Rule-Based Model Transformation For, and In Simulink

Joachim Denil[1] , Pieter J. Mosterman[1,3], Hans Vangheluwe[2,1]

[1]**MSDL**
School Of Computer Science
McGill University
Montréal, Canada
{denil,hv}@cs.mcgill.ca

[2]**ANSYMO**
University of Antwerp
2020 Antwerp, Belgium
Hans.Vangheluwe@uantwerp.be

[3]**MathWorks**
3 Apple Hill Drive
Nattick, MA, USA
Pieter.Mosterman@mathworks.com

Keywords: Model-Transformation, Simulink, Model-Based Design, Model-Driven Engineering

Abstract

Over the past decade, the design of embedded systems has come to rely on models as electronic artifacts that are both analysable and executable. Such executable models are at the core of Model-Based Design. Simulink® is a popular Model-Based Design tool that supports simulation of models in various stages of design. While Simulink supports relating the various different models used in design, the technology to do so relies on the underlying Simulink code base. Instead, this paper employs explicit models of the relations between the various different design models. In particular, a rule-based approach is presented for model-to-model transformations. The abstraction from the code base provides benefits such as a more intuitive representation and the ability to more effectively reason about the transformations. The transformation rules and schedules are designed by augmenting standard Simulink model elements (e.g., blocks) for use in model transformation based on the structured *RAMification* approach. The approach is illustrated by the transformation of a continuous-time model, part of an adaptive controller, to a disrete-time counterpart, which is consecutively optimized for simulation.

1. INTRODUCTION

Today, MATLAB® and Simulink® are popular tools used in the design, simulation, and verification of software-intensive and cyber-physical systems. Thanks to extensive automatic code generation facilities, Simulink can be employed as a visual programming language that is based on the causal block diagram formalism. Centered around such block diagram models, Simulink supports the use of Model-Based Design (MBD) for the creation of embedded systems. MBD uses models as computational representations of the system under design. This computational representation enables executable specifications, simulation for design space exploration, automatic code generation, and continuous test and verification.

Generally, embedded system design comprises a set of models that constitute a variety of different representations of the system under design. As such, transformations between the various models are important and valuable operations. While Simulink supports relating and transforming models, the technology to do so is encoded in the underlying Simulink code base. To complement the efficiency of such an implementation, model transformation technologies allow a declarative ("what" instead of "how") specification of the transformations in the problem domain (of causal block diagrams) rather than in the solution domain (of code). Thus, engineers can more intuitively specify operations on a model, for example for optimization, normalization, and composition, especially by keeping the concrete syntax for the specification of rules similar to that of Simulink.

This paper augments the Simulink environment with rule-based model transformation capabilities. Model transformation control flow as well as transformation rules are specified using the same concrete syntax as the blocks available in Simulink libraries. To this end, new libraries are created in a structured fashion using a process called RAMification [1].

The paper is organized as follows: Section 2. introduces the process of RAMification. Section 3. applies this process to the Simulink block library. Section 4. operationalizes the transformation models. In Section 5., the contribution of the paper is illustrated by the discretization and optimization of a model before simulation. Section 6. then discusses related work. Finally, Section 7. concludes and presents future work.

2. INTRODUCTION TO RAMIFICATION

Metamodeling is used to specify the abstract (and concrete) syntax of modeling languages. It has become popular thanks to a number of advantages: (1) the specification of a language is explicit rather than hidden in the code of a tool, making it easier to under-

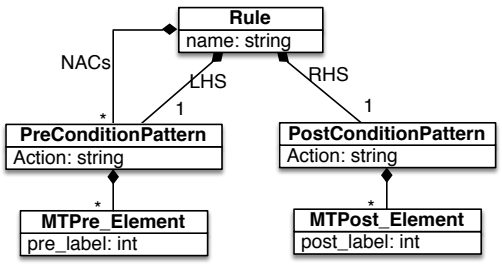

Figure 1. Structure of a transformation rule (from [1])

