Flattening Virtual Simulink Subsystems with Graph Transformation

Péter Fehér[1], Tamás Mészáros[1], Pieter J. Mosterman[2], and László Lengyel[1]

[1] Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Budapest, Hungary
{feher.peter, mesztam, lengyel}@aut.bme.hu
[2] Design Automation Department
MathWorks
Natick, MA, USA
pieter.mosterman@mathworks.com

Abstract. Nowadays embedded systems are often modeled using MATLAB®, Simulink® and Stateflow® to simulate their behavior and facilitate design space exploration. As design progresses, models are increasingly elaborated by gradually adding implementation detail. An important elaboration is the execution order of the elements in a model. This execution order is based on a sorted list of all semantic relevant model elements. Therefore, it is fundamental to remove model elements that only have a syntactic implication such as hierarchical levels with no semantic bearing. The corresponding language construct in Simulink is the virtual subsystem. Thus, to create an implementation or to execute a model, Simulink performs a flattening model transformation that eliminates virtual subsystems. The work presented in this paper raises the level of abstraction of the model transformation by modeling the transformation itself in order to unlock the potential for reuse, platform independence, etc.. To this end, the transformation is implemented by applying graph transformation methods. An analysis of the solution shows the transformation model is proper (e.g., it terminates).

1 Introduction

Advances in electronics miniaturization combined with an understanding of computing are driving an ever-increasing complexity of technical systems of truly all sorts (consumer electronics, defense, aerospace, automotive industry, etc.). Not only does the increasing capability of electronics enable more extensive logic to be implemented, the robustness and efficiency in communication protocols that it supports has been the driver of ever more network connected systems. The corresponding systems operate at the confluence of cyberspace, the physical world, and human participation. Recently, these systems have been termed Cyber-Physical Systems [1].

Raising the level of abstraction is an important tool to manage the enormous complexity of such Cyber-Physical Systems. To this end, Model-Based Design (e.g. [21], [20], [27]) introduces levels of abstraction in the form of computational models with executable semantics. At the various levels of abstraction, only concerns pertinent to

the particular design task are included while implementation aspects are deferred to be addressed in more detailed models. Throughout the design of the embedded system part of a Cyber-Physical System, these models are then elaborated to include increasing implementation detail. The elaboration terminates when a level of detail is arrived at from which an implementation can be automatically generated. The implementation may be either in software by generating C code or in hardware by generating HDL.

The support for abstractions is important in formulating a design problem in the problem space. The design then concentrates on transforming the problem formulation into a solution formulation. In this context, it is of great value that the original problem formulation can be void of solution aspects. This is why domain-specific modeling is becoming increasingly popular to describe complex systems. It is a powerful, but still understandable technique. Its main strength lies in the application of the domain-specific languages. A domain-specific language is a specialized language that can be tailored to a certain problem domain; therefore, it is more efficient, than the general purpose languages that often are tailored to a solution domain (and domain specific in that sense) [18] [17].

Modern model transformation approaches are becoming increasingly valuable in software development because of the ability to capture domain knowledge in a declarative manner. This enables various steps in the software development to be specified separate from one another with apparent advantages such as reuse. In embedded system design, the computational functionality that is ultimately embedded moves through a series of design stages where different software representations are used. For example, before generating the code that is to run on the final target, code may be generated that includes additional monitoring functionality. As another example, the software representation may be designed in floating point data types before being transformed into fixed point data types.

Today Simulink® [4] is a popular tool for control system design in industry. Therefore, applying model transformations for embedded software design purposes on Simulink models renders the developed technology easily adaptable and adoptable by industry. However, currently it is impossible to define and model declarative model transformations inside the Simulink environment. Therefore, a modeling and model processing framework is applied. The Visual Modeling and Transformation System (VMTS) [10] [7] framework has been prepared to be able to communicate with the Simulink environment. In this manner, with the help of modeled model transformation, problems can be solved at the most appropriate abstraction level. In this case the most appropriate abstraction level means that the required model optimization, modification, or traversing can be expressed in the Simulink domain. Thus Simulink users can use their well-known entities to define the required processing. This is the fundamental premise of Computer Automated Multiparadigm Modeling; to use the most appropriate formalism for representing a problem at the most appropriate level of abstraction [23] [24].

While operating at a given level of abstraction, two further mechanisms are often employed to scale system complexity: partitioning and hierarchy [22]. In the Simulink environment, hierarchy is supported as a purely syntactic construct by *virtual subsystems* and as a construct with semantic implications by *nonvirtual subsystems*. These subsystems are represented as blocks with input and output ports that are used to con-

nect subsystem blocks. Subsystem blocks may contain other subsystem bocks or primitive blocks that represent behavior without being able to be further decomposed.

Before a Simulink model is executed, the engine creates an execution list with an order in which all of the blocks are executed. The execution list is computed from the sorted list, which is also generated by the Simulink engine based on the control and data dependencies that determine how the different blocks can follow each other in an overall execution. To create this list, the semantically superfluous hierarchical layers have to be flattened. So, the virtual subsystems that are only graphical syntax and that have no bearing on execution semantics are flattened before the sorted list is generated [5] [16].

