Interactive Transformations for Visual Models*

Ulf Rüegg, Christian Motika, Reinhard von Hanxleden
Christian-Albrechts-Universität zu Kiel
{uru,cmot,rvh}@informatik.uni-kiel.de

Abstract:

Model transformations are an essential and integral concept of Model Driven Engineering (MDE). However, when using state of the art modeling tools, transformations are typically executed silently and at once in the background. This lacks flexibility and does not reveal any insights of the transformation process, which makes it hard to understand and debug specific model transformation implementations.

We present a flexible concept to define arbitrary model transformations combined with graphical visualization and user interactivity. The key idea is the integration into a view management to obtain various possible visualizations and user interaction. In the paper we focus on applying this concept to monolithic transformations by breaking them up to gain tight and fine-grain control. This includes the capability of undoing transformation steps.

To evaluate our approach, we present an Eclipse integrated visual and interactive execution layer for model transformations. As a case study we show an example transformation implementation from the synchronous textual language Esterel to SyncCharts, a synchronous Statecharts dialect.

1 Introduction

To handle the complexity of software for Cyber-Physical Systems (CPSs) [LS11], Model Driven Engineering (MDE) has recently gained attention especially within the widely used Eclipse[1] tooling environment. The main reason for this is that Eclipse is very modular, which allows various extensions. Several mature projects build on this platform, with the Eclipse Modeling Framework (EMF) leading the way. Model transformations are an integral part of MDE and there is good support inside the Eclipse framework as various Eclipse projects offer the transformation of EMF-based models.

Reasons for model transformations can be to separate concerns directly in the modeling workflow. This allows a generative approach by adding more and more specific information while transforming from one model instance to the other. This concept itself is used by Eclipse projects like the Graphical Modeling Framework (GMF). Other rationales for model transformations may be structural model changes [KPRP07], code generation and simulation desires [MFvH10] or the synchronization between different views of a model.

*This work was funded in part by the Program for the Future Economy of Schleswig-Holstein and the European Regional Development Fund (ERDF).

[1] http://www.eclipse.org

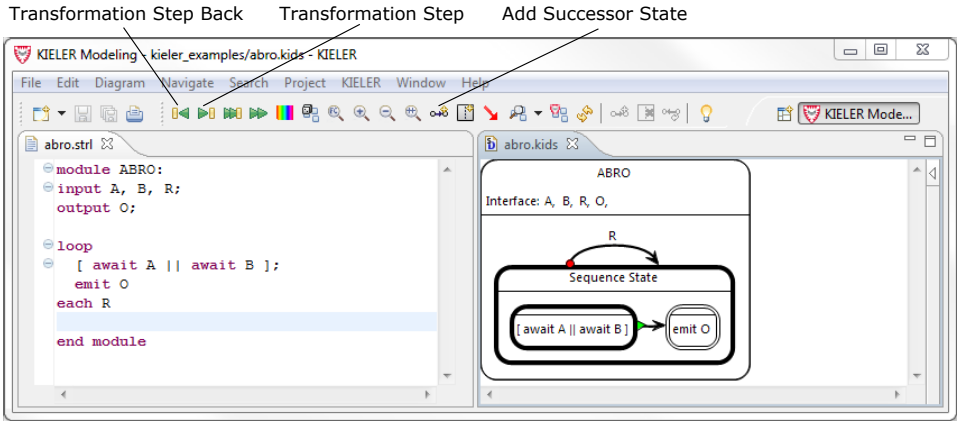


Figure 1: User interface of KIELER during a step-wise transformation

Typically, these transformations are executed silently and at once in the background. The user does not gain any insight into the process of the transformation. The concrete relations between the input model and its transformed version do get lost. Furthermore, the user has no tight control about the transformation while it is applied. Developers of transformations often have a hard time in debugging such model transformations or parts of them.

Contributions: We present a flexible concept to define arbitrary model transformations combined with graphical visualization and user interactivity. In this paper we focus on applying this concept to monolithic transformations by breaking them up into *steps* of a certain granularity.

The key idea is the integration into a view management as described by Fuhrmann [FvH10] that allows to reuse various visualization effects and to define arbitrary user-interface elements. Therefore, a developer can adapt visualizations and user interaction according to the specific use case, e. g., by allowing to execute steps, backward steps (undo), or to perform the overall transformation. The approach is not restricted to a certain transformation language. While the developer has to know about the technologies he uses, the user only needs to be familiar with the elements provided to the user-interface.

As this work is implemented as a proof of concept in the context of the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) project, it is based on and integrated into the Eclipse framework and can be reused by other EMF-based projects. We illustrate the generality of the approach with prototype implementations that combine the ideas of interactive visual transformations with already existing generic *model-view-management* implementations of the KIELER project.

