Towards Interactive Compilation Models

Steven Smyth, 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/
{ssm,als,rvh}@informatik.uni-kiel.de

Abstract. A chain of model-to-model transformations prescribes a particular work process, while executing such a chain generates a concrete instance of this process. Modeling the entire development process itself on a meta-model level extends the possibilities of the model-based approach to guide the developer. Besides refining tools for model creation, this kind of meta-modeling also facilitates debugging, optimization, and prototyping of new compilations. A compiler is such a process system. In this paper, we share the experiences gathered while we worked on the model-based reference compiler of the KIELER SCCharts project and ideas towards a unified view on similar prescribed processes.

1 Introduction

In our previous publications towards a unified view of modeling and programming [9,13] we focused on the program/model that should be compiled. Using the model-based approach, we showed how a model-based compiler can transform a program to the desired target platform step-by-step while preserving the intermediate results. The approach, named SLIC for Single-Pass Language-Driven Incremental Compilation, was used to create the compiler for the synchronous language SCCharts [18].

While working on the model-based compiler, we since recognized that providing meaningful guidance for and resources to the developer does not solely depend on the artefact that should be compiled, but also on the process which performs the transformations. Instead of using a compiler that is particularly developed for a specific use case, we built upon the experiences we gained during the development of the SLIC approach to model the entire compilation process. Modeling the process provides us with new possibilities to aid the developer in their pursuit to create complex products, such as (1) arbitrary annotated intermediate models, and (2) the ability to change the compilation model at any time. We will demonstrate our generic framework in the following sections. For more concrete information on the compiler implementation and two technical use cases, we refer the interested reader to the associated technical report [15].

To illustrate the process and to continue the story told previously [9, 13], we use the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[1]

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

SCCharts language implementation as running example. KIELER is an academic open-source project that serves as a proof-of-concept platform. However, the approach presented here is not restricted to the SCCharts compiler, SCCharts, or KIELER, since every system of consecutive processes may be modeled and executed in a similar way. For example, our reference implementation within the KIELER project also includes compilers for languages such as C, Esterel, and various domain-specific languages as well as non-compilation tasks such as simulation of compiled artefacts.

Contributions and Outline

We here propose *process systems* that lift the concept of modeling such that models are not only used to specify some system under development, but also to specify *how* the modeling tool synthesizes an artefact that can be simulated and, finally, deployed. This approach not only gives the modeler full control over what the modeling tool should do, but also allows to inspect what it actually does for a specific functional model, along each step of the synthesis process. We introduce process systems in Sec. 2, beginning with an exemplary, abstract user story on classical programming and modeling. Sec. 3 takes a more detailed view of process system models, illustrated with our exemplary realization in the KIELER project. This demonstration serves as an example for similar process system modeling and is not restricted to the work done within KIELER. Further technical use cases can be inspected elsewhere [15]. Subsequent to related work in Sec. 4 we conclude in Sec. 5.

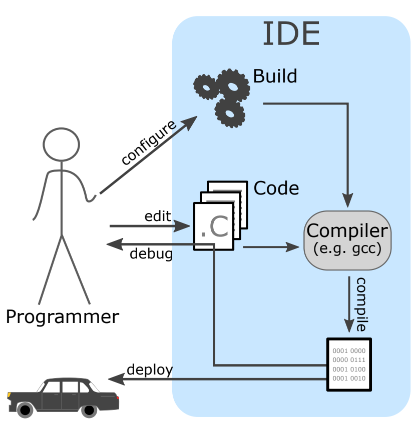
2 A Generic User Story

In this section we take a closer look at three alternative development processes sketched in Fig. 1. We assume that the developer uses an Integrated Development Environment (IDE) to work on a particular software project. Usually, the build process and/or project has to be configured by either the developer themselves or by another build expert Typically, the developer works directly on the artefact in question. However, the work foci differ.