This paper focuses on a novel solution to flattening virtual subsystems in Simulink models. This approach is based on a model transformation created in VMTS. Using model transformation to solve this issue helps raise the abstraction level of the transformation from the frequently used API programming to the level of software modeling. The solution possesses all the advantageous characteristic of the model transformation, for example, it is reusable, transparent, and platform independent. The different attributes of the transformation are also examined in detail in this paper.

The remainder of the paper is organized as follows. Section 2 introduces related work. Section 3 briefly presents the VMTS and its graph rewriting-based model transformation capabilities. The basics of the communication between Simulink and VMTS are discussed in Section 4. Next, Section 5 introduces the flattening transformation. In Section 6, the properties of the flattening transformation are examined. Next, in Section 7, an example Simulink model is processed with the presented flattener transformation. Finally, concluding remarks presented.

2 Related Work

In [8] a formal description is given about a translation process that can convert a well-defined subset of Simulink block diagram models and Stateflow® [6] state transition diagram models into a standard form of hybrid automata [9]. This transformation is implemented with graph transformation. As a result, different verification tools for hybrid automata can operate on the industry-standard Simulink and Stateflow models.

To specify program transformation such as program optimizers, other work [13] developed a successful method. This method can be uniformly applied to analysis and transformation. The underlying technological solution is based on graph transformation.

Since the design patterns are valuable parts of the different phases of the software development, there is a necessity to specify them on a high level of abstraction instead of capturing this information informally. Other work [28] uses different graph transformation to support this necessity. With the help of this approach the design patterns can be specified on a higher abstraction level.

Another algorithm that works with Simulink models is presented in [14]. This algorithm is introduced for mapping discrete-time Simulink models to Lustre programs. Here, the transformation is not formally modeled as a declarative graph transformation, though.

In other related work [11], a new data model for tool integration is presented. This approach extends existing data models by an abstract graph model. Here, the manipulation is based on model transformation as formal graph transformations.

The work presented in this paper is different in that it does not address any semantic complications. A purely syntactic model transformation is developed. Moreover, in contrast to the aforementioned exogenous transformations, the transformation developed in the work in this paper is endogenous (i.e., no change of formalism) [19]. Finally, there are no restrictions to the Simulink modeling formalism necessary as the presented work applies to the full set of Simulink blocks.

In [26] and [15] novel approaches are proposed to represent Simulink models as directed, sparse graphs. Each subsystem graph is added to the highest layer graph.

3 VMTS, The Modeling Framework

The Visual Modeling and Transformation System (VMTS) is a general purpose metamodeling environment supporting an arbitrary number of metamodel levels. Models in VMTS are represented as directed, attributed graphs. The edges of the graphs are also attributed. The visualization of models is supported by the VMTS Presentation Framework (VPF) [25]. VPF is a highly customizable presentation layer built on domain-specific plugins, which can be defined in a declarative manner.

VMTS is also a transformation system. It uses a graph rewriting-based model transformation approach or a template-based text generation. Templates are used mainly to produce textual output from model definitions in an efficient way, while graph transformation can describe transformations in a visual and formal way.

In VMTS the Left-Hand Side (LHS) and the Right-Hand Side (RHS) of the transformation are represented together. In this manner, the transformation itself can be more expressive. In order to distinguish the LHS from the RHS in the presentation layer, the VMTS uses different colors. The elements represented with blue color are created by the transformation rule. This means that if the LHS and the RHS would be depicted in two separated graphs, these elements would be only part of the RHS graph. Similarly, the red color indicates that the given element will be deleted by the transformation rule. The yellow color is used when an edge between two elements will be replaced. In this case the type and the attributes of the edge will not change. The gray background means that the element will be modified. With the help of these colors, the transformation process is easily understandable. There is always an option to apply imperative constraints to each element, but this is not depicted separately.

The control flow language of the VMTS [12] contains exactly one start state and one or more end state objects. The applicable rules are defined in the rule containers. The rule containers determine which transformation rule must be applied at the given control flow state. This means that exactly one rule belongs to each rule container. The application number of the rule can also be defined here. By default, the VMTS tries to find just one match for the LHS of the transformation rule. However, if the *IsExhaustive* attribute of the rule container is set to true, then the rule will be applied repeatedly as long as its LHS pattern can be found in the model. Figure 1 depicts an example control

flow model; actually this is the control flow of the flattening transformation, which will be presented in detail in Section 5.

