Refactoring of Simulink Diagrams via Composition of Transformation Steps

Quang Minh Tran, Benjamin Wilmes
Berlin Institute of Technology
Daimler Center for Automotive IT Innovations (DCAITI)
Berlin, Germany
E-Mail: {quang.tranminh,benjamin.wilmes}@dcaiti.com

Christian Dziobek
Daimler AG
Mercedes-Benz Cars Development
Sindelfingen, Germany
E-Mail: christian.dziobek@daimler.com

***Abstract*—Model-based design has been increasingly adopted by the industry, especially the automotive industry, for the development of embedded software. Today, Matlab/Simulink by The MathWorks is widely employed as a modeling tool in which embedded software is modeled as data flow diagrams consisting of blocks and signals. While refactoring has become an established technique for improving the structure of code in textual programming languages, refactoring Simulink diagrams is relatively unexplored. This paper introduces a technique for specifying and implementing refactoring operations for Simulink diagrams by composing elementary and composite transformation steps. How the transformation steps can be leveraged to specify and implement complex refactoring operations is demonstrated based on the two refactoring examples *Replace Goto/From With Explicit Signals* and *Merge Subsystems*. Our prototypical implementation of a refactoring extension for Simulink is also briefly described.**

Keywords-Simulink; Refactoring; Transformation

I. INTRODUCTION

The model-based design (MBD) paradigm has been widely adopted by the automotive industry to develop embedded software, with Matlab/Simulink [1] by The MathWorks being the defacto standard modeling tool. Using Simulink, software functionality is modeled as data flow diagrams by connecting functional blocks via data-carrying signals. Additional concepts of the Simulink modeling language address practical needs, like the readability of large models. For instance, model fragments can be hierarchically grouped into logical units called subsystems and related signals can be grouped into structured bus signals. The adoption of MBD using Simulink leads to models being central artifacts in development. Due to the continuously increasing software complexity and short development cycles, the creation and maintenance of models have become highly intensive and time-consuming activities.

Refactoring is an established restructuring technique which implies changing the structure of a development artifact without changing its observable behavior. Semi-automated or interactive refactoring operations have been integrated into textual programming environments like Eclipse or Visual Studio. However, at present, refactoring is practically non-existent in the Simulink Editor. The missing support for refactoring in Simulink has two potentially severe consequences. First, the model quality may be compromised if quality-improving model changes are not done due to tight development time, even if the modeler is aware of the structural deficits. Second, refactoring a huge Simulink model manually can be very labor-intensive and error-prone.

Thus, in this paper, we present a modular technique for refactoring Simulink diagrams based on the composition of predefined transformation steps. While the focus of this paper is on the underlying refactoring mechanism, we refer to a previous publication of ours for a wider spectrum of useful refactoring operations for Simulink diagrams [2].

The paper is structured as follows. In Section II, we present our meta-model for Simulink models, which serves as the basis for defining transformation steps. Our mechanism for composing transformation steps is described in Section III. How even complex refactoring operations can be specified and realized by utilizing primitive but powerful transformation steps is shown in Section IV. Insight into our prototypical implementation of the concept as an extension of the Simulink Editor is provided in Section V, followed by a summary of related work in Section VI, and our conclusion in Section VII.

II. SIMULINK META-MODEL FOR REFACTORING

The development of a refactoring technique for Simulink diagrams inevitably requires the existence of a meta-model. Unfortunately, to date, no official meta-model for Simulink diagrams has been published. Hence, we defined our own Simulink meta-model which, for the purpose of refactoring, implicitly meets the following criteria:

1) All necessary structural properties of diagrams that are required by refactorings should be captured, including model hierarchy, signal properties and bus structure
2) Support for incomplete diagrams, such as those with unconnected signals, since some refactorings can be triggered at any time during the modeling
3) Layout information must be captured because the execution of a refactoring operation should preserve the layout as much as possible
4) Establish a degree of granularity that enables local structural changes during a refactoring operation without affecting irrelevant model parts