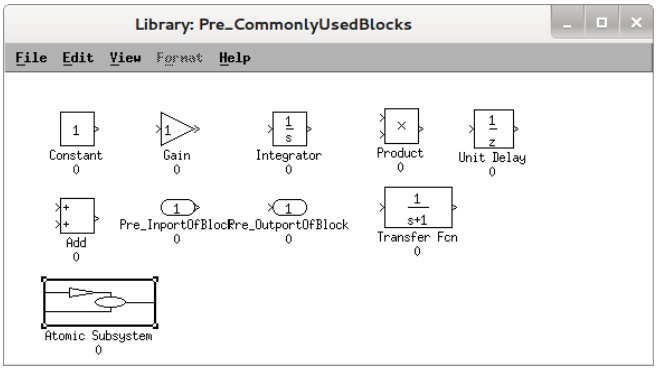

Figure 2. The created pre-condition library of blocks usable in the LHS and NAC of a transformation rule

stand and correct, (2) the specification can be altered by users of the tool instead of requiring a new tool release, (3) one can reason about the specification and the models it describes, and, (4) a syntax-directed modeling environment can automatically be generated from the metamodel, as shown in AToM^3 [2] and its descendant AToMPM. Kühne et al. [1] argue that the advantages of metamodeling not only apply to the modeling language definitions, but also to transformation languages. To demonstrate the advantages, they explicitly (meta)model visual rule-based transformation languages for visual modeling languages.

Rule-based model transformation languages work on typed, attributed, and directed graphs that represent the model. A transformation rule represents manipulation operations on the represented model. Figure 1 depicts the structure of a transformation rule. A rule consists of a left-hand side (LHS) pattern representing the pre-condition for the applicability of the rule. The right-hand side (RHS) pattern defines the outcome of the operation. A set of negative application condition (NAC) patterns can be defined to prevent the application of the rule.

Transformation rule patterns are very similar to the models that can be created as Simulink block diagrams. First, those patterns are not necessarily well-formed models in the Simulink language. The pattern language is thus a *relaxed* version of the original modeling language. For example, a pattern may be specified that includes a block without any incoming connections. Such unconnected input ports would be invalid in a Simulink model. Second, a number of blocks should be added to the metamodel of the transformation rules. The pattern languages is thus an *augemented* version of the original modeling language. For example, it is important to support specifying an Abstract Block that is matched to any block in the Simulink model. Also, extra parameters must be added to the metamodel to identify elements across the pre- and post-condition patterns. This identification is implemented by adding a label to the corresponding modeling elements. Finally, the datatypes of the model elements must be adapted to allow for specifying constraints on the element in the pre-condition pattern as well as actions in the post-condition pattern in order to compute the new value of parameters. The pattern language is thus a *modified* version of the original modeling language.

The structured conversion of an original metamodel of a language into a tailored pattern metamodel that is usable in transformation rules is performed in three steps: relaxation, augmentation, and modification, and hence referred to as RAMification of a metamodel. Next, RAMification is applied to the Simulink tool.

3. RAMIFICATION OF SIMULINK® BLOCK DIAGRAMS

A transformation editor for Simulink block diagrams is created in the Simulink tool. The transformation editor is separated into two parts. The first part focuses on creating transformation rules with a LHS, RHS, and a set of NACs. The second part creates a scheduling language to combine different rules and rule types.

3.1. RAMification of Simulink® Blocks

The process of RAMification first creates two new libraries, one for the pre-condition blocks and one for the post-condition blocks. The blocks that are necessary in the transformation patterns are copied from the original block library into both these new libraries. A unique name is given to the library, a *Pre_* and *Post_* string is added to the name for later identification. This name is used by Simulink to assign a unique type to the blocks in the library. For example, the *Constant* block in the *Pre_CommonlyUsedBlocks* library has the type */Pre_CommonlyUsedBlocks/Constant*. Figure 2 shows the pre-condition library with some commonly used