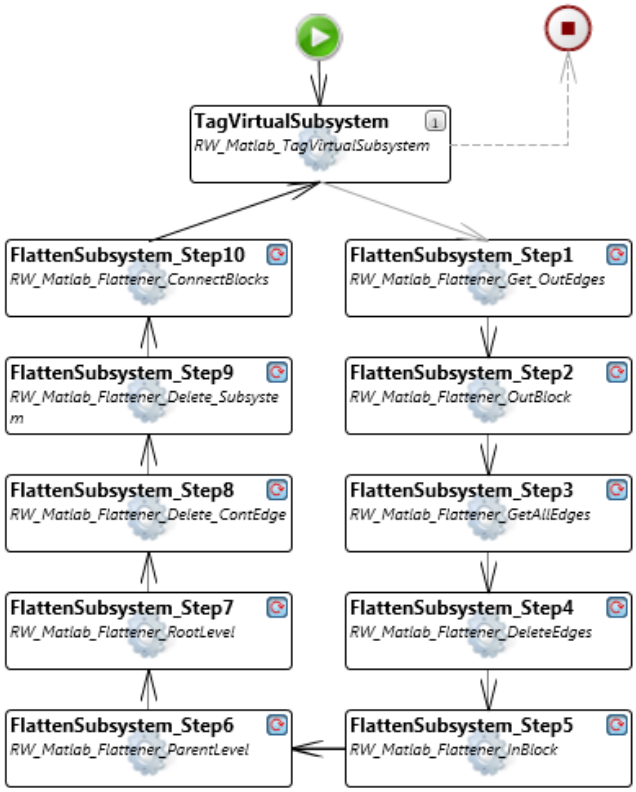


Fig. 1: The control flow of the TRANSFLATTENER transformation

The edges are used to determine the sequence of the rule containers. The control flow follows an edge in order of the result of the rule application. In VMTS, the edge to be followed in case of successful rule application is depicted with a solid gray flow edge and in case of a failed rule application with a dashed gray flow edge. Solid black flow edges represent the edges that can be followed in both cases.

4 Communication between Simulink® and VMTS

Since the model is created in Simulink, which is part of the MATLAB® [3] environment and the transformation is created in another system (VMTS) there is a need to establish a communication method between the two systems.

To be able to represent Simulink models in VMTS, the metamodel of the Simulink languages, which are organized in various different libraries (also called *blocksets*), is

required. Since in Simulink there is no hard boundary between the different languages, that is, a given block can be connected to almost everything else, a common Simulink metamodel was created. This metamodel contains all the elements of the Simulink library. The generation of this metamodel consists of the following two steps.

First, a *core* metamodel was created that contains the *Block* element, which is the common ancestor of all the nodes in Simulink models, and a descendant *Subsystem* node, which expresses the common ancestor of Simulink Subsystems. This metamodel also contains the *Signal* edge and a *Containment* edge to reflect containment hierarchy between nodes.

Then, by programmatically traversing the base Simulink library, this metamodel has been extended with the other nodes found in the different specialized libraries. For each Simulink element, exactly one node was generated. This resulted in several hundred new metamodel elements.

In addition to the metamodel, the VMTS must be prepared to read and write Simulink models. Thus, a new kind of data exchange layer was generated for communicating with MATLAB. To modify Simulink models, the P/Invoke technology [2] has been chosen. This has the advantage that the MATLAB interpreter can be called directly through DLL calls, instead of manipulating the textual model (mdl extension) files. This way the VMTS is independent of file format changes, and the changes performed on the VMTS model can be made visible, live during the transformation execution, on the Simulink diagrams as well. Furthermore, the values that are only available during simulation time of a Simulink model can be accessed also.

5 The Flattening Transformation

This section introduces and discusses the details of the transformation TRANSFLATTENER. The transformation is created in VMTS. As it was mentioned before, its goal is to process Simulink models and flatten its virtual subsystems.

For a better understanding the final control flow is presented first, which is shown in Fig. 1. This model defines how the transformation rules follow each other.

At first, the transformation checks if there is a virtual *Subsystem* block in the model. So the RW_MATLAB_TAGVIRTUALSUBSYSTEM transformation rule attempts to match a simple *Subsystem* block that has the *IsVirtualSubsystem* attribute set to true. If the transformation engine does not find a match for this rule, then there is no virtual *Subsystem* in the model, thus the transformation terminates. Otherwise, the transformation steps into the loop, which processes this *Subsystem*.

In Simulink, a block cannot be directly connected to a block on a different hierarchical level. When a block is moved out from a *Subsystem*, it loses all its edges automatically. This means that the transformation must take into account the connections between the blocks, and must take care of creating the necessary edges. Moreover, when a *Subsystem* is flattened, the *Inport* and *Outport* blocks of the *Subsystem* are deleted. So the transformation also must ensure that the blocks connected to the appropriate ports will be connected to each other after the processing.

The *Inport* and *Outport* blocks represent the input and output ports of the *Subsystem*. Each port of the *Subsystem* is associated with an appropriate block in the *Subsystem*.

For example if a *Subsystem* has two input ports and three output ports, then there are two *Inport* and three *Outport* blocks in the *Subsystem*, and these blocks have a reference number to the port they belong to. This behavior is presented in Fig. 8.

In the aforementioned loop, which is responsible for processing the virtual *Subsystem*, the first applied rule is the RW_MATLAB_FLATTENER_GET_OUTEDGES transformation rule. It is depicted in Fig. 2. In order to handle the deletion of the ports this rule matches the outgoing edges of a *Subsystem*, and deletes them if a match is found. In the meantime, the identifier of the block connected to the *Outport* port of the *Subsystem* (the *toNode* in Fig. 2) and the port number that the edge is connected to are stored in the appropriate *Outport* block (the *endNode* in Fig. 2).

