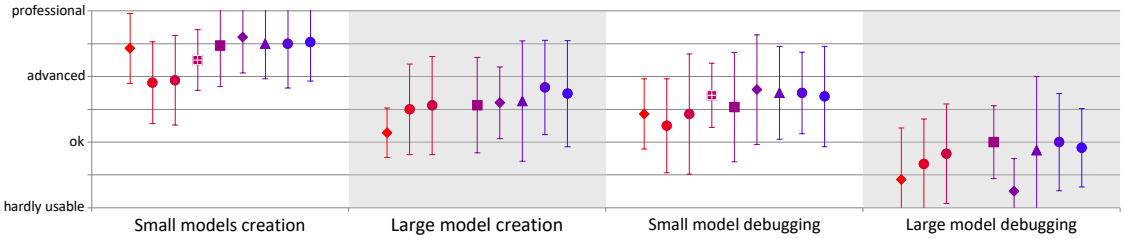
Figure 10: Built-in code mapping

4 Preliminary Results

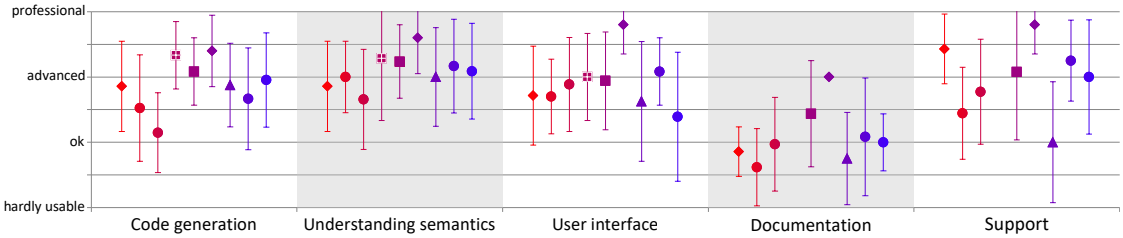
Since the development of SCCharts started in 2014, the language and its KIELER implementation have been used in teaching and in recent years also in industrial context. We handed out extensive questionnaires on the language and tooling during this time. First results of the surveys are summarized in the *SCCharts: The Railway Report* published in 2015 [27]. It shows the ratings of the participants of the railway project at our department at the Kiel University. Goal of the project was to use the model-based approach to model a railway controller that controls eleven trains concurrently on a model railway demonstrator. The report showed that SCCharts and the tooling can compete with other predominant languages, such as C or Java, or the synchronous language Esterel [4], in this context. It showed also that there is still room for improvements, especially when it comes to debugging and curing causality issues. Since then, 127 students, including participants from external departments, and 8 professionals at the Synchron Workshop in Bamberg 2016 returned a survey. Their feedback helped us to improve our modeling tools steadily. The final summary including all survey results is published as technical report [26].

An excerpt from the results can be seen in Fig. 11. The ratings show distinct groups of participants in chronological order from left to right in each category. The survey's symbol indicates the kind of the associated project. A diamond ◆ marks a railway project, which we consider a medium-size project for students spanning a whole semester. A circle ● indicates a Mindstorms project with several smaller tasks usually solvable in one to two weeks. A square ■ stands for a synchonous lecture at our department, a triangle ▲ marks an external project, and a crossed square ▣ marks the survey conducted during the Synchron Workshop in 2016. The first group represent the ratings of the railway participants from 2014. The crossed square results are the ratings of the workshop professionals. As they only got a short version of the survey, they only have results in some categories.

Fig. 11a shows the rating results of the model creation and debugging capabilities. The discrepancy between small and large models was rated more extreme in the larger railway projects.



(a) Model creation and debugging



(b) Tooling and support

Figure 11: Survey ratings (from TR-1904 [26])

Naturally, what was seen as a large model, differs within the project groups. However, all participants agreed that debugging, particularly of large models, is hard and remains the Achilles' heel of SCCharts and perhaps model-based development in general. However, the relative trends indicate that the tooling improved over time and we plan to focus even more on usability, maintainability, and debugging features.

Fig. 11b shows several aspects of the KIELER tooling. The *code generation* was rated worse in the first two Mindstorms projects, which can be attributed to the resource limitations of the used Real-Time Operating System (RTOS). In later versions, several optimizations [5, 31] were added to make larger models executable on RTOSs with limited resources. All ratings of the *understanding semantics* question are above average. The tooling provides clear representations of the features in use and illustrates the processes involved during model creation and compilation in a comprehensible way. The rating of the *user interface* nearly stayed the same in all projects. Although *documentation* improved over time, this is still another weak spot of the SCCharts project. Despite the fact that the documentation and examples got expanded before the last iteration of the embedded systems class, the latest improvements do not seem to have a big impact on the ratings and are not enough to give a better than ok impression. Better ways of presenting the actual state of the project should be explored and implemented in the future. Naturally, the in-house projects scored better in the *support* ratings.