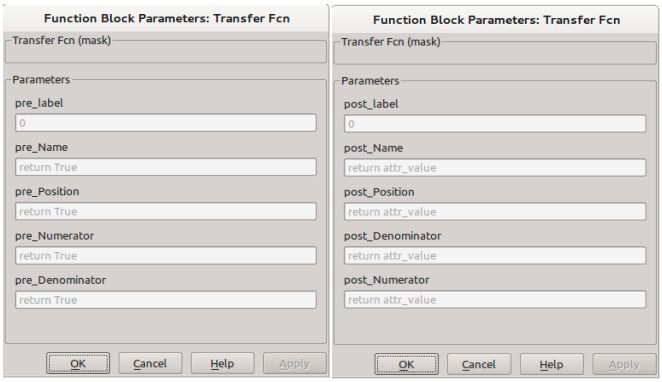

Figure 3. RAMified parameters of the Transfer Function block

blocks. The RAMification continues with the following steps:

- Relaxation: For the relaxation of the metamodel, no specific actions are necessary as Simulink by default allows the specification of models that cannot be simulated.
- Augmentation: Three new blocks are added to both the libraries. An *AbstractBlock* is added for transforming any kind of block in the Simulink block diagram. The other two blocks are explicit representations of the *Input Port* and *Output Port* of a block. This is necessary to reconnect blocks during the execution of a transformation rule. For each of the blocks, a mask [1] is created with a *label* attribute to identify nodes across the LHS, NACs, and RHS patterns.
- Modification: For each block, the attributes are added to the mask with a *pre_* or *post_* prefix. The datatype of the attribute is a string. The strings represent Python executable code to capture: a constraint on the value in case of pre-condition block or action code to change the parameter in case of a post-condition block. Figure 3 shows the mask of the Transfer Function block in the pre-and post-condition library.

Note that blocks were RAMified manually. MATLAB scripts can be built to automatically RAMify the Simulink libraries. This is future work.

3.2. Creating Rules

Now that a set of pre-condition and post-condition blocks are present in both libraries, a set of rules can be created. For this, three subsystem blocks are used, as

[1] A mask is a custom user interface for a block, see http://www.mathworks.com/help/simulink/gui/mask-editor-overview.html

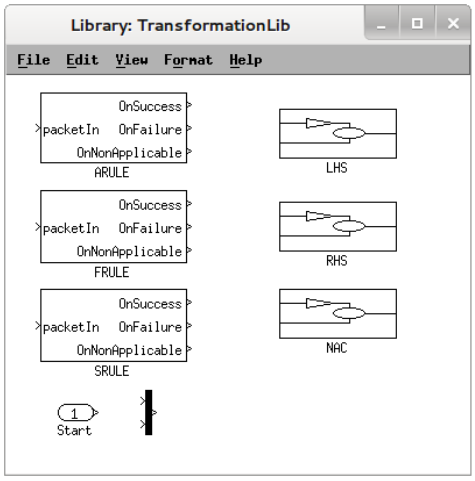

Figure 4. Library elements to create rules and schedules

shown in Figure 4. The first block represents a LHS. The name of the LHS subsystem of the rule is prefixed with the *LHS* prefix. A similar mechanism is employed for the NACs and the RHS. The mask of the LHS and NAC block contains a field for the *Constraint*. The mask of the RHS block has an *Action* attribute. The LHS, NAC, and RHS blocks are subsystem blocks. Instantiating the blocks allows for the opening of the subsystem to create pre- and post-condition patterns. Only blocks from the previously created pattern libraries can be used within the subsystems.

3.3. Combining Rules

Finally, a schedule can be defined to combine different rules. Figure 4 shows the different rule types that can be employed to schedule rules:

- ARULE: The atomic rule finds a single instance of the pre-condition pattern and rewrites this in the model.
- FRULE: The For-all rule finds all the occurrences of the pre-condition pattern and rewrites all the occurrences in the model. It is assumed that all the occurrences are independent.
- SRULE: The star rule applies the rule sequentially as long as a match of the pre-condition pattern is satisfied in the model. That is, after the model has been matched and transformed, the resulting model is checked for a match of the pre-condition pattern and possibly transformed by the same rule.

Again a prefix on the name of the subsystem is used to define the type of the transformation rule. Instantiating the blocks allows for opening the subsystem where a LHS, RHS, and a set of NACs can be defined. The semantics of the scheduling language is slightly differ-

ent from the normal Simulink semantics. It also uses a data communication mechanism to combine the different rules, but at any time during execution, only a single Simulink model is present in the schedule. Rules can be connected using the defined ports of the blocks. The *packetIn* port is the input port of a transformation. On successful rule application, the model is output on the *onSuccess* port. If the rule cannot be applied to a model, the non-transformed model is written on the *OnNonApplicable* port. Finally, the *OnFailure* port outputs the model in case an exception occurs during transformation.