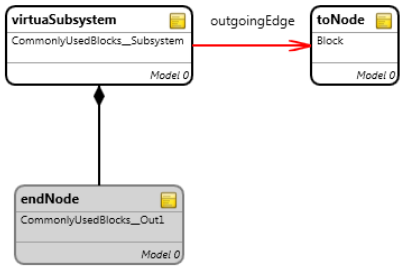


Fig. 2: The transformation rule RW_MATLAB_FLATTENER_GET_OUTEDGES

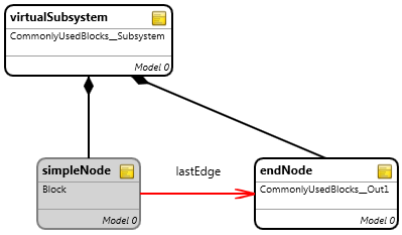


Fig. 3: The transformation rule RW_MATLAB_FLATTENER_OUTBLOCK

After processing outgoing edges from all *Subsystem* type nodes, the transformation moves to the next transformation rule: RW_MATLAB_FLATTENER_OUTBLOCK (Fig. 3). This rule matches blocks that have outgoing edges to an *Outport* block and if a match is found deletes the edge connecting them. It also copies the information stored in the *Outport* block into the other one (the *simpleNode* in Fig. 3) and extends it with the port number where the edge starts. This way this block knows all the information about the edges the transformation must create after the block is moved out of the *Subsystem*. This information consists of the followings:

- The port number the edge starts,
- The identifier the edge is connected to,
- The port number the edge ends,
- The necessary attributes of the edge.

In the previous two steps the transformation focused on the blocks connected to the *Out* ports and *Outport* blocks of the *Subsystem*. The next rule embodies the same functionality with every block in the *Subsystem*. The RW_MATLAB_FLATTENER_GETALLEDGES rule is depicted in Fig. 4. It does not differentiate based on the type of the block, which means that it matches *Inport* blocks as well as the blocks processed previously (i.e., the ones connected to the *Outport* blocks). The reason for this is the possibility that

a block may have multiple outgoing edges and the transformation needs information about every edge. The rule is matched for every element in the *Subsystem* exactly once and stores all information about their outgoing edges.

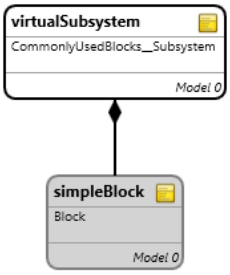


Fig. 4: The transformation rule RW_MATLAB_FLATTENER_GETALLEDGES

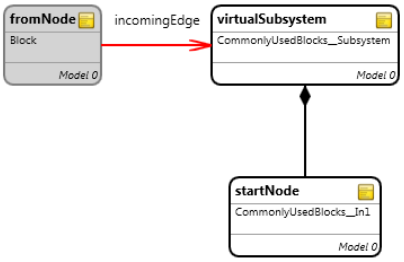


Fig. 5: The transformation rule RW_MATLAB_FLATTENER_INBLOCK

After the first three rules of the loop, every block of the *Subsystem* knows the relevant information of their outgoing edges. Moreover, the blocks connected to the *Outport* blocks know which blocks are connected to the appropriate *Out* port as well. This means that the transformation can delete the edges inside the *Subsystem*, so if a block is moved out of it, then no dangling edges remains. The rule RW_MATLAB_FLATTENER_DELETEEDGES simply does that (i.e., deletes the edges between the blocks of the *Subsystem*).

In the next step the transformation deals with the blocks connected to *In* ports. The transformation rule RW_MATLAB_FLATTENER_INBLOCK is shown in Fig. 5. It matches the incoming edges of *Subsystem* nodes and if a match is found deletes them. As it has been mentioned, the RW_MATLAB_FLATTENER_GETALLEDGES rule matches *Inport* blocks as well, so these elements know the necessary information about their edges. In this manner the current rule can copy this information to the blocks connected to the appropriate *In* ports. The information is also extended with the port number of the edge directed from the block to the *Subsystem*. Upon completion of this rule the blocks connecting to the *Subsystem* know the characteristic of the edges the transformation must create after the *Subsystem* is deleted.

At this point the blocks related to the *Subsystem* store the necessary information about their outgoing edges. The transformation also deleted the edges connecting to the *Subsystem* and the ones between its blocks. This means that the blocks are ready to be moved to a higher hierarchical level. The higher hierarchical layer means the *Subsystem* containing the *Subsystem* that is actually processed, if there is any. In case there is not, then the root layer is the higher layer. The transformation rule RW_MATLAB_FLATTENER_PARENTLEVEL looks for the "parent" *Subsystem*. If it finds a match, then the blocks contained by the processed *Subsystem*, except the *In* and *Out* ones, are replaced into the parent. The rule is depicted in Fig. 6. If the processed *Subsystem* still contains elements besides the *Inport* and *Outport* blocks after this rule, then it means, that the

Subsystem is not contained by anything, it is at the root level. So the RW_MATLAB_-FLATTENER_ROOTLEVEL rule must delete only the *Containment* typed edges between the *Subsystem* and its blocks. In this manner the blocks are not contained by anything, they are moved to the root level as well.