Outline: Section 1.2 briefly introduces into the KIELER framework, which serves as the context for our implementation. Section 1.3 discusses meta modeling and model transformations. Related projects are referenced in Section 1.4. In Section 2.1 we describe the

motivation and the idea of user interaction during model transformations. Section 2.2 explains the term *visual transformation*. Section 3 illustrates the integration in KIELER with a case study. Section 4 shows how the presented technique can be used to enhance existing user interaction paradigms. Section 5 concludes and gives some outlook on future work.

1.1 View Management

Fuhrmann and von Hanxleden [FSMvH10] describe the basic ideas of a view management. It consists of three parts:

Effect: Effects are activities that can be released by the view management. Examples are coloring of diagram elements or automatic layout.

Trigger: Triggers serve as event listeners and can result in released activities controlled by the view management, i. e., effects. Examples are button, selection, or simulation events.

Combination: Combinations connect certain triggers with effects. Special logic can be provided. While effects and triggers are mostly predefined, combinations are often written by developers and can be rather simple.

In the model transformation approach presented in this paper we use a view management infrastructure to achieve a clean separation between the transformational and the visual part. We react to the user pressing buttons monitored by view management button triggers. A transformation combination reacts accordingly with certain visual transformation effects.

1.2 KIELER and KiVi

The KIELER framework is a set of open source Eclipse plug-ins that integrate with common Eclipse modeling projects, such as GMF and especially the modeling backbone EMF. Eclipse provides several different projects for model transformations that are already used inside KIELER. A key enabler of KIELER is the automatic layout of graphical models. This enables features such as structure-based editing [FSMvH10] or the synchronization between textual and graphical models [Sch11]. These features only modify the underlying domain model and rely on an automatically synthesized or updated and arranged graphical view of this model.

Figure 1 shows the KIELER Eclipse environment with an Esterel code editor and a graphical Statechart editor during a visualized interactive transformation. Additionally, toolbar buttons serving as the UI for interactivity during the transformation are visible at the top.

The KIELER View Management (KiVi) is the implementation of a view management and an essential core part of the KIELER framework.

1.3 Meta Models and Model Transformations

A basic prerequisite for MDE are models that base on meta models. The latter define the abstract syntax of models and hence allow the specification of languages as object-oriented structure models. The Meta Object Facility (MOF) is a meta modeling framework defined by the Object Management Group (OMG)[2], which has been taken shape for the *Eclipse* world as the Eclipse Modeling Framework (EMF) with its *Ecore* meta model language that we use in the context of this project. The MOF defines four layers M0 to M3, where M3 corresponds to highest level of abstraction (meta-meta models). The M2 layer contains the aforementioned meta models, M1 the concrete models, and M0 the physical, real-world objects.

Model transformations play a key role in generative software development. They describe the transformation of models (i. e., meta model instances) that conform to one meta model into models which then conform to another or even the same meta model. Mens and Gorp [MG06] specify some terminology concerning transformations. A transformation is called *exogenous* if the M1 source model and the M1 target model are derived from different M2 meta models. *Endogenous* transformations base on the same M2 meta model. They are called *in-place* if the source and target M1 model is the same instance. A *transformation rule* describes how to transform a certain element of a source model into the corresponding element in the target model.

There are several model transformation systems available today that are well integrated into the Eclipse platform. Xpand[3] realizes a model to text template based approach, Xtend is a functional meta model extension and transformation language based on Java with a syntax borrowed from Java and OCL, and there exist other transformation frameworks such as QVT or ATL.

In our implementation we will use the Xtend language as it is widely used and was refactored for a seamless integration into the Eclipse IDE. Additionally, its extensibility features allow to escape to Java for sequential or complex transformation code fragments. Nevertheless, our approach is conceptually open to use any transformation language.

1.4 Related Work

With the growing importance of model transformations, much work has been devoted to improve the definition, verification, and comprehensibility of such transformations by providing suitable formalisms and tooling.

The Epsilon Wizard Language (EWL) [KPRP07] allows interactive in-place model transformations and is integrated into EMF. So-called *wizards* can be applied to certain elements of a model. Wizards specify a *guard* determining for which model elements they are applicable, and a *body* defining the actual model transformation. In this approach the user must

[2] http://www.omg.org/spec/MOF/2.0/PDF
[3] http://wiki.eclipse.org/Xpand

select both, the single model element and the wizard to execute. No advanced automatic layout or further visualizations can easily be used. Wizards can only be specified using EWL. It is not possible to mix transformation technologies.

The ideas presented in this paper concern unidirectional transformations. However, there exist approaches that allow to define bidirectional transformations. These are usually more complex and not always desired. E. g., the re-generation of source code after changes were made to its previously generated documentation is unwanted in most cases. One example are Triple Graph Grammars (TGGs) [GW06], a formalism used to define the correspondences between two distinct models. A triple graph consists of two graphs representing the models and a third correspondence graph associating objects of the two models with each other.