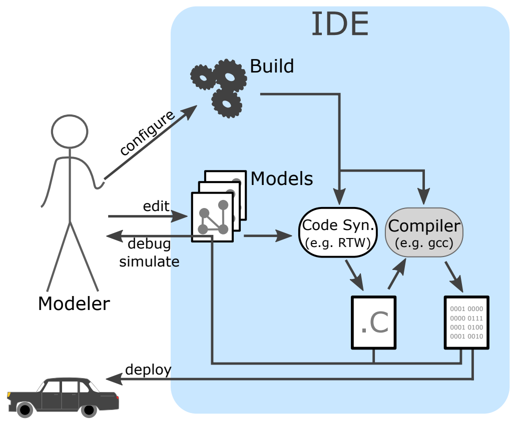
2.1 Programming vs. Modeling

Fig. 1a gives an abstract view on a **classical programming** development process, which is fairly straightforward. The IDE might be the Eclipse CDT[2]. The developer often has to be a programming expert and generally also configures the build process. While they usually work on one file at a time, they must keep an eye on the whole project, which is usually a collection of files, because it might influence the compilation. When they complete a development step, they issue

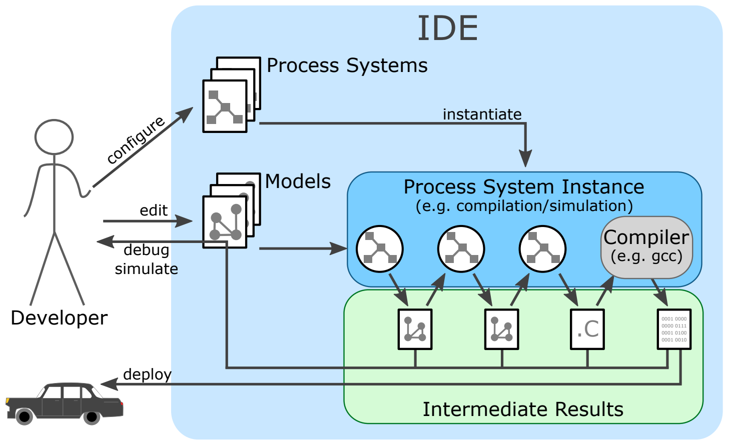
[2] https://www.eclipse.org/cdt



(a) An abstract view on classical programming development process



(b) An abstract view on classical modeling development process



(c) Development story with interactive process system instances

Figure 1: Three alternative development processes

a compilation command. An embedded (often external) compiler then compiles the source files to binary code that can be executed or embedded elsewhere if the source code is error free. Errors and warnings are fed back to the editor inside the IDE. They mark the erroneous line and give more or less processed information about the actual error or warning.

The **classical modeling** work-flow (Fig. 1b) looks actually quite similar. The modeler has to configure their project and can explore the project's files. Instead of editing a text file, the modeler usually works on a domain-specific, often graphical, model. The IDE, which may be something like Eclipse or a classical modeling tool, such as Matlab/Simulink or Ptolemy, uses an integrated code synthesis, such as the Real-Time Workshop (RTW) in Matlab/Simulink, to synthesize code. Similar to the classical programming paradigm, as soon as a development step is finished, the source models are compiled to a classical, general purpose language, such as C. Afterwards, they are compiled to binary code like before, with the addition that the user feedback often includes some sort of simulation. Here it depends on the concrete design choices if the simulation runs inside the IDE or on the compiled product.

Although the development processes are quite similar, there is a subtle shift in the focus on the developer. In the first case, the developer has to be a programmer, whereas the models in the second case are typically maintained by a domain expert. However, even in the second case programming experts are sometimes required to aid the modeler with special requirements or IDE extensions.

2.2 Interactive Process Systems

Fig. 1c depicts the more interactive approach that we advocate here. With *interactive process systems*, operating procedures, such as compilation or simulation, can be created and modified by the developer. Here, these process systems are simply models just like the working artefact, but perhaps models of another meta-model. When the modeler wants to compile (resp. simulate) the actual status of the model, the respective process system gets instantiated. Afterwards, the issued command can be processed by that system's instance. The feedback, including errors, *intermediate*, and *final results*, is directly available as individual model instances of appropriate meta-models. They can be inspected by the modeler or used as source for further systems. In the figure, we see an instantiated process system. The artefact is processed sequentially by single processors, e. g., model-to-model transformations, of the instance. All intermediate steps are observable. Eventually, the user wants to deploy binary code and one of the intermediate results may be general purpose source code that can be sent to an external compiler as before. Conceptually, the compiler call is just another process in the system's sequential chain.