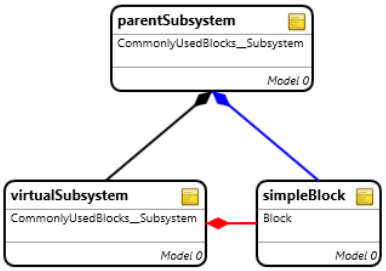


Fig. 6: The transformation rule RW_MATLAB_FLATTENER_PARENTLEVEL

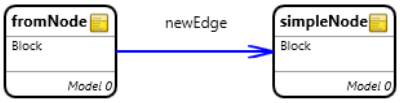


Fig. 7: The transformation rule RW_MATLAB_FLATTENER_CONNECTBLOCKS

Next, the transformation can safely delete the *Subsystem*. However, it cannot leave any dangling edges, so if it was contained by another *Subsystem*, then the transformation must delete the *Containment* edge after which the *Subsystem* is deleted.

Finally, the transformation must recreate the edges between the moved blocks. This is based on the information stored in the blocks. The transformation rule RW_MATLAB_FLATTENER_CONNECTBLOCKS creates the necessary edges (Fig. 7). The block storing the information is the source block; the identifier provides the target block. The appropriate ports are also saved along with other relevant information.

With this rule the loop ends and the transformation returns to the RW_MATLAB_-FLATTENER_TAGVIRTUALSUBSYSTEM transformation rule, which checks for another virtual *Subsystem*. If there is one, then the engine steps into the loop again, otherwise it terminates.

The next section discusses the formal analysis of this transformation.

6 Analysis of the Transformation

The previous section has presented the transformation TRANSFLATTENER and its rules. Before its usage, it is advisable to perform the analysis of the transformation definition, which is the subject of this section. First, the functionality of the transformation is examined and then its further attributes, such as correctness and termination, are verified.

The coverage of Proposition 1 and Proposition 2 is depicted in Fig. 8. This picture may also help illustrate the relation of the different blocks.

Definition 1. *The **inner elements** of a Subsystem are all elements, except the Inport and Outport typed blocks, contained by the Subsystem.*

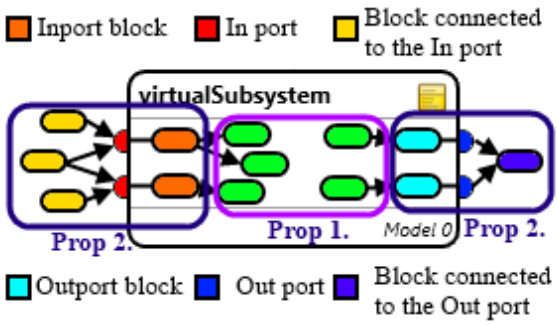


Fig. 8: The structure of a subsystem

Proposition 1. *After the transformation* TRANSFLATTENER*, the inner elements of the processed Subsystem are connected if and only if they were connected before the transformation.*

Proof. Three transformation rules (RW_MATLAB_FLATTENER_OUTBLOCK, RW_MATLAB_FLATTENER_GETALLEDGES and RW_MATLAB_FLATTENER_CONNECTBLOCKS are responsible for the connection between the blocks. First, the RW_MATLAB_FLATTENER_OUTBLOCK deletes the edges pointing to the different *Outport* blocks, which represent the *Out* ports of the *Subsystem*. The rule also stores the identifiers of the blocks that had an edge pointing from the same *Out* port. (The identifiers of these blocks are already stored in the *Out* block because of the RW_MATLAB_FLATTENER_GET_OUTEDGES rule.) Next, the RW_MATLAB_FLATTENER_GETALLEDGES stores for each block the identifier of those blocks that the given block has an edge pointing to. It also stores the details of the edges, that is, from which port point to which port. The rule examines every block exactly once. Next, the other rules of the transformation place the elements onto a higher level in the hierarchy. Since it is not possible in Simulink that elements in different hierarchy level are directly connected, it is ensured that after the replacing there is no edge pointing to or from the moved block. The transformation also deletes all edges between the inner elements to avoid the dangling edges. Finally, when the elements of the *Subsystem* are already placed onto a higher hierarchical level, the RW_MATLAB_FLATTENER_CONNECTBLOCKS creates the edges based on the stored information for each block. Since this is the only rule that generates edges and does this based on the stored information in the different blocks, there will be no new edges between the inner elements of the *Subsystem*. Note that the information about the edges is stored for each and every inner element that has outgoing edges, thus these edges will be regenerated.

In this manner the inner elements of the Subsystem are connected if and only if they were connected before the transformation execution. □

In order to not change the functionality of a Simulink model it is required that a block is reachable from another block if and only if it was reachable before the transformation.