The report [26] gives more details on different languages comparisons and discusses the tooling ratings in more detail.

5 Conclusion

SCCharts and the SCCharts tooling are governed by evolutionary processes. The language, the technology, and the ways to model efficiently in KIELER will change. However, regardless of technological changes, we believe that the concept of strong interactive developer guidance will prevail and help modelers to develop more efficiently. We humbly advise model-based tool developers to adhere to the principles presented earlier:

Guidance The modeler should not be burdened with maintaining an overview over all potential conflicts, but should be assisted with finding solutions to these. Modeling tools should provide transient views automatically while the modeler works on their model.

Observation The modeler should be able to understand what is happening during transformations/compilations. Intermediate results, automatic mappings and meaningful annotations facilitate understandability and ease manual verifications.

Both principles, regardless of the concrete views in use, help to refine the source model, which in turn creates more refined dedicated views. For SCCharts, we depicted this cycle in the accompanying poster, which can be seen in Appendix A.

As future work, we want to evaluate further by which means the tooling can be improved to help the modeler and whole developer teams even more. For example, a new code-generation approach for verification is currently under evaluation [28]. We also investigate new user front-end technologies, such as combined web/desktop application possible via frameworks such as electron[3]. A first front-end based on the Theia framework has already been developed [8, 21]. It supports model-based compilation via KiCo using the Language Server Protocol (LSP) and can be used as web and desktop application. Especially, debugging model-based programs is a greater concern as the survey evaluation showed. Based on work by Grimm towards debugging SCCharts [11], new possibilities to debug model-generated code during run-time execution are also under investigation.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[2] F. E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[3] C. André. SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR 96–56, I3S, Sophia-Antipolis, France, Rev. April 1996.

[4] G. Berry and L. Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, volume 197 of *LNCS*, pages 389–448. Springer-Verlag, 1984.

[5] J. Busse. SCCharts modeling for embedded systems with limited resources. Bachelor thesis, Kiel University, Department of Computer Science, Sept. 2016. https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jbus-bt.pdf.

[3] https://electronjs.org

[6] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Comput. Lang.*, 6(1):47–57, Jan. 1981.

[7] V. Curcin, M. Ghanem, and Y. Guo. The design and implementation of a workflow analysis tool. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 368(1926):4193–4208, 2010.

[8] S. Domrös. Moving model-driven engineering from Eclipse to web technologies. Master's thesis, Kiel University, Department of Computer Science, Nov. 2018. https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sdo-mt.pdf.

[9] B. P. Douglass. Uml statecharts. *Embedded Systems Programming*, pages 22–42, Jan. 1999.

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

[11] L. Grimm. Debugging SCCharts. Bachelor's thesis, Kiel University, Department of Computer Science, Sept. 2016. https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lgr-bt.pdf.

[12] L. Grimm. From Lustre to graphical dataflow programs. Master's thesis, Kiel University, Department of Computer Science, May 2019. https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lgr-mt.pdf.

[13] 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.

[14] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[15] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[16] S. Lenga. Model-based compilation of legacy C programs. Bachelor thesis, Kiel University, Department of Computer Science, Sept. 2016. https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sle-bt.pdf.

[17] C. Motika. *SCCharts—Language and Interactive Incremental Implementation*. Number 2017/2 in Kiel Computer Science Series. Department of Computer Science, 2017. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel.

[18] 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 461–480, Corfu, Greece, Oct. 2014.

[19] S. Naujokat, M. Lybecait, D. Kopetzki, and B. Steffen. Cinco: a simplicity-driven approach to full generation of domain-specific graphical modeling tools. *International Journal on Software Tools for Technology Transfer*, 20(3):327–354, Jun 2018.

[20] M. Rahimi-Barfeh. Incremental compilation of SCEst. Bachelor thesis, Kiel University, Department of Computer Science, Sept. 2017. https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mrb-bt.pdf.

[21] N. Rentz. Moving transient views from Eclipse to web technologies. Master's thesis, Kiel University, Department of Computer Science, Nov. 2018. https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nir-mt.pdf.

[22] F. Rybicki, S. Smyth, C. Motika, A. Schulz-Rosengarten, and R. von Hanxleden. Interactive model-based compilation continued – interactive incremental hardware synthesis for SCCharts. In *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)*, volume 9953 of *LNCS*, pages 150–170, Corfu, Greece, Oct. 2016.