The starting point of the transformation is indicated by the start block. Complex transformation combinations, with loops, can be defined with help of the Simulink *Mux* block.

3.4. Dealing with Hierarchy

During the creation of a transformation language in and for Simulink models, the existence of hierarchy in Simulink models was not mentioned. Hierarchy is an important aspect of the Simulink language, though. It allows structuring and reusing parts of Simulink models during design. When defining a pre-condition pattern without the explicit notion of hierarchy, the semantics of our rule assume that the pattern can be applied within any subsystem. The post-condition pattern does not change the hierarchy explicitly if no subsystems are present. If it is feasible to change hierarchy levels of certain blocks or to create new hierarchy levels then this can be specified in the patterns by using the same hierarchy mechanisms that are present in Simulink already. It requires that the *Subsystem* is also RAMified as shown in Figure 2.

4. SIMULINK® TRANSFORMATION ARCHITECTURE

Finally, the transformations must be made operational. This section defines the elements that are necessary to execute the model transformations in Simulink. First, the metamodel of Simulink models is introduced in order to understand the corresponding abstract syntax of generated models.

4.1. The Simulink® Metamodel

A simplified metamodel specifying the abstract syntax of a Simulink block diagram is shown in Fig. 5. The central entity in a Simulink block diagram is the *Block*. Blocks represent dynamic systems with input and output such as arithmetic operators, continuous-time integrators, and relational operators. The Block is the super class for all defined blocks in the metamodel. Figure 5 only shows a small subset of the blocks and associated parameters defined in the Simulink tool. Links are defined between block ports and represent the time-varying signals shared between connected blocks. Different ports are defined in the metamodel. The *input* ports and *output* ports must be associated with a single block. Depending on the block subclass, a block can have multiple input-type ports and usually a single output port associated with it. The *subsystem* block, however, often has multiple output ports. For more information about the metamodel of Simulink see [3].

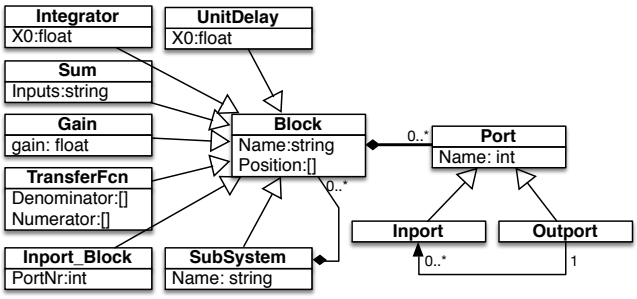

Figure 5. Simplified metamodel of a Simulink® block diagram

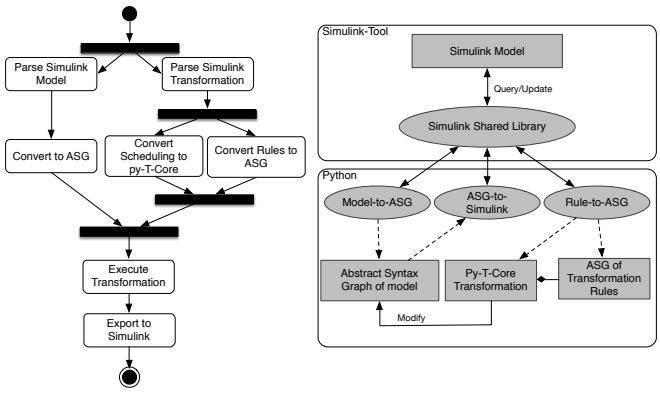

Figure 6. Architecture and workflow of Simulink® model transformation

4.2. Workflow

Figure 6 shows the workflow and architecture supporting the execution of Simulink model transformations. The process starts by parsing the Simulink model, using the *model-to-ASG* module, into an abstract syntax graph (ASG) representation that is usable by the transformation engine. Input ports and Output ports of the blocks are explicitly represented in our metamodel. These are added during the creation of the ab-

stract syntax graph. Hierarchy is added by creating containment relations between subsystems and their contained blocks. Similarly, a transformation in Simulink is parsed using the *Rule-to-ASG* module. The transformation rules are converted into an abstract syntax representation. An extra subsystem is added at the top of the LHS, NAC, and RHS to support the hierarchy of Simulink. The schedule is converted into a Python file that combines the different transformations.