Since the transformation cannot modify the functionality of a processed model, the following are required by the transformation:

- Let o denote the set of blocks outside of the *Subsystem* that have an outgoing edge to a given *In* port. Let i denote the set of inner elements that are connected to the *Inport* block related to the given *In* port. With this notation, after the transformation all elements in o must have an edge pointing to all elements in i.
- Regarding the *Out* ports we can state the same requirements. Let o denote the set of blocks outside of the *Subsystem* that have an incoming edge from a given *Out* port; and i denote the set of inner elements that are connected to the *Outport* block related to the given *Out* port. In this case, after the transformation all elements in i must have an outgoing edge to all elements in o.

Proposition 2. *After the transformation* TRANSFLATTENER *leaves the ports of the processed Subsystem the functionality of the Simulink model does not change.*

Proof. The RW_MATLAB_FLATTENER_GET_OUTEDGES and the RW_MATLAB_FLATTENER_OUTBLOCK transformation rules are responsible for not changing the functionality of the model when the *Out* ports are removed. First, the RW_MATLAB_FLATTENER_GET_OUTEDGES rule stores information in each *Out* block. This information contains the identifier of the blocks to which any edges point from the appropriate *Out* port. Moreover, the port attributes of these edges are also stored. The rule also deletes these edges. Next, the RW_MATLAB_FLATTENER_OUTBLOCK attempts to match those edges that are pointing from one of the inner elements of the *Subsystem* to one of the *Outport* blocks. For every match, the rule deletes the edge and copies the information stored in the *Outport* block to the matched element. It also extends this information with the number of the port from where the matched edge starts. With the help of these two transformation rules it is ensured that the blocks that have outgoing edges to one of the *Outport* blocks possess the proper information. The proper information means the identifiers of the block, where the edges from the appropriate *Out* port point to. The edges based on this information will be recreated by the RW_MATLAB_FLATTENER_CONNECTBLOCKS rule after the elements are placed to a higher hierarchical layer.

To ensure that the functionality of the model remains the same in case of the *In* ports, two rules are needed as well. These rules are the RW_MATLAB_FLATTENER_GETALLEDGES and the RW_MATLAB_FLATTENER_INBLOCK. As it was described in the proof of the Proposition 1, the RW_MATLAB_FLATTENER_GETALLEDGES rule stores for each and every block, so for the *Inport* blocks as well, to which port of which blocks it has outgoing edges. Next, the RW_MATLAB_FLATTENER_INBLOCK matches each edge that points from a block outside of the *Subsystem* to one of its *In* ports. The rule deletes this edge, since after deleting the *Subsystem* there cannot be any dangling edge. The rule also copies the stored information from the appropriate *Inport* blocks to the matched block and extends it with the number of the port the matched edge starts from. With the help of these two transformation rules it is ensured that the blocks, which have outgoing edges to one of the *In* ports, have the information, where the appropriate *Inport* block has outgoing edges. The edges based on this information

will be created by the RW_MATLAB_FLATTENER_CONNECTBLOCKS transformation rule after the elements are placed into a higher hierarchical layer.

It can thus be stated that after eliminating the ports of the Subsystem the functionality of the Simulink model does not change. □

Consequence of Proposition 2: The number of the edges in the Simulink model changes as follows: $\sum(-s_i - f_i + (s_i * f_i)) + \sum(-s_o - l_o + (s_o * l_o))$, where s_i stands for the number of edges going into the i^{th} *In* port of the *Subsystem*, f_i means the number of edges going out from the i^{th} *Inport* block, s_o stands for the number of edges going out from the o^{th} *Out* port of the *Subsystem* and l_o means the number of edges going into the o^{th} *Outport* block.

Proposition 3. *Proposition 1 and Proposition 2 together state that all inner elements of the processed Subsystem are connected if and only if they were connected before the iteration and the functionality of the model does not change. Considering the two propositions it can be stated, that the functionality of the model does not change after the Subsystem is flattened.*

Proof. The proof follows from the proofs of Proposition 1 and Proposition 2. □

Proposition 4. *Each iteration of the transformation* TRANSFLATTENER *moves the inner elements of the processed Subsystem exactly one level higher.*

Proof. The RW_MATLAB_FLATTENER_PARENTLEVEL and the RW_MATLAB_FLATTENER_ROOTLEVEL transformation rules are responsible for this behavior. The RW_MATLAB_FLATTENER_PARENTLEVEL attempts to find a match, where the processed *Subsystem* is a child element of another *Subsystem*. If such a match is found then the rule deletes the *Containment* edge between the processed *Subsystem* and its inner elements and also creates a *Containment* edge between the parent *Subsystem* and the aforementioned inner elements. If there is no match for this rule, then it can be stated that the processed *Subsystem* is at the root level. So the RW_MATLAB_FLATTENER_ROOTLEVEL rule simply deletes the *Containment* edges of the processed *Subsystem*. This means that its inner elements are moved to the root level. Neither of these two transformation rules match for the *Inport* and *Outport* blocks of the *Subsystem*. □

Definition 2. *The* execution hierarchical layers *are layers created not for purely graphical purpose, but have a bearing on execution semantics.*

This means that since the *virtual Subsystems* are created to help organize and understand the models and do not have any additional role, the *execution hierarchical layers* are created only by *nonvirtual Subsystems*.