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9

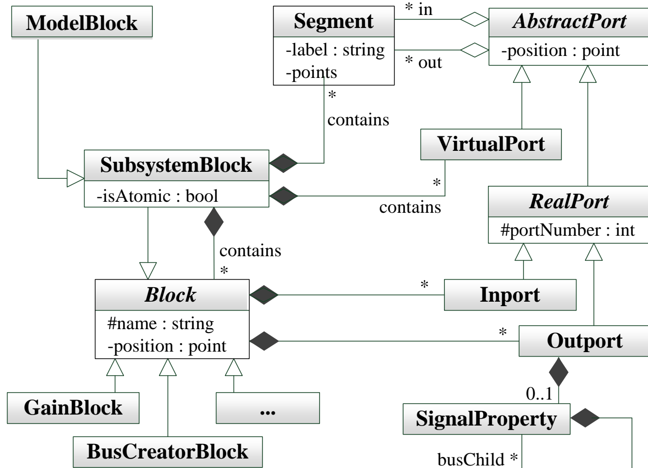

Figure 1. Excerpt of our meta-model for Simulink diagrams

Category	Transformation Steps
Elementary	addBlock(blockType, [pos]) addInportBlock(destSubsys,[pos]) addOutportBlock(destSubsys,[pos]) copyBlock(block, destSubsys, [pos]) replaceBlock(block, newBlockType) deleteBlock(block) **addSegment(srcPort, targetPort)** rerouteSegmentToNewTargetPort(seg, newTargetPort) rerouteSegmentToNewSourcePort(seg, newSourcePort) branchSegmentToNewTargetPort(seg, newTargetPort)
Composite	moveBlocks(blocks, destSubsys, [pos]) deleteBlockWithSignals(block) **addCrossHierarchicalSignal (sourcePort, targetPorts)** rerouteSegmentCrossHierarchicallyToNewTargetPort (sourcePort, newTargetPort) branchSegmentCrossHierarchicallyToNewTargetPort (seg, newTargetPort)

Table I
EXCERPT OF THE TRANSFORMATION STEPS COLLECTION WITH THE STEPS BEING DISCUSSED IN MORE DETAIL MARKED BOLD

Figure 1 shows an excerpt of our meta-model as class diagram. In this meta-model, *Block* is the superclass for all other block types. A *Block* has a unique name in its hierarchical scope and an ordered list of *Inport* and *Outport* instances. Its position is stored in the field *position*. A *Subsystem* is a block that can contain child blocks including other subsystems. An entire model is also a *Subsystem*.

Regardless of whether a signal is completely connected, i.e., constituting an uninterrupted path from one source block to one or more destination blocks, it is divided into one or more segments. A segment connects a source and a target port - which can be of the following types: A *real port* belongs to a block and is either an *inport* (for an incoming signal) or *outport* (for an outgoing signal). A *virtual port* is either a branching point of a signal or an end point of an incompletely connected signal. Both real ports and virtual ports have an (x, y) position. In contrast to a virtual port, a real port has a port number. A segment is called unconnected if its source is a virtual port but not a branching point, or its target is a virtual port.

III. Transformation Steps and Their Composition

Instead of formulating each refactoring operation individually, we have set the goal to define basic transformation and modification steps that can be aggregated for specifying and implementing complex refactoring operations. As a result, on top of the meta-model in Section II, we have identified a collection of transformation steps (see Table I). A transformation step modifies an instance of the meta-model, i.e., a Simulink model. While defining the steps, we had to address the following key questions:

How powerful in terms of the effect should a transformation step be?

The use of a powerful transformation step reduces the complexity of a refactoring specification but is more difficult to reuse. For instance, there are two possible ways to define the transformation step *deleteBlock* that deletes a block. One way is, if a block is removed, its incoming and outgoing segments remain and become unconnected segments. A more powerful version of *deleteBlock* would also remove the incoming and outgoing segments. The former is especially useful if after the deletion, the references to the now unconnected segments are still needed - if, for instance, the segments are rerouted to other blocks in a following transformation step. For the sake of reusability, we have decided to keep basic transformation steps as granular as possible. If necessary, more powerful versions are defined by composing more fine-grained steps, such as *deleteBlockWithSignals*, which is realized by using *deleteBlock* and then deleting the incoming and outgoing segments using *deleteSegment*.

How can transformation steps be composed to define more complex transformation steps?