A bidirectional approach to fully integrate textual and graphical modeling is presented by Schneider [Sch11]. This includes the immediate synchronization of either model upon any changes that were made in the other model and possibly additional visualization. He also provides an exemplary implementation by using KIELER's SyncCharts editor and a textual Domain Specific Language (DSL) for SyncCharts. Such a synchronization is only possible for two representations that are transferable into each other without losing contents. However, if the synchronization is realized by using two distinct transformations for each direction, it will be possible to use the approaches presented here.

ATL and QVT are transformation languages coming with sophisticated tooling, which provides well-known code-line based debugging mechanisms. We present concepts that allow to debug a transformation by looking at intermediate states of the graphical model itself instead of code lines and variable contents. Also, this allows to debug transformations written in a language that does not have native debugging facilities.

Vanhooff et al. [VAB+07] describe concepts to decompose transformations into sub-transformations and to compose sub-transformations using diverging technologies. They distinct the specification, implementation, and execution of a transformation to obtain a clear separation of concerns. Consequently, they differentiate between the roles of a transformation specifier, developer, and assembler. It might be interesting to integrate these strict distinctions into our progress of defining a transformation. We also aim at the reuse of different sub-transformations by possibly new developers who, i. e., would benefit from clear specifications.

Contrary to our use of the term *visual transformation*, meaning the visualization of intermediate transformation results, Biermann et al. [BEK+06] present an environment allowing the visual definition of model transformations by using enhanced graphs.

2 Visualized Interactive Transformations

Figure 2 shows the transformation of an Esterel program into a SyncChart, which will be explained in further detail in Section 3.1. Ordinary transformations take the Esterel program, apply the transformation, and return the SyncChart presented on the right hand side of the figure. This does not reveal the relationships of a source and a target model's

element and prevents the understanding of how the transformation works.

We aim to improve the comprehensibility and understanding of model transformations by adding interactivity and visualizations. The *interactive* and the *visual* part of the presented approach are addressed separately in the following, as they differ in domain dependency.

2.1 Interactive Transformations

Interactivity means the interaction of a user with the program. Using an interactive transformation the user decides which part of a model to transform, at what point of time, and to which extent, e. g., for only half the model, it should be transformed.

To provide this functionality, UI elements, e. g., buttons or keyboard shortcuts, are mandatory to allow the user to control the transformation. Furthermore, the overall transformation needs to be split into smaller pieces that are executable individually. As mentioned in Section 1.3, a transformation is the complete conversion of a source model into a target model. A *transformation rule* is the definition of how to convert an element of the source model into an element of the target model. Hence, a transformation rule is just a piece of the overall transformation and we refer to the execution of this piece as *step*. The possible granularity of those steps depends on the definition of the transformation rules for a specific transformation.

Figure 3 exemplarily sketches a possible interaction of a user with a transformation. A source model is transformed into the target model in four steps, each of which is triggered by the user. In every step a certain transformation rule is applied, which is determined, among others, by the current user input and/or the model state.

Certain information is necessary to execute a transformation. We divided this information into a *transformation description* and a *transformation context*. The context contains static information such as the files holding the transformation rules or the facilities of a certain transformation technology to apply a transformation. The description holds dynamic in-

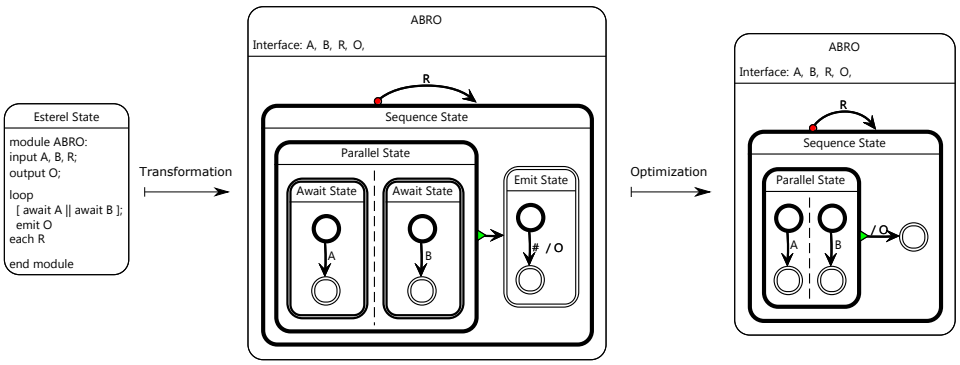


Figure 2: Overall transformation of the ABRO program.