Note that this approach is agnostic to the question whether the intermediate results (or, in fact, the original model) are graphical or textual. If the syntax is graphical, a key enabler to be able to represent the artefacts is the integration of automatic layout facilities into the modeling tool. In KIELER, we make use of the Eclipse Layout Kernel (ELK)[3] to synthesize the views. We argue that this is also an example of *pragmatic modeling* concepts [5,19], which aim to enhance modeler productivity by allowing to seamlessly switch between textual and graphical representations tailored to specific use cases. To quote a practitioner: "In our experience over many years my colleagues and I concluded that textual modeling is the only practical way, but that a graphical view of the models is a must-have as well. [KIELER] closes exactly that gap."[4]

The interactivity of the approach becomes apparent in the ability to observe all intermediate steps, to run system instances as they are needed and to create new or change existing systems all during run-time. There is no need to go

[3] https://www.eclipse.org/elk/

[4] Dr. Andreas Seibel, BSH Hausgeräte GmbH, E-Mail from Oct. 6, 2017.

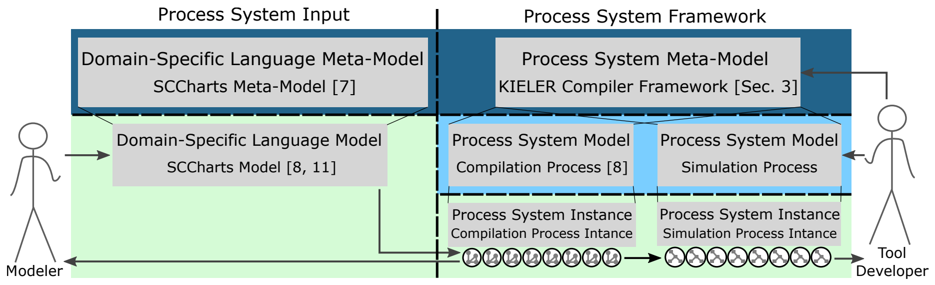


Figure 2: Process system's different model layer and user roles

through long re-build and re-start cycles. As described before, a process system can basically perform any kind of job. As an example, the figure depicts a model-to-model (M2M) compilation. Here, technically, the term *interactive* subsumes the dynamic nature of the approach, meaning that instances of systems are generated dynamically as they are needed. These instances carry dynamic properties on their own and live as long as they are required. This also resembles the classical class-object hierarchy of the object-orientated paradigm.

Another take on this is to view process systems as a—rather abstract—data flow model. Traditionally, data flow models are collections of *actors* that consume and produce *data* [7]. Conceptually, the processors of a process system correspond to actors, and the data they consume and produce correspond to the intermediate results generated along a synthesis chain. One difference is that in process systems, each actor typically "fires" only once, and the "schedule" that governs how the process system is executed/instantiated is rather simple, usually just a single sequential execution of all processors. However, more complex, dynamic execution schemes are also possible, as alluded to later in Sec. 3.2. Alternatively, if we want to focus on the schedule of the processors, we can also view process systems as control-oriented state machines, where one processor can be "active" at a time, and when it is finished, control advances to the next processor. This view can be helpful for example to define more elaborate schedules, but hides what is actually produced and consumed by the processors, which is why we consider the data-flow analogy typically more fitting than the state-oriented analogy.

Due to the interactivity of the approach, tool developers and modelers can easily create, explore, and modify different aspects of the whole development process. The difference is not disparate work-flows, but the diverse work-flow artefacts that are being worked on. Fig. 2 shows the different layers of models and the two main roles of users. On the left side, the modeler mainly works on the system's input, e. g., a particular model in a specific domain-specific language (DSL). The model's meta-model also belongs to the system's input, but is usually outside of the modeler's scope w.r.t. making changes during a particular project. In the example in the figure, the modeler works on an SCCharts model, whose syntax is defined in the corresponding meta-model. Again, it is not important here wheter the syntax is textual or graphical. This decision can be made as the case arises depending on the preference of the domain expert. At this, the form of editor must not be the same as the graphical view of the model. Transient view and automatic layout technologies [14] may help the modeler to explore the model without getting distracted by tedious tasks, such as placement and alignment of graphical elements.

On the framework's side on the right, there is also the framework's meta-model for defining system models. Derived from this, different systems can be created that hold the necessary instructions. These systems can be instantiated to be applied on a specific artefact. In the example shown, the created SCCharts model is fed into an compilation system instance. During compilation, several observable intermediate results are created. The result of the whole context also serves as input for a simulation instance.