Proposition 5. *The transformation* TRANSFLATTENER *does not move elements between execution hierarchical layers.*

Proof. The RW_MATLAB_TAGVIRTUALSUBSYSTEM is the first rule of the transformation. The rule attempts to match virtual *Subsystems*. At the beginning of each iteration, a virtual *Subsystem* is tagged as the *Subsystem* under processing. As a consequence, only the elements of a virtual *Subsystem* are modified during the actual iteration. As a consequence none of the objects of a non-virtual *Subsystem* are moved to a higher hierarchical level. □

Proposition 6. *The transformation* TRANSFLATTENER *flattens all virtual Subsystems.*

Proof. The RW_MATLAB_TAGVIRTUALSUBSYSTEM transformation rule is the first rule of the iteration: This rule expresses the loop condition. At each time the rule is evaluated, it attempts to match a virtual *Subsystem*. If such a match is found then the found *Subsystem* will be processed. The other rules of the iteration move the inner elements of this *Subsystem* onto a higher hierarchical layer and also delete this *Subsystem*. None of the rules creates any type of *Subsystems*. This means that every iteration decreases the number of virtual *Subsystems* by one in the model. The iteration continuous till the RW_MATLAB_TAGVIRTUALSUBSYSTEM cannot find a successful match anymore. This means there are no remaining virtual *Subsystems* in the model. □

Proposition 7. *The transformation* TRANSFLATTENER *always terminates.*

Proof. To examine the termination of the transformation the following must be checked:

- The control flow cannot go into an infinite loop,
- The transformation rules, which are applied exhaustively, terminate in finite steps.

The control flow terminates if the RW_MATLAB_TAGVIRTUALSUBSYSTEM rule cannot find a match. This happens if and only if there are no virtual *Subsystems* in the model. In case there is no virtual *Subsystem* in the model at the starting point, the transformation terminates without stepping into the iteration. Otherwise a match is found and the transformation steps into the loop. Proposition 6 states that the transformation flattens all virtual *Subsystem*. Since in Simulink the *Subsystems* cannot be recursively defined (i.e. the containment loops are forbidden) there are finitely many *Subsystems* in the model. This means that after finite number of iteration there will be no remaining virtual *Subsystems* in the model, so the RW_MATLAB_TAGVIRTUALSUBSYSTEM rule cannot find a successful match anymore, thus the control flow terminates.

In the transformation each rule is applied exhaustively except the RW_MATLAB_-TAGVIRTUALSUBSYSTEM rule. The exhaustively applied rules must be checked whether they terminate in finite steps:

- RW_MATLAB_FLATTENER_GET_OUTEDGES rule: This rule matches an edge between the *Out* port of a *Subsystem* and other blocks. If a match is found the rule deletes this edge. This means that every application of the rule reduces the number of edges between the *Out* ports and other blocks, thus after finites steps it cannot be applied anymore.
- RW_MATLAB_FLATTENER_OUTBLOCK rule: This rule works by the same principle. The only difference between the two rules is that this one attempts to match an edge between an inner element and the *Outport* block of the *Subsystem*. The deletion of the matched edge, without creating any new, ensures its termination.
- RW_MATLAB_FLATTENER_GETALLEDGES rule: This rule marks the matched block at each application. After several steps there is no remaining unmarked inner element in the *Subsystem*. Since there is a condition in the LHS of the rule that the block cannot be marked before the rule is applied, the system cannot find a match.
- RW_MATLAB_FLATTENER_DELETEEDGES rule: The rule simply matches an edge between the blocks of the *Subsystem* and if a match is found deletes it. After finite steps there are no edges left between the blocks, and the rule cannot be applied.