We distinguish between *elementary* and *composite* transformation steps. An elementary step modifies an instance of the meta-model without using other transformation steps, while a composite step consists of an ordered list of (possibly elementary or composite) child steps. Performing an elementary step directly changes an instance of the meta-model. A composite step can be performed by executing each step in the list in the specified order. Back to the previous example, *deleteBlock* is an elementary step while *deleteBlockWithSignals* is a composite step.

How should a step affect the layout?

Layouting of Simulink models ultimately addresses the positioning of blocks and signals in the Simulink Editor. Since the layout plays a crucial role for the readability of a Simulink diagram and layouting thus needs to be considered by refactoring operations, transformation steps can receive

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9

```
Precondition: sameSubsystem(srcPort,targetPort)
              ∧¬ targetPort.hasIncomingSegment
1: function ADDSEGMENT(srcPort,targetPort, [name])
2: p ← srcPort.containingSubsystem
3: newSeg ← new Segment(name)
4: newSeg.source ← srcPort
5: newSeg.target ← targetPort
6: srcPort.outSegs ← srcPort.outSegs ∪ {newSeg}
7: targetPort.inSeg ← newSeg
8: p.childSegments ← p.childSegments ∪ {newSeg}
9: end function
```

Figure 2. Algorithm for elementary transformation step *addSegment*

layout information from parent composite transformation steps. If layout information is not provided, predefined layout heuristics or Simulink itself determine the layout. For an improved layout after refactoring, the automatic layouting algorithm for Simulink diagrams [3] can be used.

Based on these basic principles, we show by examples how elementary and composite steps modify an instance of the meta-model. Figure 2 depicts (informal) pseudo code for the elementary step *addSegment* that adds a new segment from a source port *srcPort* to a target port *targetPort*. It presumes that *srcPort* and *targetPort* are in the same subsystem and *targetPort* does not have an incoming segment. If these conditions are satisfied, a new segment *newSeg* is created. A name is given depending on the type of refactoring in which this step is used. The segment's start and end ports are set to *srcPort* and *targetPort*, respectively. Additionally, the new segment is added to the collection of outgoing segments of *srcPort* and assigned to *targetPort* as the incoming segment. Finally, *newSeg* is added to the current subsystem.

Figure 3 shows the algorithm for *addCrossHierarchicalSignal* which adds (possibly cross-hierarchical) signals from a source port to one or several target ports. Unlike *addSegment*, *addCrossHierarchicalSignal* is a composite transformation step because it makes use of other transformation steps such as *addSegment*, *addOutportBlock* and *addInportBlock*. Note that the algorithm contains control structures, as well as other commands, and is not purely a list of transformation steps as indicated before. Due to space limitations, we abstract from implementation details here. As mentioned, the steps' algorithms as described in this paper are executed on an instance of the meta-model. Each call of an elementary transformation step is registered in an ordered step list which is then executed step by step on the real Simulink model.

The precondition of *addCrossHierarchicalSignal* states that the list of target ports must have at least one element and all target ports must be in the same subsystem. If a precondition is not satisfied, the entire refactoring in which this step is used will not be applied. If satisfied, it determines the subsystem where a forward constructed signal from source to target and a backwards constructed signal from target to source would meet (least common subsystem). Then, the signal is forwarded from the source

```
Precondition: sameSubsystem(targetPorts) ∧ targetPorts ≠ ∅
              ∧ ∀ tp ∈ targetPorts: ¬tp.hasIncomingSegment
 1: function ADDCROSSHIERARCHICALSIGNAL
                                        (srcPort,targetPorts)
 2: leastSub ← getLeastCommonSubsystem(srcPort,targetPorts)
 3: curPort ← srcPort
 4: curSubsys ← curPort.containingSubsystem
 5: while curSubsys ≠ leastSub do
 6:     outBlock = addOutportBlock(curSubsys)
 7:     ret = addSegment(curPort,outBlock.inport)
 8:     curPort ← curSubsys.outportOf(outBlock)
 9:     curSubsys ← curPort.containingSubsystem
10: end while
11: subsysPath ← getSubsystemPath(targetPorts(1),leastSub)
12: for p ∈ sortByHierarchyTopDown(subsysPath) do
13:     inBlock = addInportBlock(p)
14:     addSegment(curPort,p.inportOf(inBlock))
15:     curPort ← inBlock.outport
16: end for
17: for tp ∈ targetPorts do
18:     addSegment(curPort,tp)
19: end for
20: end function
```