In general, the modeler will be more interested in the actual project's model and the systems' results, whereas the tool developer's focus will lie on the systems and the underlying framework, including the relevant meta-models. However, both can utilize all aspects of the development process to drive their work. For example, the modeler may also change a particular system to toggle optimizations if necessary. More obviously, the tool developer can use different model inputs to test and extend the framework. This could also lead to closer feedback loops between domain experts and tool developers.

3 Process Systems in KIELER

In the KIELER Compiler (KiCo) the smallest compilation unit is called a *processor*. We moved away from the specific term *transformation* to emphasize that a processor does not have to perform a transformation. Instead processors are categorized into *transformer*, *optimizer*, and *analyzer* to specify their role. A variety of tasks can be implemented as processors, such as M2M transformations, optional optimizations, and, e. g., object counting, but this should be restricted to this atomic task to facilitate modularity and reuse.

3.1 Static Process Systems

A list of processors forms a *process system*. These systems describe a single compilation from a certain source type down to the desired target. When compared to the object-orientated programming paradigm, process systems can be seen as classes. They can be instantiated to perform a task for a concrete artefact. In the previous publications we described two compilation approaches, the netlist-based and the priority-based approaches, for SCCharts [9, 18]. Each of these is an own process system, more specifically a *compilation system*.

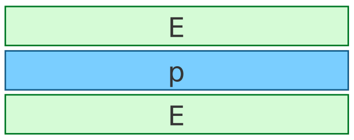


Figure 3: The atomic compilation unit, a processor `p`, receives a source and a target environment when instantiated.

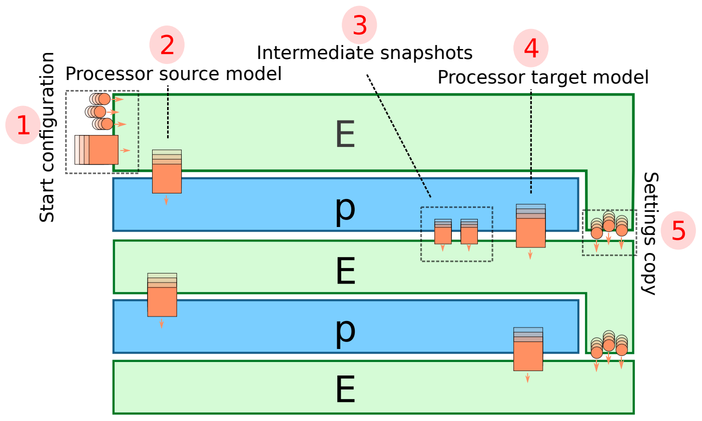


Figure 4: Concept of a compilation context with two processors.

When a system is instantiated, an instance for every processor within the system is created. A *processor instance* is then connected to a *source environment* from which it will receive input data and a *target environment* to work on and store data for the next processors. The simplest system possible is shown in Fig. 3. It consists of a single processor with its corresponding source and target environments. Once the system is fully instantiated, a new *compilation context* (gen. process context) exists, which can be used to compile an artefact. While the context, including all environments and all data, is observable during compilation, it will remain accessible even after the compilation finished until discarded, so that all data and results can be inspected as long as desired.

Conceptionally, the developer is free to choose the nature of their environments. In KIELER we use typed, but arbitrary data storages. Hence, processors may store arbitrary (ancillary) data in the environments, but have a form of type-safety when accessing it.

The developer does not have to bother with all the instances and environments. The KiCo framework will do most of the work. In general, when invoking a compilation programmatically, one only needs two lines of code. First, a *context* has to be created. The context needs to know which system model it should use and on which artefact the compilation should be invoked. Once the context is created, the compilation can be executed. List. 1.1 shows an excerpt from the KIELER project where a compilation is started asynchronously as soon as the user presses a particular button. The programmer could make adjustments to the context before the compile method is executed, but in this case, it is not necessary.

```
val context = Compile.createCompilationContext(view.activeSystem, model)
context.compileAsynchronously
```

Listing 1.1: Compilation invocation excerpt from the KIELER project

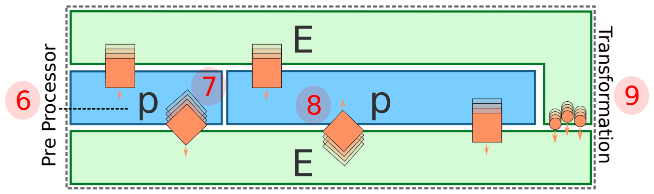


Figure 5: To save resources several processors can be grouped together. Generally, everything that happens between two environments is commonly called a transformation.