[23] C. Schneider, M. Spönemann, and R. von Hanxleden. Just model! – Putting automatic synthesis of node-link-diagrams into practice. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*, pages 75–82, San Jose, CA, USA, Sept. 2013.

[24] A. Schulz-Rosengarten, S. Smyth, and M. Mendler. Towards object-oriented modeling in SCCharts. In *Proc. Forum on Specification and Design Languages (FDL '19)*, Southampton, UK, Sept. 2019.

[25] S. Smyth, C. Motika, K. Rathlev, R. von Hanxleden, and M. Mendler. SCEst: Sequentially Constructive Esterel. *ACM Transactions on Embedded Computing Systems (TECS)—Special Issue on MEMOCODE 2015*, 17(2):33:1–33:26, Dec. 2017.

[26] S. Smyth, C. Motika, A. Schulz-Rosengarten, S. Domrös, L. Grimm, A. Stange, and R. von Hanxleden. SCCharts: The mindstorms report. Technical Report 1904, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2019. ISSN 2192-6247.

[27] S. Smyth, C. Motika, A. Schulz-Rosengarten, N. B. Wechselberg, C. Sprung, and R. von Hanxleden. SCCharts: the railway project report. Technical Report 1510, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Aug. 2015. ISSN 2192-6247.

[28] S. Smyth, C. Motika, and R. von Hanxleden. Synthesizing manually verifiable code for statecharts. In *Proc. Reactive and Event-based Languages & Systems (REBLS '18), Workshop at the ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*, Boston, MA, USA, Nov. 2018.

[29] S. Smyth, A. Schulz-Rosengarten, C. Motika, and R. von Hanxleden. The KIELER SCCharts Editor—A modular open-source modeling suite with automatic diagram synthesis. In *Proceedings of the Design, Automation and Test in Europe University Booth (DATE '19)*, Florence, Italy, Mar. 2019.

[30] S. Smyth, A. Schulz-Rosengarten, and R. von Hanxleden. Towards interactive compilation models. In *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018)*, volume 11244 of *LNCS*, pages 246–260, Limassol, Cyprus, Nov. 2018. Springer.

[31] S. Smyth, A. Schulz-Rosengarten, and R. von Hanxleden. Watch your compiler work — Compiler models and environments. Technical Report 1806, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2018. ISSN 2192-6247.

[32] 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. Long version: Technical Report 1311, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2013, ISSN 2192-6274.

[33] 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.

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

A Accompanying Poster

Guidance in Model-based Compilations

Kiel University

Department of Computer Science

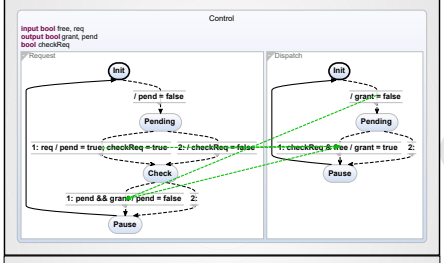
Causality Guidance

The modeler should not be burdened with maintaining an overview over all potential conflicts, but should be assisted with finding solutions to these. Modeling tools should provide transient views automatically while the modeler works on their model.

Transformation Observation

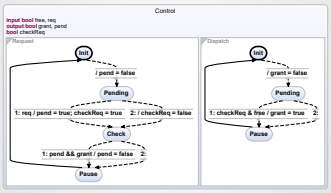
The modeler should be able to understand what is happening during transformations. Intermediate results, automatic mappings and meaningful annotations facilitate understandability and ease manual verifications.

Data Dependency View

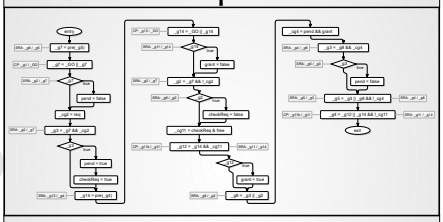


The data dependency view shows dependencies between different variable accesses to the same variable. Usually they govern the scheduling order of the program. The dependencies can be visualized on different granularity levels in the model.

Source Model

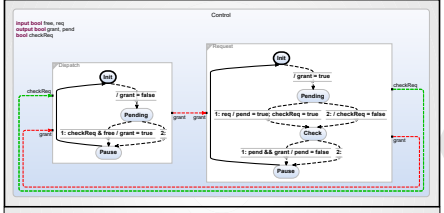


Transformation Snapshots



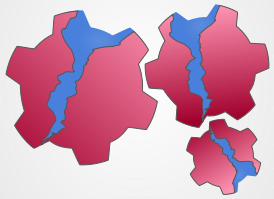
Every intermediate step of the transformation chain can be preserved as intermediate snapshot model by the transformations and enriched with individual annotations [1, 3]. The framework handles the mapping automatically. In the example, different classical optimizations, such as copy propagation (CP) and smart register allocation (SRA), annotate which nodes were modified by their process.