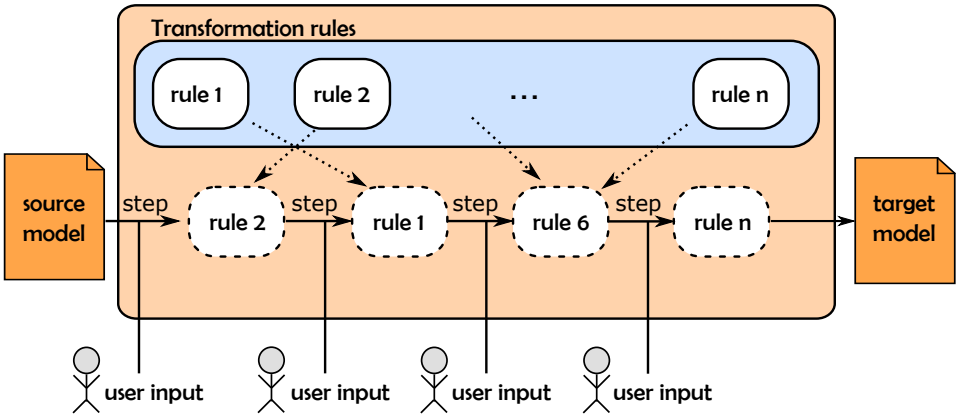


Figure 3: A possible interaction of a user with a big transformation, broken up into several transformation steps to show potential for interactivity during a model transformation

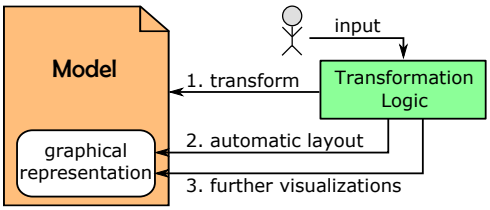


Figure 4: Applying visualizations after a transformation

formation such as the model elements that should be transformed and the name of the transformation rule that should be applied in the next step.

2.2 Visual Transformation Approach

As mentioned above, the visualization of intermediate steps of a transformation depends on the domain. Basically, there are two cases to distinguish. First, consider endogenous transformations with the same M2 meta models. In this case, changes inflicted by the transformation rules can be applied incrementally to the model until the transformation is completed. Additional visualizations, such as highlighting, can be added easily.

Second, exogenous transformations work with M1 models that differ in M2 meta models. In this case, it is necessary to embed the notation of the source model into the target model or vice versa. Therefore, after the execution of a transformation rule the target model might contain notations of both meta models.

Additionally, automatic layout is mandatory when working with graphical models [FvH10].

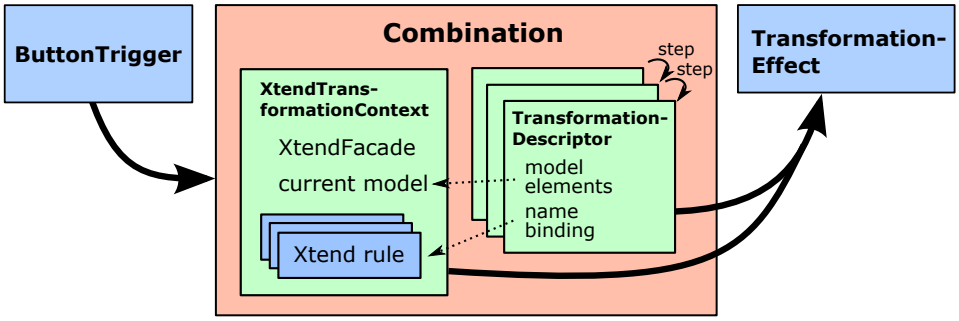


Figure 5: Integration of transformations into view management

A transformation rule is likely to add or remove graphical elements. Newly added elements need to be positioned properly in the diagrams and they need to preserve the comprehensibility of intermediate steps. If left to the user, this positioning will tend to be rather time consuming and frustrating when the same elements have to be arranged manually over and over again. This would nullify the benefits of an interactive transformation.

Figure 4 shows the integration of visualizations into the transformation process for a graphical model. First, the model is transformed according to the user's inputs. Second, automatic layout is applied to the graphical representation of the model. Third, an arbitrary number of visualizations is added. The visualizations are either defined by the transformation logic or by the user. These three points are performed after each transformation step initiated by the user, see Figure 3.

3 Integration into KIELER

To evaluate our approach we integrated the execution of single transformation steps in KIELER. We did this in a generic way that can be used for an arbitrary transformation. In Section 1.2 we introduced the KiVi project, which allows the execution of `Effects` by using `Combinations`. Figure 5 illustrates the integration of transformations in KiVi.

To describe a transformation, we introduced an `ITransformationContext` interface and a `TransformationDescriptor`, relating to the context and descriptor mentioned in Section 2.1. The `ITransformationContext` interface has to be implemented for a specific transformation technology, e. g., an `XtendTransformationContext` by using Xtend. The interface demands only an `execute` method which is responsible for executing a transformation step specified by a `TransformationDescriptor` and for storing possible results.