Figure 3. Algorithm for composite transformation step *addCrossHierarchicalSignal*

port up to the ancestor and from there down to the subsystem containing the target ports by creating outport blocks, inport blocks and signals for the intermediate subsystems. Finally, in the subsystem containing the target ports, branching signals are added from the newly added inport block to the target blocks. Note that *addCrossHierarchicalSignal* avoids redundant blocks and signals by creating a single signal path from the source port to the parent subsystem of the target ports before branching it to the target ports.

IV. SPECIFICATION OF REFACTORINGS

The transformation steps can be leveraged to formulate refactoring operations, as shown next using two examples: (1) *Replace Goto/From With Explicit Signals* creates explicit (possibly cross-hierarchical) signals from the source *Goto* block to all associated *From* blocks, (2) *Merge Subsystems* merges two subsystems into a single subsystem. These two refactorings are part of our Simulink refactoring catalog [2].

A. Replace Goto/From With Explicit Signals

Motivation: An advantage of data flow diagrams such as Simulink is that the data flow between blocks is explicit thanks to visual signal connections. However, Simulink provides *Goto/From* blocks as a means to define implicit, non-visual signal connections between blocks that may reside on different model levels - usually to reduce the visual complexity. Similar to the *Goto* construct in imperative programming languages, the use of *Goto/From* blocks, especially of global scope, may dramatically reduce the understandability of the model because tracing the data flow becomes more difficult. *Goto/From* blocks can be replaced by explicit signal connections without changing the behavior of the model. This can be a tedious task when done manually.

Copyright (c) IARIA, 2013.

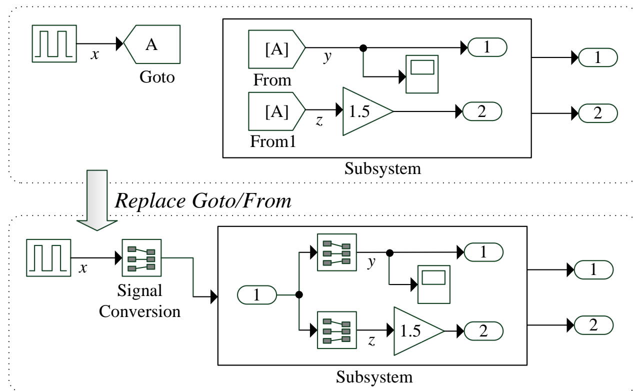

Figure 4. Example for refactoring *Replace Goto/From with Explicit Signals*

Precondition: -

```
 1: function REPLACEGOTOFROMWITHEXPLICITSIGNALS
                                              (gotoBlock)
 2: fromBlocks ← gotoBlock.fromBlocks
 3: inSeg ← gotoBlock.inSeg
 4: subsystemList ← buildSubsystemList(fromBlocks)
 5: gotoConverter ← replaceBlock
                          (gotoBlock,'SignalConversion')
 6: curOutport ← gotoConverter.outport
 7: for s ∈ sortByHierarchyTopDown(subsystemList) do
 8:     targetPorts ← ∅
 9:     for fromBlocks ∈ s do
10:         for fromBlock ∈ fromBlocks do
11:             fromConverter ← replaceBlock
                          (fromBlock,'SignalConversion')
12:             targetPorts ← targetPorts ∪ fromConverter.inport
13:         end for
14:     end for
15:     targetRootPort ← addCrossHierarchicalSignal
                              (curOutport,targetPorts)
16:     curOutport ← targetRootPort
17: end for
18: end function
```

Figure 6. Algorithm for refactoring *Replace Goto/From With Explicit Signals*

Mechanics: Figure 6 shows (informal) pseudo code for *Replace Goto/From with Explicit Signal*. Take a look at Figure 4 for an example.

As first operation, the function *buildSubsystemList* is called to obtain a list of the corresponding *From* blocks' parent subsystems. This list is used for signal forwarding while avoiding redundant signal paths at the same time. Then, the transformation step *replaceBlock* is used to replace the *Goto* block with a *Signal Conversion* block. *Signal Conversion* blocks are used here solely for preserving signal names. More specifically, if the incoming signal of the *Goto* block has a different name than the signal names leaving the *From* blocks, the use of *Signal Conversion* blocks would allow these names to continue to exist after the refactoring.