Induced & Causality Dataflow View

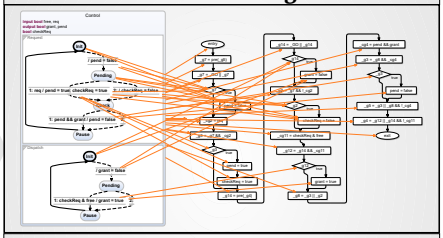


The induced dataflow view [4] shows communication between concurrent regions. It visualizes the datafow of the program; even if the underlying model uses the control flow paradigm. A variant thereof, the causality dataflow view, focusses on identifying data dependency cycles. If the source model is changed, such that conflicting values are written to the variable *grant*, the dependency cycle is depicted in red.

Model-based Compiler

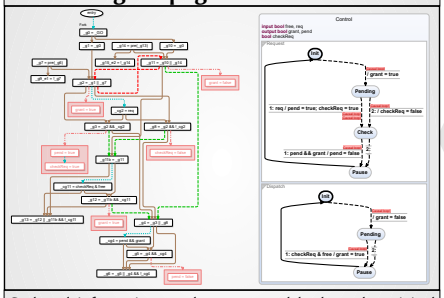


Automatic Element Tracing



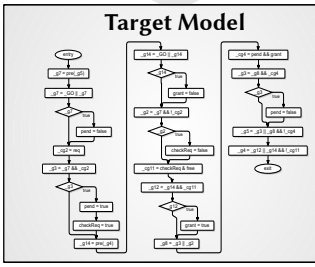
The compiler framework keeps track of the transitive model element relations [2]. Hence, the relation between the elements of arbitrary intermediate models of the compilation chain can be made visible. In the example the nodes of the final model of the netlist-based compilation [1] are mapped back to the original model.

Scheduling Propagation

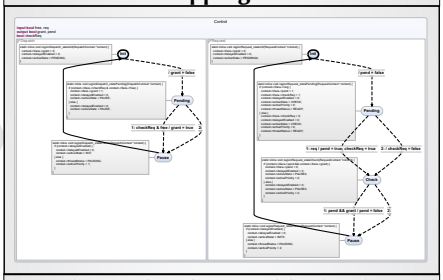


Gathered information can be propagated back to the original model to provide reasonable feedback. Here, complex scheduling relationships are simply displayed as annotations [3].

Target Model



Built-in Code Mapping



Generated target code can be annotated directly in the original model if supported by the chosen code generator. For example, SCCharts' state-based code generation creates one function for each state (besides other functions) [5].

Contact Persons

Steven Smyth,
Alexander Schulz-Rosengarten,
Prof. Dr. Reinhard von Hanxleden

Department of Computer Science,
Kiel University, Germany
Phone: +49 (0) 431 880-7290
ssm@/als@/rvh@informatik.uni-kiel.de
http://rtsys.informatik.uni-kiel.de

[1] C. Motika, S. Smyth, R. v. Hanxleden. Compiling SCCharts - A Case-Study on Interactive Model-Based Compilation. In Proc. of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14), Corfu, Greece, Oct. 2014.

[2] F. Rybicki, S. Smyth, C. Motika, A. Schulz-Rosengarten, R. v. Hanxleden. Interactive Model-Based Compilation Continued – Interactive Incremental Hardware Synthesis for SCCharts. In Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'16), Corfu, Greece, Oct. 2016.

[3] S. Smyth, A. Schulz-Rosengarten, R. v. Hanxleden. Towards Interactive Compilation Models. In Proc. of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'18), Limassol, Cyprus, Oct. 2018.

[4] N. Wechselberg, A. Schulz-Rosengarten, S. Smyth, R. v. Hanxleden. Augmenting State Models with Data Flow In Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday, Springer International Publishing, 2018

[5] S. Smyth, C. Motika, R. v. Hanxleden. Synthesizing Manually Verifiable Code for Statecharts. In Proc. of the Reactive and Event-based Languages & Systems (REBLS'18) Workshop, Boston, MA, USA, Nov. 2018

The KIELER SCCharts Editor is part of

KIELER
The Key to Efficient Modeling

On the web:
http://www.informatik.uni-kiel.de/rtsys/kieler
http://www.sccharts.com

Poster "The Kraken" presented at the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018), Limassol, Cyprus, 2018