Upon execution of the transformation schedule, the ASG representation of the model is modified. When the entire transformation process is completed, the ASG model is exported back into Simulink. The transformation can be configured to overwrite the existing model or to create a new Simulink model with a different name. An alternative generator for the *rule-to-ASG* is also defined. The alternative generator augments the transformation rules with instrumented code to directly modify the Simulink model so visual feedback can directly be seen in the Simulink tool.

The following subsections explore components of the transformation architecture in more detail.

4.3. Communicating with Simulink®

The *Model-to-ASG, ASG-to-Simulink* and *Rule-to-ASG* modules, shown in Fig. 6, must communicate with the Simulink tool. The shared libraries provided by the Simulink tool allow Simulink models to be queried and updated. A code excerpt to open communication with Simulink is shown in Listing 1. The shared libraries are provided on all platforms and are less susceptible to changes compared to the parsing of the serialized versions of models in textual mdl format.

Code Listing 1. "Communicating with Simulink®"

```
class InvokeSharedObject:
    def __init__(self):
        self.libeng = cdll.\
        LoadLibrary("libeng.so")
        self.libmx = cdll.\
        LoadLibrary("libmx.so")
        self.libmat = cdll.\
        LoadLibrary("libmat.so")

    def engOpen(self,startcmd):
        fun = self.libeng.engOpen
        fun.argtypes = [c_char_p]
        fun.restype = c_void_p
        return fun(startcmd)

    def ...
```

4.4. T-Core: Graph Rewriting

T-Core is a minimal collection of model transformation primitives. They allow for rapid implementation of transformation languages [4]. T-Core is not restricted to any form of specification of transformation units, be it rule-based, constraint-based, or function-based. It can also represent bidirectional and functional transformations as well as queries. T-Core modularly encapsulates the combination of these primitives through composition, re-use, and a common interface. It is an executable module that is easily integrable with a programming or modeling language.

The scheduling language used here is Python, resulting in a transformation language called Py-T-Core [4]. Py-T-Core defines the transformation units in the scheduling language presented in Section 3. such as ARULE, FRULE, and SRULE.

5. CASE STUDY

Our approach is illustrated through an adaptive control example based on previous work [5]. The design of an adaptive controller from requirements to an implementation moves through various stages during each of which myriad design choices must be made:

- For a digital implementation, the continuous-time model of the controller must be discretized. A number of different methods are available in the literature.
- To implement on a computer architecture, a task-based representation must be created from the discretized model.
- The tasked representation requires that communication of data between different tasks be handled.
- The effects of sensors and actuators, such as analog to digital conversion (ADC) and digital to analog conversion (DAC), must be validated in the computational model.

The steps defined above are usually handled in an incremental manner, whereby the outcome of each step is simulated and checked against the requirements of the system. Figure 7 shows the structure of the adaptive controller. The *plant* block represents the system that is controlled by the *control* block. To this end, the *control* block takes as input the output of the *plant* block and compares this to a reference behavior generated by the *r* block to determine a control input for the *plant* block. The controller in the *control* block is based on two parameters *th1* and *th2* that are continuously adapted by an *adaptation* block based on the *plant* block output, the reference behavior and a reference model that is represented by the *reference model* block. The reference model generates a desired behavior of the *plant* block output based on a smoothing function that provides a measure of the discontinuous behavior the plant should achieve. As shown in Fig. 7 this smoothing behavior is imple-

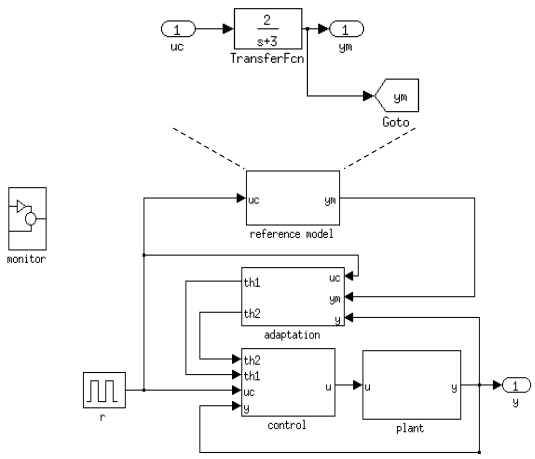

Figure 7. Model of the adaptive controller in continuous-time

mented as a first order low-pass filter.