To use the benefits of KiVi we created a `TransformationEffect`, which is executed by KiVi and takes the aforementioned context and description. This allows the seamless integration into the KiVi: Further effects can be attached, e. g., a `HighlightEffect` to highlight the changed model elements.

```
1  public class SampleCombination extends AbstractCombination {
2
3      private static final String BUTTON = "de.cau.cs.kieler.button";
4      private ITransformationContext context;
5
6      public SampleCombination() {
7          KiviMenuContributionService.INSTANCE.addToolbarButton(this, BUTTON,
8          "Click me");
9          context = new XtendTransformationContext();
10     }
11
12     public void execute(ButtonState butState) {
13         if (butState.getButtonId() == BUTTON) {
14             // create descriptor
15             TransformationEffect effect =
16                             new TransformationEffect(context, descriptor);
17             schedule(effect);
18         }
19     }
20 }
```

Listing 1: A `Combination` executing a `TransformationEffect`

A specific transformation can be implemented by providing a KiVi `Combination`, which reacts to certain `Triggers`, e. g., a `ButtonTrigger` indicating a button click. Figure 5 illustrates that a `Combination` uses an unchanging context, which contains all information necessary to execute a transformation rule. The descriptor changes from step to step as different transformation rules are applied to different model elements.

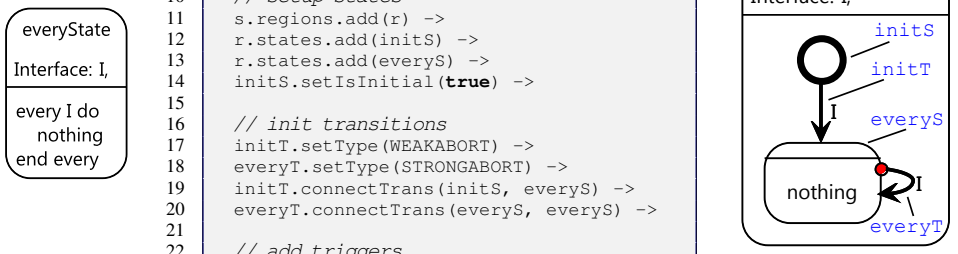
Listing 1 presents a general example of a `Combination`'s implementation. In line 7 a button labeled `Click me` is contributed to KiVi's user interface and in line 9 an `XtendTransformationContext` is created. If the contributed button is pressed by the user, the `execute` method specified in line 12ff will be called. A `TransformationEffect` is constructed with `context` and `descriptor`. The effect is scheduled for the execution in line 17.

3.1 Case Study: Interactive Transformation from Esterel to SyncCharts

To evaluate our approach, we implemented two different transformations. First, a transformation that synthesizes a SyncChart out of an Esterel program. The theory behind this transformation was presented by Prochnow et al. [PTvH06]. Additionally, they presented a transformation that optimizes the structure of a SyncChart, e. g., by removing needless states.

As mentioned in Section 2.1, some considerations are necessary prior to implementing each of the two transformations. The next two sections present these considerations as well as the implementation of an exemplary transformation rule for both, the Esterel to SyncCharts transformation and the SyncCharts optimization.

The Statecharts formalism, proposed by David Harel, is a well known approach for modeling control-intensive tasks. Statecharts extend Mealy machines with hierarchy, par-




```
1  Void rule(State s, EveryDo e):
2    let r = new Region:
3    let initS = new State:
4    let everyS = new State:
5    let initT = new Transition:
6    let everyT = new Transition:
7
8    initializeRule(s, e) ->
9
10   // setup states
11   s.regions.add(r) ->
12   r.states.add(initS) ->
13   r.states.add(everyS) ->
14   initS.setIsInitial(true) ->
15
16   // init transitions
17   initT.setType(WEAKABORT) ->
18   everyT.setType(STRONGABORT) ->
19   initT.connectTrans(initS, everyS) ->
20   everyT.connectTrans(everyS, everyS) ->
21
22   // add triggers
23   initT.addTriggerToTrans(e.delay) ->
24   everyT.addTriggerToTrans(e.delay) ->
25
26   // recursive
27   finalizeRule(everyS, e.statement)
28 ;
```

(a) Input (b) Xtend transformation rule (c) Transformed

Figure 6: Transformation of the `every` statement

allelism, signal broadcast, and compound events. SyncCharts [And96] are the natural adoption of Statecharts to the synchronous world. The imperative, textual language Esterel [Ber99] is based on the same synchronous model of time as SyncCharts. An example of an Esterel program and a SyncChart is given in Figure 6a and Figure 6c, respectively.