Next in the algorithm, *subsystemList* is iterated. In each iteration, the *From* blocks within the current subsystem are replaced by *Signal Conversion* blocks. Finally, the composite transformation step *addCrossHierarchicalSignal* is used to foward signals to the *Signal Conversion* blocks.

B. Merge Subsystems

Motivation: During creation and maintenance of a Simulink model, reorganizing activities are frequent. In particular, it is often necessary to combine functionalities residing in separate subsystems into a single subsystem. With the current modeling support of the Simulink Editor, the modeler would have to cut and paste the content of one subsystem into the other subsystem. Then, the signals must be reconnected to re-establish the initial signal relationships. If lots of signals must be connected manually, this activity becomes both labor-intensive and error-prone.

Mechanics: While Figure 5 provides an example of this refactoring, Figure 7 shows (informal) pseudo code for merging two subsystems A and B. The precondition specifies that A and B must be non-atomic (virtual) subsystems. This restriction exists since merging atomic subsystems may change the behavior of the model.

In essence, the algorithm uses suitable transformation steps to move the content of B to A (line 18), adjust the signal connections (line 23, 26, 32, and 35) and finally delete B (line 37). Before the actual transformation, some book keeping needs to be done. In particular, *inSegsOfB* contains all incoming segments of B. Hash tables *inMap* and *outMap* are used to keep track of the references between inport and outport blocks of B to the source ports of the signals reaching them for reconnecting signals.

For *inMap*, if an inport block *inpBlock* of B has an incoming segment entering B at the inport corresponding to *inpBlock*, we store the mapping between *inpBlock* and the *source port* of that incoming segment returned by *getSrcPort*. In this context, the source port is returned by *getSrcPort*, which checks if the root source of the segment is an outport of A. If yes, we go into A and retrieve the source of the signal within A. Otherwise, the source port is the root source of the segment and is located on the common parent subsystem of A and B. For *outMap*, if an outport block *outBlock* of B has an outgoing segment leaving B at the outport corresponding to *outBlock*, we store the mapping between *outBlock* and the segment.

For transformation, *moveBlocks(B.content,A)* moves B's content to A. *inMap* is used to reestablish incoming signal connections to the blocks that used to be in B. In particular, for each inport block *inpBlock* stored in *inMap*, if the source port of *inpBlock* is in A, *inpBlock* is replaced by a *Signal Conversion* block before connecting the source port to the inport of that *Signal Conversion* block. Otherwise, the source port is connected to the inport of A corresponding to *inpBlock*. Then, using *outMap*, the initial outgoing segments of B are rerouted to the outports of A corresponding with the outport blocks being moved from B. Finally, the initial incoming segments of B and B itself are removed.

It should be noted that the decision of which of the two subsystems to be merged serve the role of A and B in the