Usually a compilation system includes more than one processor. Fig. 4 shows how processors interact with their environments to orchestrate the entire compilation. As described before, once a context has been created, it needs an input artefact and can be configured if necessary. The first environment in the context receives the *start configuration* ① as can be seen in the figure. After the invocation of the compilation, the first processor begins its work. It fetches the model from its source environment ② and begins its computations. That model is the *source model* from the processor's perspective. While working on the model, the processor can do several snapshots of the current state and store them in its *target environment* ③. These intermediate states can be inspected during or after the compilation. At the end of the processor's job, the result is saved ④. In the example, the graphic indicates that the result is of the same meta-model as the source model. However, any type can be used. E. g., as targets are often other programming-languages, the backends usually give simple text as results. Once the processor terminates, the next processor starts its job. From its perspective, the former target environment now becomes the source environment and the processor can work on the next one. The framework takes care that all settings, model references, and additional auxiliary data get copied to the new environments if necessary ⑤.

To facilitate modularity and consume less resources, processors often perform pre- or post-processing jobs for transformers without the need of dedicated environments as depicted in Fig. 5. Hence, a processor can run with the same environments as another one ⑥. In the example, the job saves a second model with a different meta-model in the environment ⑦. This secondary model may store ancillary data (e. g. loop information from a loop analyzer), which can be picked up by subsequent processors ⑧. Usually, what is commonly called *transformation* is everything that happens between the source and the target environments ⑨. The result that is stored in the last environment represents the result of the whole compilation.

Note that pre- or post-processors also store these data in the target environment of the main processos as they are not allowed to change the source environment. However, the developer is not required to handle these inputs differently as the framework will ensure correct accesses. In fact, technically, KiCo pro-

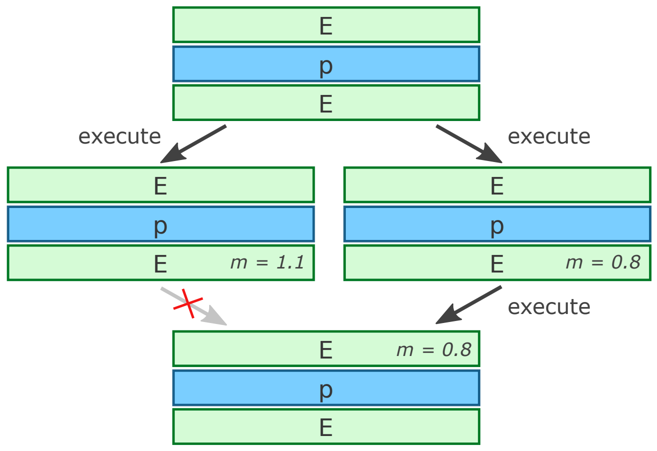


Figure 6: When joining different branches, model measures rate the quality of the preceding results to determine the new source environment.

cessors internally always only work on the target environment. The framework automatically creates a copy from the source environment before a processor is called.

To be even more resource-saving, a compilation can be set to *in-place*. Compiling in-place does not create new model instances to work on. The processors all work on the same models, hence intermediate results are only observable during compilation and only one at a time. At the end of the compilation, only the final result remains. Conceptually, this would also look like the schema in Fig. 5 where only two environments exist and all processor instances live in between.

3.2 Dynamic Process Systems

The KiCo framework can also handle branching. As we tend to create small, concise systems and this mechanic is rarely used at the moment, we will only sketch the two predominant concepts of KiCos decision-making. Succeeding processors are always executed in KiCo. In the first approach, when joining different branches, KiCo compares quality measures (m) inside the joining environments to decide which result will be the source for the joining processor as can be seen in Fig. 6. The measure is determined by a post-processor and can be handled and customized like every other processor. Which characteristics of the model are used to determine the value is up to process system. By definition, a smaller value generally means a better result, e. g., model size. In the figure two paths branch from a source. On both paths, a processor performs its job. The result is then judged by a *measuring processor*. Compared to the source model, the result on the left branch is greater, i. e. worse. Subsequently, the right branch is chosen for the joining processor. Note that this mechanism can also be used to exclude invalid paths. An invalid model results in an infinite measure (∞) and is discarded. This, however, depends on the task of the processor. For example, if an optimizer fails, it should simply return the source model with $m = 1.0$ as a failed optimization should not change the artefact semantically.