The continuous-time model can be simulated using the provided solvers in the Simulink tool. Figure 8 shows the behavior of the model. For each of the itemized steps, different rule-based transformations can be developed. In the case study, a rule-based model transformation is created and executed in Simulink to produce a discretized approximation of the adaptive controller. Afterwards, optimization transformations are performed on that model.

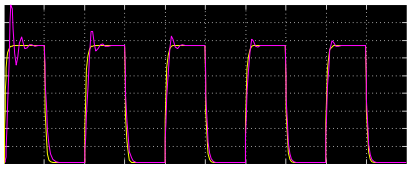

Figure 8. Simulation result of the continuous-time adaptive controller in Simulink®

5.1. Discretization

The discretization of the continuous-time model is based on approximations where continuous-time integration blocks are transformed into sets of blocks representing the approximation [6].

Examples of these approximations are:

Forward Euler: the most basic explicit method for numerical integration. The approximation is as follows: $\frac{1}{s} \approx \frac{T}{z-1}$, with T the discrete time-step.

Backward Euler: the first-order implicit method with the following approximation: $\frac{1}{s} \approx \frac{T \times z}{z-1}$

Bilinear or trapezoidal Approximation: refers to an implicit second-order approximation.

Heun Integration: is the explicit second-order version of the trapezoidal rule.

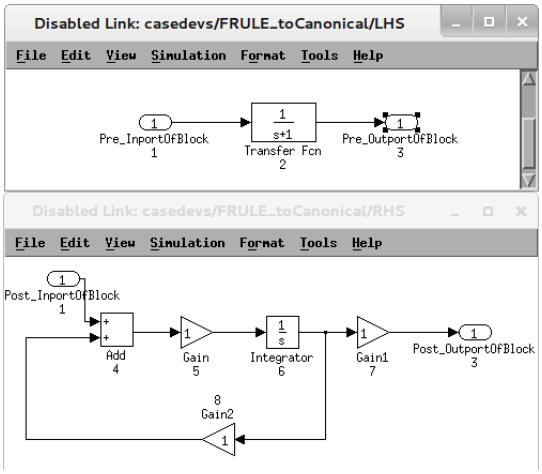

Figure 9. Expanding a first-order transfer function to its canonical representation

Implicit methods introduce direct algebraic relations that often present challenges for applications with bounded response times (e.g., hardware-in-the-loop simulation). They are, however, more stable and can be used with a larger time step, thus rendering them computationally less intensive compared to the explicit methods.

In addition to continuous-time integration blocks, the continuous-time adaptive controller model also contains transfer function blocks. A transfer function describes the relationship between input and output in the Laplace domain. As the transfer function blocks used are continuous, a transformation is defined to a canonical representation that only contains gain, integrator, and sum blocks. Figure 9 shows a transformation to expand a first-order transfer function into its canonical representation. The numerator and denominator values of the transfer function block are used to set the parameters of the different gain-blocks in the RHS. A more general transformation sequence can be defined to transform an n-order transfer function to its canonical version.