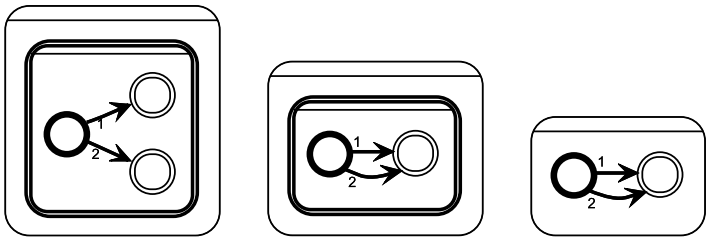
3.1.1 Transformation from Esterel To SyncCharts

The Esterel to SyncCharts transformation presented by Prochnow et al. [PTvH06] uses atomic transformation rules to convert each available Esterel statement into an equivalent SyncCharts state. This atomic characteristic can be leveraged to specify the granularity of a *step*. Therefore, the smallest step that a user can perform is the transformation of one single Esterel statement.

As the source (Esterel) and the target (SyncCharts) M2 meta model differ, this transformation can be classified as exogenous. Hence, a new notation has to be introduced. SyncCharts offer so-called *textual macro states*, which are states containing arbitrary text. We use these as container for the Esterel code. It does not require any changes to the meta model, seems natural, and sustains a good overview. See Figure 6a for an example. Other notations are possible, for instance, using a placeholder in a SyncCharts macrostate saying that the state still contains Esterel elements.

```
1  Boolean ruleXapplies(State s):
2     !s.isSimpleState() && hasMultipleSimpleFinalSubStates(s) ? true : false
3  ;
4
5  Void ruleX(State s):
6     let finals = s.collectSimpleFinalSubstates():
7     let first = finals.first():
8     finals.withoutFirst().bendIncomingTransitionsTo(first) ->
9     finals.withoutFirst().removeState()
10 ;
```

Listing 2: Example optimization rule removing multiple simple, final states



(a) Input (b) First Step (c) Second Step

Figure 7: Optimization of a SyncChart

Figure 6 shows the Xtend implementation of the transformation of the Esterel statement `every`. The input shown in Figure 6a is transformed by the transformation rule in Figure 6b into the result annotated with references to the Xtend elements shown in Figure 6c. In lines 2–6 of the transformation rule, all necessary new SyncCharts elements are created, e. g., `initS` is the newly created initial state, see the annotation in Figure 6c. The new elements are configured and the transitions are connected to their states in lines 11–24. The `inititalizeRule` call, seen in line 8, is responsible for removing the Esterel code and for naming the SyncCharts state according to the Esterel statement. Line 27 shows the `finalizeRule` which makes sure that further Esterel statements are processed correctly, in this case the `nothing` statement.

3.2 Optimization of the SyncChart

The optimization of SyncCharts was presented by Prochnow et al. [PTvH06] as a post processing of the Esterel to SyncCharts transformation, which initially creates several superfluous SyncCharts elements, e. g., additional hierarchy levels. As can be seen in Figure 2, the optimized result removes hierarchy levels that made the SyncCharts harder to understand than necessary. The optimization does not change any semantics but refines the structure of a SyncChart. Hence, it can also be applied independently of the Esterel to SyncCharts transformation.