In the second approach, as a process system is also a model and accessible from the contained processors, a processor can alter the process system and, therefore, affect the succeeding processors. Hence, it is also possible to decide for the next processor during run-time. This is particularly helpful when a static schedule is not determinable, as has been shown by Rahimi-Barfeh [12].

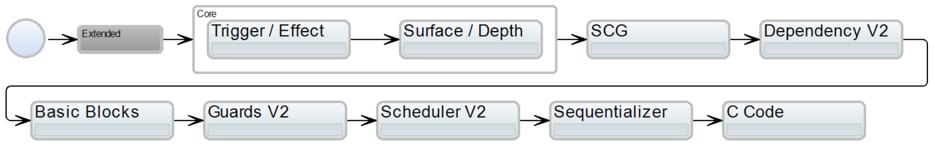
3.3 SCCharts Compilation

The default version of the netlist-based compilation system uses 33 processors. List. 1.2 shows a shortened description for the netlist-based compilation of SCCharts. Every processor has its own unique identifier. However, compilation systems are often composed of other systems, which can be referenced. Here, the downstream compilation builds upon the standard high-level SCCharts compilation (line 4–5) and nine further processors identified by their identifiers.

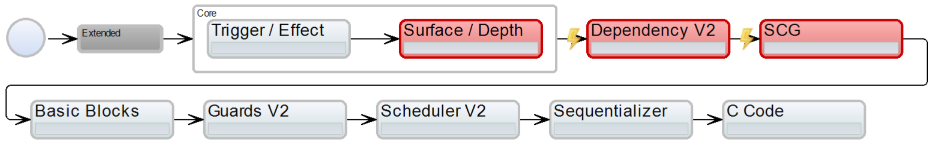
As these descriptions define compilation models interactively, we use concepts such as transient views [14] to visualize the system graphically and, if necessary, point to problems such as unknown processors or type incompatibility between processors. *Interactively* means that we can inspect, change, and save the model during runtime to invoke altered compilation runs without the need of long re-configure and re-start cycles. Fig. 7a shows the automatically generated graphical representation of the netlist-based compilation system during editing. This view is synchronized with the editor of the model's description

```
1  public system de.cau.cs.kieler.sccharts.netlist
2      label "Netlist−based␣Compilation"
3
4  system de.cau.cs.kieler.sccharts.extended
5  system de.cau.cs.kieler.sccharts.core
6  de.cau.cs.kieler.sccharts.scg.processors.SCG
7  post process de.cau.cs.kieler.scg.processors.threadAnalyzer
8  de.cau.cs.kieler.scg.processors.dependency
9  de.cau.cs.kieler.scg.processors.basicBlocks
10 post process de.cau.cs.kieler.scg.processors.expressions
11 de.cau.cs.kieler.scg.processors.guards
12 de.cau.cs.kieler.scg.processors.scheduler
13 de.cau.cs.kieler.scg.processors.sequentializer
14 de.cau.cs.kieler.scg.processors.codegen.c
```

Listing 1.2: Model description of the netlist-based SCCharts compilation



(a) SCCharts netlist-based compilation system. In this view, the Extended system is collapsed and the Core system is expanded.



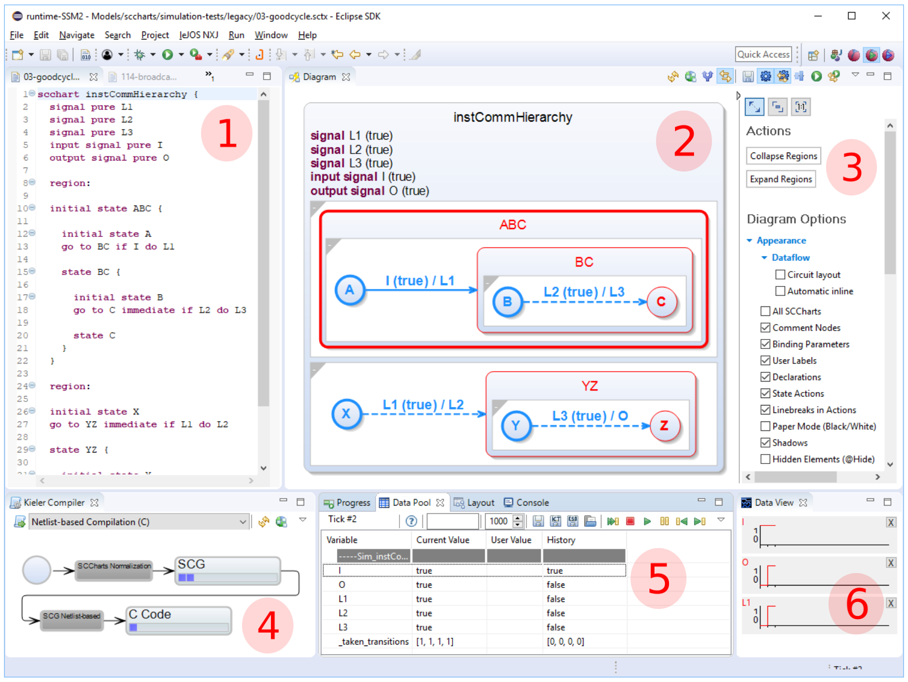
(b) While modeling, errors, such as type incompatibility, can be highlighted immediately.