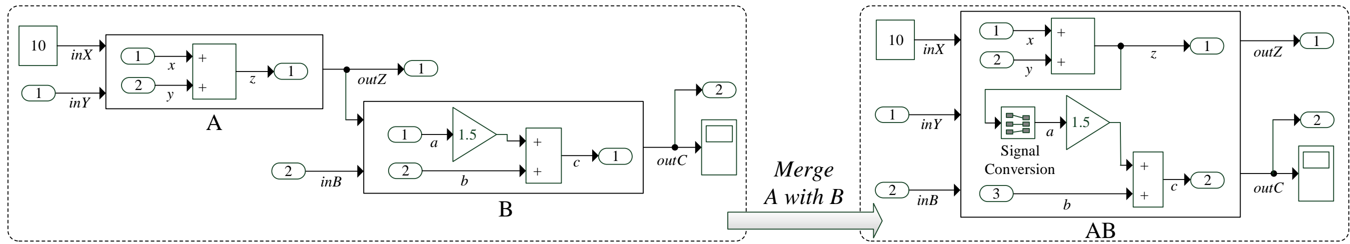

Figure 5. Example for refactoring *Merge Subsystems*

```
Precondition: ¬A.isAtomic ∧ ¬B.isAtomic
 1: function MERGESUBSYSTEMS(A, B)
 2: inSegsOfB ← B.inSegments
 3: inpBlocksInB ← B.inportBlocks
 4: for inpBlock ∈ inpBlocksInB do
 5:     inport ← B.inportOf(inpBlock)
 6:     if inport.hasIncomingSegment then
 7:         inMap.value(inpBlock) ← getSrcPort(inpBlock,A)
 8:     end if
 9: end for
10: outBlocksInB ← B.outportBlocks
11: for outBlock ∈ outBlocksInB do
12:     outport ← B.outportOf(outBlock)
13:     if outport.hasOutgoingSegment then
14:         outSeg ← outport.outSegment
15:         outMap.value(outBlock) ← outSeg
16:     end if
17: end for
18: moveBlocks(B.content, A)
19: for inpBlock ∈ inMap.keys do
20:     srcPort ← inMap(inpBlock)
21:     if srcPort.containingSubsystem = A then
22:         converter ← replaceBlock
                        (inpBlock,'Signal Conversion')
23:         addSegment(srcPort, converter.inport)
24:     else
25:         targetPort ← A.inportOf(inpBlock)
26:         addSegment(srcPort,targetPort)
27:     end if
28: end for
29: for outBlock ∈ outMap.keys do
30:     outport ← A.outportOf(outBlock)
31:     outSeg ← outMap.value(out)
32:     rerouteSegmentToNewSource(outSeg,outport)
33: end for
34: for seg ∈ inSegsOfB do
35:     deleteSegment(seg)
36: end for
37: deleteBlocks(B)
38: end function
```

Figure 7. Algorithm for refactoring *Merge Subsystems*

algorithm affects the port order of the inports and outports within the resulting merged subsystem. This is due to the way Simulink automatically assigns port numbers when a port is added or deleted. However, since the port order does not affect the behavior, the refactoring does not change the model behavior. We have also defined a refactoring operation called *Reorder Ports* that can be used to rearrange the port order of inports or outports of a subsystem. If required, this refactoring can be used to achieve the desired port order.

V. IMPLEMENTATION

We have implemented a prototype in Matlab's *m* language that integrates refactoring support directly into Simulink Editor. Specifically, a refactoring operation can be directly triggered in Simulink Editor via a menu item or shortcut. Based on the *Template* design pattern [4], the prototype implements a generic workflow of refactorings as a graphical wizard. The behavior of a specific refactoring operation such as the required graphical dialogs for user input and the (interactive) specification of transformation steps can be easily defined and integrated into the prototype.

The meta-model in Section II is implemented as Matlab classes. The elementary and composite transformation steps in Section III are provided in the form of Matlab functions. In addition, the prototype also contains a collection of Matlab functions for model analysis that are useful for refactoring purposes. For instance, the functions *getLeast-CommonSubsystem* used from Figure 3 and *buildSubsystem-List* from Figure 6 are stored in a special collection since they are needed by multiple refactorings. The functions for transformation steps and model analysis serve as a high-level and compact API for formulating refactoring operations.

The prototype also features a graphical preview that shows the list of transformation steps to be executed in a tree. Moreover, it shows the Simulink diagram before and after a refactoring operation.

We have tested our prototype on several industrial Simulink models from the automotive domain at Daimler. The biggest time factor turned out to be the time required to convert a Simulink model into an instance of the meta-model. In an extreme case, for a model of about 20,000 blocks and a refactoring operation that affects almost the entire model, the parsing time took roughly 10 minutes. For most models and operations, however, the parsing time was just a matter of seconds. The transformation itself usually took only seconds, or at most, a few minutes.

VI. RELATED WORK

In textual programming, refactoring has become a standard technique for restructuring code without changing its observable behavior [5], such as for object-oriented languages [5] and functional languages [6]. Modern Integrated Development Environments (IDEs) like Eclipse, NetBeans and Visual Studio offer built-in support for refactoring.

In model-based development, UML models have been targeted for refactoring support [7]. Refactoring of data flow

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9

diagrams such as Simulink, however, is only scantly researched. Sui et al. [8] propose an implementation approach for an automated refactoring tool aimed at visual dataflow programming languages. However, the focus of their paper is rather on the tool architecture aspect than on specifying refactoring operations modularly.