Figure 10 depicts the transformation rule to replace an integrator block with its forward Euler approximation. Again, the parameters of the LHS integrator block, such as the initial condition value, are used in the RHS. The rule defined in Fig. 10 does not involve hierarchy. It is thus applied to any integrator block in the model. However, when discretizing for code generation (not

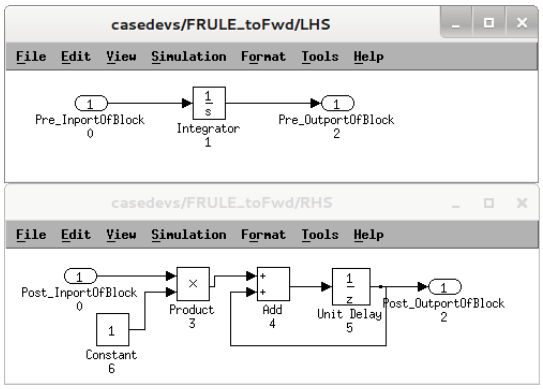

Figure 10. Transformation rule for discretizing an integration block with its forward Euler approximation

for explicitly modeling the solver, as in [7]), the plant model must remain continuous. This can be achieved by placing all the blocks in the LHS and RHS into a RAMified subsystem block. The name of the subsystem in the model is utilized to inhibit the application of the rule on the plant subsystem. This is performed by returning False when the name contains the 'plant' string on the subsystem's *pre_Name* condition.

Similar transformations are defined for approximating using a backward Euler, Bilinear approximation and Heun integration.

5.2. Optimizations

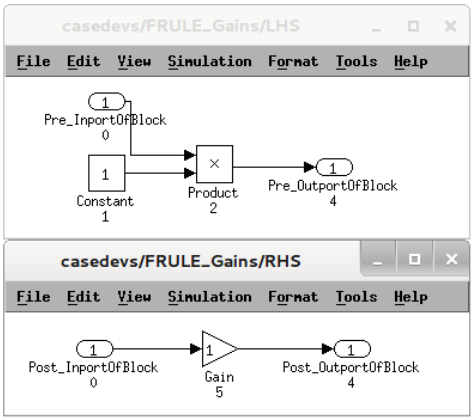

Figure 11. Transformation rule for optimization

Optimization transformation rules were created that will remove blocks from the model. Figure 11 shows a transformation rule that performs constant propagation by replacing a pattern of a Constant block connected to a Product block by a Gain block. Other optimization rules include the removal of a Gain block with a gain value of 1 and reconnecting the links and the removal of the Sum of two constants into a single constant. Depending on the code generation options chosen, these optimizations result in code that is more efficient compared to the generated code of the original model.

5.3. Putting it all together: the schedule

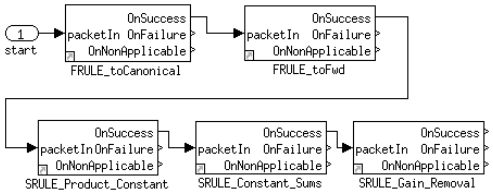

Figure 12. Transformation schedule

The schedule of the transformation is straightforward, as shown in Figure 12. All transformations are connected via the onSuccess-port. If another approximation is needed, the FRULE_toFwd must be replaced by the counterpart transformation for bilinear, Heun, or Backward Euler. After transformation, the solver of the Simulink tool is changed in order to reflect the changes made by the transformation. In this case, the time-step was chosen at 10 ms. Thus, the solver was changed to a fixed-step solver with a time-step of 0.01.

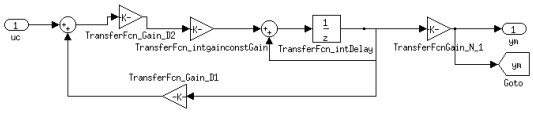

Figure 13. Resulting blocks in the reference model subsystem after transformation

Figure 13 shows the blocks in the *reference model* subsystem of Fig. 7. The model was post edited to improve the block layout as the transformation places all the blocks on top of one another (as specified in the model transformation model). Functional testing shows that simulation of the model yields a discretized result virtually identical to that in Fig. 8.

6. RELATED WORK

The process of RAMification has been applied on previous occasions. In [8], automatic RAMification of metamodels has been implemented in the AToM^3 tool. Similarly, the process of RAMification has been applied to MetaEdit+, a commercial domain-specific modeling

tool, in [9]. Both processes start from a designed metamodel to create the pattern metamodels. In the work presented here, even though the metamodel is only implicitly defined within the Simulink tool, the process is shown to still be applicable.