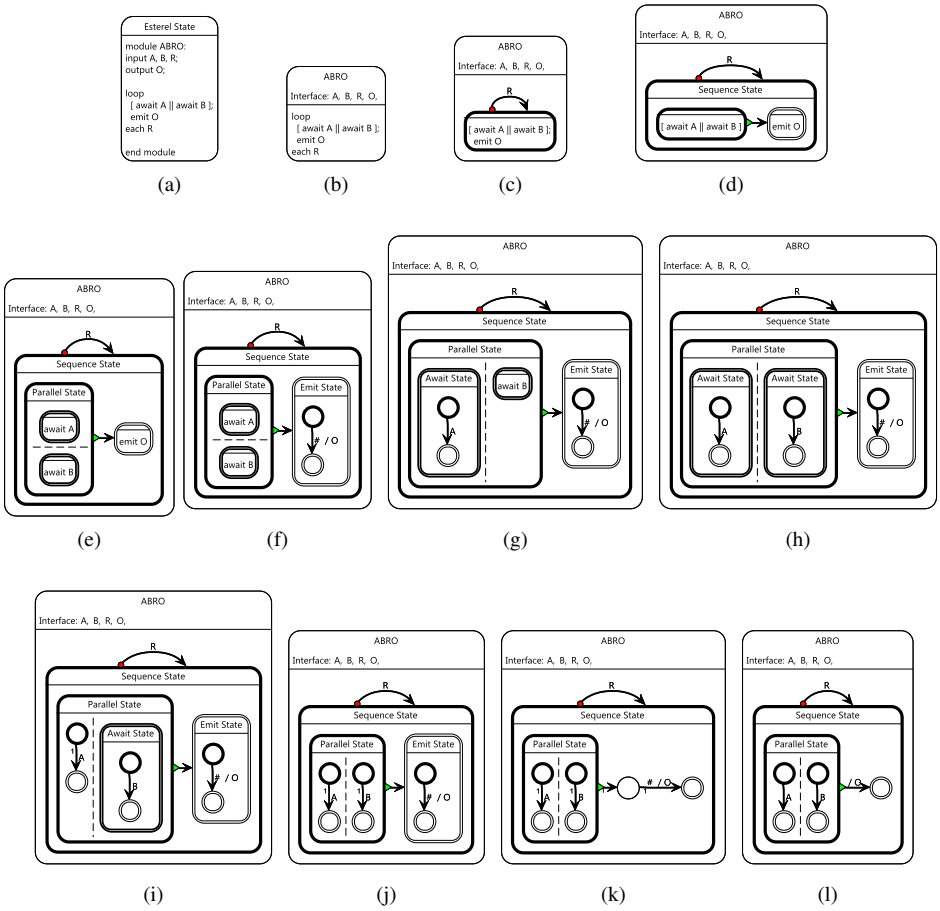


Figure 8: Step-wise transformation and optimization of the ABRO program

Like the transformation from Esterel to SyncCharts, the optimization of SyncCharts is defined by a collection of transformation rules, also called *optimization rules*. Each rule can be applied to a SyncCharts state which meets certain criteria. To illustrate, the inner macro state in Figure 7a contains two simple, final states, one of which is not necessary. Hence, the optimization rule connects the transitions to one of the final states and removes the other final state. In Listing 2, lines 5–10, a possible Xtend implementation realizing this optimization is presented. The criterion to apply this optimization rule is the existence of a macro state that contains more than one simple, final state. We defined this as a *predicate*, seen in lines 1–3 of Listing 2.

The visualization of intermediate steps is straightforward. Both the source and the target model base on the SyncCharts meta model. Hence, it is an endogenous in-place transfor-

mation and no new notation has to be introduced. We specify the execution of a single optimization rule as a step and present the modified and automatically layouted SyncChart as intermediate step.

Figure 7 shows the successive execution of two optimization rules. First, the previously mentioned rule removing multiple simple, final states is applied. The intermediate result is shown in Figure 7b. Second, the superfluous inner macro state is removed yielding the SyncChart presented in Figure 7c. A further optimization would remove the transition with the higher transition number as it is never taken.

3.2.1 Example: Transforming ABRO

Figure 8 shows the transformation of the ABRO program, the *Hello World* of the synchronous world, including all intermediate steps. Steps (a)–(h) transform Esterel statements into equivalent SyncCharts elements. Step (i)–(l) optimize the SyncChart's structure by removing unnecessary hierarchy levels, e. g., (j)–(k), or by combining transitions, e. g., (k)–(l).

The order of the transformation steps is not necessarily fixed. It can be changed by the user by selecting certain states interactively. Starting from step (e) not only (f) can be the next intermediate result but also (g). This can be achieved by providing further information to the transformation logic. In our implementation this information is the selection of the `Parallel State` of (g).

Additionally, the order of the transformation steps can be changed by the user later on. Figure 1 shows the UI that allows the user to go steps backwards and to select another continuation point to proceed the transformation.

Comparing (h) and (l), the importance of the SyncCharts optimization becomes clear as (l) is smaller and easier to understand.

The example shows that the step-wise transformation reveals more information than the direct transformation from (a) to (l). User interaction can take place between arbitrary transformation steps allowing to change the order of the transformation steps or to perform steps backwards. Thus, the user's comprehensibility is improved and he can focus on steps he is interested in.

4 A View Management Approach to Structure-Based Editing

As another practicability evaluation we will use *structure-based editing* which is presented by Fuhrmann et al. [FSMvH10] as a means to simplify the manual editing process of graphical models.

The main concept of structure-based editing is to provide predefined model transformations for common editing tasks, see Figure 9 for three examples. The first transformation adds a successor state `S1` to the existent state `S`. The other two transformations make `S1` a macro state and add a region to the state `S1`. With traditional editing paradigms the user

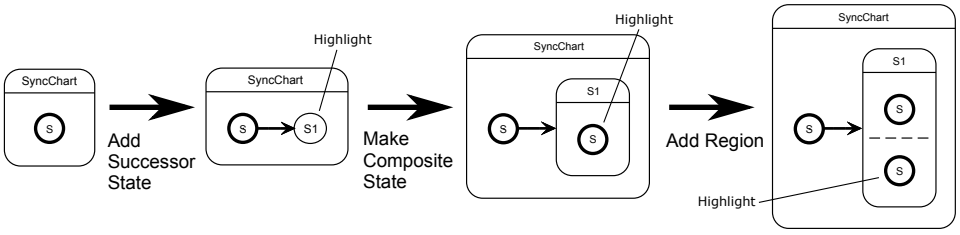


Figure 9: Possible editing proceeding using KSBasE

would have to enlarge the macro state, add a new state, and connect the two states with a new transition. With structure-based editing the user just selects the `S` state and executes the predefined transformation by using a context menu or a keyboard shortcut.