Figure 7: Example of an automatically generated graphical (view) of a compilation system

and instantaneously re-generated upon change. The referenced high-level SCCharts systems can be expanded and collapsed for readability. Problems appear in red. The generated views are also used as control panel in the KIELER project to invoke the compilations and to select intermediate results.

In the example depicted in Fig. 7a, the Surface / Depth processor creates an SCCharts model which is then transformed to its corresponding Sequentially Constructive Graph (SCG), a sequentially constructive variant of a control-flow graph, by the SCG processor. The subsequent Dependency processor expects an SCG as input. If one would swap the SCG and Dependency processors, the compilation chain becomes type incompatible, as depicted in Fig. 7b.

Fig. 8 shows a complete example of a running KIELER instance during simulation. In the SCCharts editor tool, the abstract model is described with a textual syntax 1. A graphical view of the model is instantaneously generated by the transient view framework [14] 2. The user can further influence the visualization of the presented data via options on the right sidebar 3. However, these options consist mainly of rather coarse convenience settings to set the current focus to specific points of interests. 4 – 6 show examples of different information views. These can be configured (and saved per perspective) individually. Together with the transient live visualization 2, they resemble the systems and intermediate result regions from the previous figures. The selected compilation system is depicted in 4. A view to manipulate the running simulation is open in 5. Selected data observers can be inspected in 6. Note that information of the running simulation is visible in the model diagram 2, the simulation view 5, and the observers 6 simultaneously. The variable states and current active model elements can be highlighted directly in the model. The user can input new environment settings in the simulation view. Here, one can also control single



```
sccchart instCommHierarchy {
  signal pure L1
  signal pure L2
  signal pure L3
  input signal pure I
  output signal pure O

  region:

  initial state ABC {

    initial state A
    go to BC if I do L1

    state BC {

      initial state B
      go to C immediate if L2 do L3

      state C
    }
  }

  region:

  initial state X
  go to YZ immediate if L1 do L2

  state YZ {

```

signal L1 (true)
signal L2 (true)
signal L3 (true)
input signal I (true)
output signal O (true)

- Appearance
  - Dataflow
    - ☐ Circuit layout
    - ☐ Automatic inline
- ☐ All SCCharts
- ☑ Comment Nodes
- ☑ Binding Parameters
- ☑ User Labels
- ☑ Declarations
- ☑ State Actions
- ☑ Linebreaks in Actions
- ☐ Paper Mode (Black/White)
- ☑ Shadows
- ☐ Hidden Elements (@Hide)

Tick #2 1000

Variable	Current Value	User Value	History
-----Sim_instCo...			
I	true		true
O	true		false
L1	true		false
L2	true		false
L3	true		false
_taken_transitions	[1, 1, 1, 1]		[0, 0, 0, 0]

Figure 8: Complete example of a running KIELER instance during simulation.

forward and backward steps of the simulation. Furthermore, the actual and past data of selected variables can also be visualized in the data observer 6.

4 Related Work

Steffen already showed a close relation between compilation and modeling back in 1997 [16]. He proposed to use *consistency models* to detect inconsistencies between different model descriptions. This relates to giving a semantics to a programming language by translation into an intermediate language. Over the years, a number of modeling compilation approaches have been developed such as Cinco [10], a meta-level modeling tool generator, and Marama [6], which provides metatools for language specification and tool creation. KiCo's process categorization into specialized work units is in line with ETI's process system [17]. While targeting a slightly different group of experts, such an even more generic process synthesis approach could also be implemented in KiCo. We discuss further possible future routes in the conclusion in Sec. 5. 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 [14].