External transformation tools for Simulink models have been presented in related work [10, 11]. For example, in [12, 13], the VMTS tool is used to create and execute transformations on Simulink models to flatten and to set the data-types of the different links between the blocks. These approaches import the Simulink model in another tool. Exporting and importing is performed by using the shared libraries or by parsing the Simulink model file. In [14], transformations are defined to structure the model before clone detection. The tool used to execute these transformations is unknown. In contrast, the work presented here designs and executes the transformations directly in the Simulink tool itself with the help of the T-Core transformation libraries.

Haber *et al.* create a special type of transformations for and in Simulink, named Delta-Simulink[15]. It is employed for variability modeling where different operations (remove, add, and replace block) are used to transform the model in the context of product lines. The operations only work on subsystems, connections, inports, and outports, though.

7. CONCLUSIONS AND FUTURE WORK

In this paper the commercial Simulink tool is augmented with rule-based model transformation support based on a process called RAMification. Special transformation libraries were created that allow specifying model transformations and transformation rules directly within Simulink, using the same concrete syntax as Simulink models. It was explained how this infrastructure can be made operational. The contribution of the paper was demonstrated using a case study where a set of transformation rules were defined to discretize and optimize a model before simulation and code generation. The work will be extended by creating scripts to automatically RAMify Simulink blocksets.

ACKNOWLEDGEMENTS

The partial support for this work by the NECSIS project (Automotive Partnership Canada) is gratefully acknowledged.
The partial support for this work by MathWorks® is gratefully acknowledged.
The authors would also like to thank Tamás Mészáros from the Budapest University of Technology and Economics for providing us with their code for communicating with the Simulink tool.

REFERENCES

[1] T Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Explicit transformation modeling. *Lecture Notes in Computer Science*, 6002:240–255, 2010.

[2] Juan De Lara, Hans Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM3. *Software and Systems Modeling*, 3(3):194–209, 2004.

[3] Sandeep Neema. Analysis of Matlab Simulink and Stateflow Data Model. Technical report, Vanderbilt University, 2001.

[4] Eugene Syriani, Hans Vangheluwe, and Brian LaShomb. T-Core: a framework for custom-built model transformation engines. *Software & Systems Modeling*, August 2013.

[5] Zhi Han and PJ Mosterman. Detecting data store access conflict in Simulink by solving Boolean satisfiability problems. In *American Control Conference (ACC)*, 2010.

[6] Uri M Ascher and Linda R Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. 1997.

[7] Pieter J. Mosterman and Justyna Zander. Advancing Model-Based Design by Modeling Approximations of Computational Semantics. In *EOOLT*, pages 3–7, 2011.

[8] Eugene Syriani, Jeff Gray, and Hans Vangheluwe. Modeling a Model Transformation Language. *Domain Engineering*, pages 211–237, 2013.

[9] Simon Van Mierlo and Hans Vangheluwe. Adding Rule-Based Model Transformation to Modelling Languages in MetaEdit +. *Electronic Communications of the EASST 7th International Workshop on Graph Based Tools (GraBaTs 2012)*, 54, 2012.

[10] Aditya Agrawal, Gyula Simon, and Gabor Karsai. Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. *Electronic Notes in Theoretical Computer Science*, 109:43–56, December 2004.

[11] Ingo Stürmer, Ingo Kreuz, Wilhelm Schäfer, and Andy Schürr. The MATE Approach: Enhanced Simulink® and Stateflow® Model Transformation. In *Proceedings of MathWorks Automotive Conference*, 2007.

[12] P Fehér, T Mészáros, PJ Mosterman, and L Lengyel. Flattening Virtual Simulink Subsystems with Graph Transformation. In *Proceedings of the Workshop on Complex Systems Modelling and Simulation*, 2013.

[13] P Fehér, T Mészáros, L Lengyel, and PJ Mosterman. Data Type Propagation in Simulink Models with Graph Transformation. In *3rd Eastern European Regional Conference on the Engineering of Computer Based Systems*, 2013.

[14] Bakr Al-batran, Bernhard Schätz, and Benjamin Hummel. Semantic Clone Detection for Model-Based Development of Embedded Systems. In *MODELS 2011, LNCS 6981*, pages 258–272. 2011.

[15] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-class variability modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems (VaMoS)*, New York, New York, USA, 2013. ACM Press.