In the KSBasE[4] framework developed by Matzen this approach is integrated into the KIELER environment and already provides a convenient way to add new predefined model transformations for arbitrary meta models. The approach presented in this paper, leveraging view management concepts, now allows to further interact with modified or created model elements during runtime. We re-implemented the structure transformations seen in Figure 9 as a proof of concept, by using KiVi and the ideas presented above. This allows to apply further visualizations to created or modified model elements. E. g., Figure 9 indicates the highlighting of newly created states. Predefined model transformations can be contributed to the Eclipse user interface via a `Combination` and be executed as a `TransformationEffect`. Furthermore, it is possible to apply visualizing effects upon an executed transformation by using a `Combination`.

5 Conclusions and Outlook

In this paper we presented an approach which enriches model transformations by interactivity and visualization. This aims primarily at improving comprehensibility for both users and developers. Our approach allows user interaction during model transformations and visualization of intermediate steps of such transformations. The seamless integration into KiVi leaves opportunities for extensions for interaction as well as for visualization. As an illustrating case study we presented the Esterel to SyncCharts transformation and the SyncCharts optimization. We showed how the concepts can be applied to other existing use cases where model transformations are used, such as the structure-based editing.

We plan to perform further evaluations, including usability tests, of the presented use cases and further domains. Concerning KiVi we aim to improve the possibilities to contribute to the user interface and implement a framework which allows the user to define `Combinations` during runtime, e. g., by using a scripting language. Likewise it would be possible to define new interactive model transformations on-the-fly.

[4] http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/KSBasE

References

[And96] Charles André. SyncCharts: A Visual Representation of Reactive Behaviors. Technical Report RR 95–52, rev. RR 96–56, I3S, Sophia-Antipolis, France, Rev. April 1996.

[BEK+06] Enrico Biermann, Karsten Ehrig, Christian Köhler, Gnther Kuhns, Gabriele Taentzer, and Eduard Weiss. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In *Model Driven Engineering Languages and Systems (MoDELS'06), LNCS*, volume 4199/2006, pages 425–439. Springer Berlin/Heidelberg, 2006.

[Ber99] Gérard Berry. *The Esterel v5 Language Primer*, 1999. ftp://ftp-sop.inria.fr/meije/esterel/papers/primer.ps.

[FSMvH10] Hauke Fuhrmann, Miro Spönemann, Michael Matzen, and Reinhard von Hanxleden. Automatic Layout and Structure-Based Editing of UML Diagrams. In *Proceedings of the 1st Workshop on Model Based Engineering for Embedded Systems Design (M-BED 2010)*, Dresden, March 2010.

[FvH10] Hauke Fuhrmann and Reinhard von Hanxleden. Taming Graphical Modeling. In *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, volume 6394 of *LNCS*, pages 196–210. Springer, October 2010.

[GW06] Holger Giese and Robert Wagner. Incremental Model Synchronization with Triple Graph Grammars. In *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06)*, pages 543–557. Springer Verlag, 2006.

[KPRP07] Dimitrios S. Kolovos, Richard F. Paige, Louis M. Rose, and Fiona A.C. Polack. Bridging the Epsilon Wizard Language and the Eclipse Graphical Modeling Framework. In *Modeling Symposium, Eclipse Summit Europe*, 2007.

[LS11] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. Lulu, 2011.

[MFvH10] Christian Motika, Hauke Fuhrmann, and Reinhard von Hanxleden. Semantics and Execution of Domain Specific Models. In *2nd Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2010) at conference INFORMATIK 2010*, GI-Edition – Lecture Notes in Informatics (LNI), Leipzig, Germany, September 2010. Bonner Köllen Verlag.

[MG06] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125 – 142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).

[PTvH06] Steffen Prochnow, Claus Traulsen, and Reinhard von Hanxleden. Synthesizing Safe State Machines from Esterel. In *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, Ottawa, Canada, June 2006.

[Sch11] Christian Schneider. On Integrating Graphical and Textual Modeling. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, February 2011.

[VAB+07] Bert Vanhooff, Dhouha Ayed, Stefan Van Baelen, Wouter Joosen, and E Berbers. Uniti: A unified transformation infrastructure. In *In ACM/IEEE 10th International Conference on Model-Driven Engineering Languages and Systems (MoDELS 2007*, 2007.