The proposed process systems can be seen as a variant of *scientific workflows* [4], as implemented in tools such as TAVERNA [11], for M2M transformations combined with state-of-the-art pragmatic modeling techniques. While there are similarities, such as a loose processor concept and type-checking, the focus of KiCo lies on M2M transformations where every intermediate result is a fully functional artefact. The processor system itself, including the environments in a running context, is also considered a simple model here. There is no need for a specialized description language or special data storages, e. g., for processor meta-data such as processor run-time. In our approach, the system's model can be influenced during design- and run-time, which includes alterations by the contained processors. Furthermore, as long as the transient view framework supports the meta-model of the intermediate results, views can be generated instantaneously and there is no difference between the different artefacts, even if they are positioned on distinct meta-levels.

When it comes to general compilation techniques, numerous well-understood approaches (e. g. Copy Propagation [1] and Register Allocation [3]) can be applied to our compiler to improve the results. However, as classical compilers are more or less a *blackbox*, working with intermediate results becomes difficult. For example, as depicted elsewhere [15], the gcc[5] possesses settings to toggle different optimizations or to print out intermediate representation of the basic blocks [2] of a source program. However, the interplay between the different modules and the textual representation of data seems to only target compiler experts and is arguably rarely useful for the common user.

5 Conclusions

We presented how the compiler framework works that was used to create the reference implementation for the synchronous language SCCharts. We showed that process systems, such as compilation or simulation, themselves are also models and how this can help both, the domain expert and the tool developer, with the goal to get better results faster and to increase modeler productivity. While both may have different foci during a project's lifetime, both can use a similar framework to drive their development and to help each other.

For us, in Model-driven Software Development (MDSD), programming with models does not only mean to model a program with, e. g., a sophisticated IDE that provides us with new tools to construct the program. For example, modern programming IDEs provide features such as syntax highlighting, code completion, reference counting, refactoring, etc. Many of these features focus on creating the model and then they are done. In our approach, the whole process is modeled. The user can inspect and change every part of it interactively. They can influence the compilation improving the final result or add new processors that provide new models and views to give better feedback. We thus argue that MDSD is not solely about modeling a particular artefact. It is also about the way to get to the final deployable software.

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

Besides further improvements for the SCCharts compiler and streamlining the MDSD user experience, we see further future work. For example, the KIELER project includes several modules that still use dedicated components that perform dedicated model transformations to prepare the models for specific tasks. As illustrated in Sec. 2, we are currently working on the compilation and simulation systems. However, the KiCo framework could also be used to generalize even more of these processes, e. g., deployment tasks. This would also facilitate the re-usability of the approach beyond the classical compilation task. Furthermore, we want to combine our approach with the continuing trends of mobile location-independent technologies, such as mixed web/desktop applications using tools such as electron[6]. We are optimistic that this will further increase the possibilities for and flexibility of prototyping and team-driven software development.

References

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. 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.
4. 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.
5. 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.
6. 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.
7. E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers (JCSC)*, 12(3):231–260, 2003.
8. 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.
9. 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.
10. 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.

[6] https://electronjs.org

11. T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.
12. M. Rahimi-Barfeh. Incremental compilation of SCEst. Bachelor thesis, Kiel University, Department of Computer Science, Sept. 2017. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mrb-bt.pdf.
13. 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 8802 of *LNCS*, pages 443–462, Corfu, Greece, Oct. 2016.
14. 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.
15. 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.
16. B. Steffen. Unifying models. In *STACS 97, 14th Annual Symposium on Theoretical Aspects of Computer Science, Lübeck, Germany*, pages 1–20, Mar. 1997.
17. B. Steffen, T. Margaria, and V. Braun. The Electronic Tool Integration platform: concepts and design. *International Journal on Software Tools for Technology Transfer*, 1(1):9–30, Dec 1997.
18. R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pages 372–383, Edinburgh, UK, June 2014. ACM.
19. R. von Hanxleden, E. A. Lee, C. Motika, and H. Fuhrmann. Multi-view modeling and pragmatics in 2020 — position paper on designing complex cyber-physical systems. In *Proceedings of the 17th International Monterey Workshop on Development, Operation and Management of Large-Scale Complex IT Systems, LNCS*, volume 7539, Oxford, UK, Dec. 2012.