The current version of Simulink Editor does not provide refactoring support. Tools such as Model Advisor [9] or Model Examiner [10] can automatically detect violations of modeling guidelines and do provide, to a limited extent, so-called repair scripts for repairing guideline violations. Nonetheless, the focus of these tools is on automated detection of guideline violations and not on providing complex refactoring operations with possible user input or interaction.

The approach which is most related to our work has been developed in the MATE project [11]. It is an approach to visual specification and transformation for Simulink and Stateflow models based on graph transformation techniques. Specifically, modeling guideline violations and possible repair scripts are formulated in the graphical specification language called Story Driven Modeling (SDM). It turned out, however, that a purely visual specification language, such as SDM, is not powerful enough for complex real specification scenarios such as those including regular expressions, complex mathematical calculations and complex navigation through a network of linked objects.

VII. Conclusion and Future Work

In this paper, we have introduced our technique for specifying and implementing complex refactoring operations for Simulink diagrams based on the composition of transformation steps. The concept has been successfully implemented as a prototype that integrates refactoring support into Simulink Editor. Using the infrastructure provided by the prototype, we were able to implement many refactoring operations from our catalog [2] with little effort.

As the next step, we plan to extend our catalog and tool with further useful refactoring operations. Our future work will also address the automated identification of model constructs for which the application of certain refactorings is recommendable - so-called model smells, in analogy to code smells known from code refactoring [5]. There exist several techniques for Clone Detection in a Simulink diagram, as explored by Deissenboeck et al. [12] and Petersen [13], which could be used to identify similar or identical fragments in a Simulink diagram and suggest applicable refactoring operations for eliminating them. Moreover, we plan to evaluate the developed techniques and tool in real development environments at Daimler.

In addition, having automated transformation and refactoring techniques for Simulink models on hand, advanced applications are rendered possible. For instance, Simulink models could be automatically optimized by search-based algorithms using our transformation steps, as suggested for code [14], with respect to measurable model quality criteria, which already exist for Simulink diagrams [15].

References

[1] The MathWorks, "Matlab/Simulink," http://www.mathworks.de/products/simulink/ [Last access: 11/06/2013].

[2] Q. M. Tran and C. Dziobek, "An approach to design and maintenance of Simulink models by using transformations/refactorings and generation operations," in *Proceedings of the Model-Based Development of Embedded Systems Workshop (MBEES)*, 2013, pp. 1–12.

[3] L. Klauske and C. Dziobek, "Improving modeling usability: Automatic layouting for Simulink," in *Proceedings of the 2010 MathWorks Automotive Conference*, 2010, pp. 1–8.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[6] H. Li, "Refactoring Haskell programs," Ph.D. dissertation, University of Kent, 2006.

[7] G. Sunyé, D. Pollet, Y. L. Traon, and J.-M. Jézéquel, "Refactoring UML models," in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, 2001, pp. 134–148.

[8] Y. Y. Sui, J. Lin, and X. T. Zhang, "An automated refactoring tool for dataflow visual programming language," *SIGPLAN Notices*, vol. 43, no. 4, pp. 21–28, Apr. 2008.

[9] The MathWorks, "Model Advisor," http://www.mathworks.de/de/help/simulink/ug/consulting-the-model-advisor.html [Last access: 11/06/2013].

[10] Model Engineering Solutions, "Model Examiner," http://www.model-engineers.com/de/model-examiner.html [Last access: 11/06/2013].

[11] I. Stürmer, I. Kreuz, W. Schäfer, and A. Schürr, "The MATE approach: Enhanced Simulink and Stateflow model transformation," in *Proceedings of the 2007 MathWorks Automotive Conference*, 2007, pp. 1–9.

[12] F. Deissenboeck *et al.*, "Clone detection in automotive model-based development," in *Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 603–612.

[13] H. Petersen, "Clone detection in Matlab Simulink models," Master's thesis, Technical University of Denmark, DTU Informatics, 2012.

[14] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–11:61, 2012.

[15] J. Scheible and H. Pohlheim, "Automated model quality rating of embedded systems," in *Proceedings of the 4th SQMB Workshop*, 2011, pp. 1–10.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9