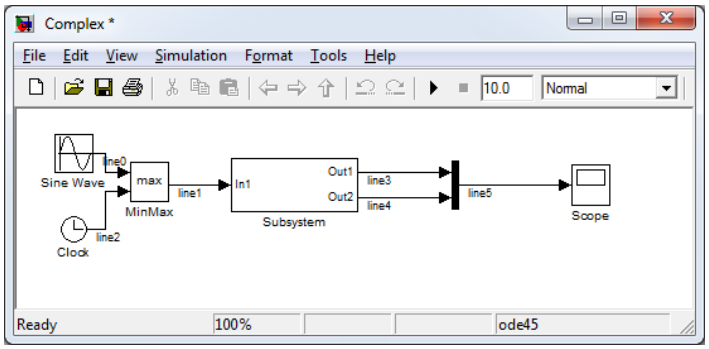
- RW_MATLAB_FLATTENER_INBLOCK rule: This rule is equivalent to the RW_-MATLAB_FLATTENER_GET_OUTEDGES, but this one operates between the *In* ports and the *Subsystem*. It deletes the matched edges as well, therefore cannot be applied after a certain number of steps.
- RW_MATLAB_FLATTENER_PARENTLEVEL rule: The rule matches and deletes the containment edge between an inner element and the actual *Subsystem*. It is irrelevant that it creates a new edge between the parent *Subsystem* and the inner elements, since the LHS of the rule checks containment edges between the actual *Subsystem* and other blocks. This ensures that the rule cannot be applied indefinitely.
- RW_MATLAB_FLATTENER_ROOTLEVEL rule: The rule simply deletes the containment edge between the actual *Subsystem* and its inner element. The deletion of the matched item without creating any new items ensures its termination.
- RW_MATLAB_FLATTENER_DELETE_CONTEDGE rule: The principle is the same. The rule attempts to match a containment edge between the actual *Subsystem* and a parent one. If it succeeds, then it deletes this edge.
- RW_MATLAB_FLATTENER_DELETE_SUBSYSTEM rule: The rule deletes the actual *Subsystem*. The LHS checks that the *Subsystem* must be marked. This marking occurs in the RW_MATLAB_TAGVIRTUALSUBSYSTEM rule, which is applied exactly once before every iteration. In this manner there is only one element that can be a match for this rule.
- RW_MATLAB_FLATTENER_CONNECTBLOCKS rule: The rule matches blocks that store information about edges to create. After the rule creates such an edge, it removes the information from the block. Since the RW_MATLAB_FLATTENER_-GETALLEDGES rule was applied for a finite number of times and this is the only transformation rule that stores information about the edges to create, the RW_-MATLAB_FLATTENER_CONNECTBLOCKS rule will be applied for a finite number of times as well.

Since both the control flow and the transformation rule terminate after finite number of steps, the transformation terminates as well. □

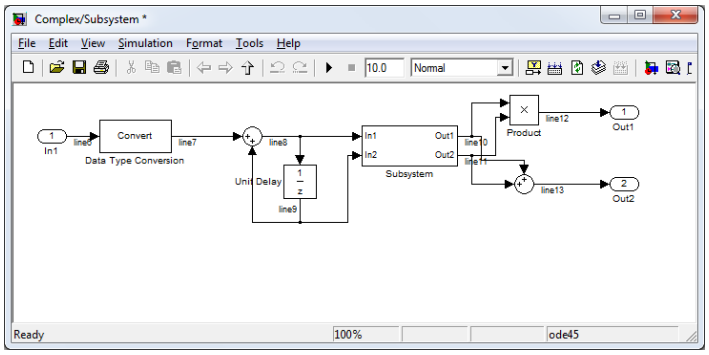
7 Experimental Results

The presented model transformation was applied on different Simulink models. One of these source models is depicted in Fig. 9. The root level is shown in Fig. 9(a). This model contains a virtual Subsystem with one In port and two Out ports. The inner structure of this Subsystem is shown in Fig. 9(b). It can be seen, that each Inport/Outport block relates to exactly one In/Out port. This hierarchical layer also contains a virtual Subsytem, which is presented in Fig. 9(c).

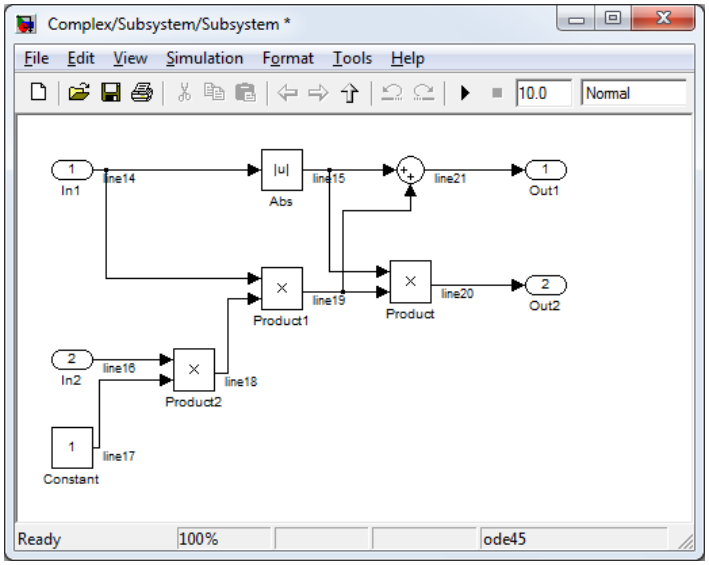
After the transformation TRANSFLATTENER terminates, the structure of the model changes, as it is shown in Fig. 10. All inner elements were moved to the next level, and eventually, since the model did not contain any non-virtual Subsystem, to the root level. The Subsystem blocks were deleted with their Inport and Outport blocks. The connection within the blocks were correctly maintained. The example also demonstrates that the transformation handles well when an Out port of a virtual Subsystem is connected



(a) The root level of the Simulink® model



(b) The model contained by the fist Subsystem



(c) The containment of the nested Subsystem

Fig. 9: The example Simulink® model

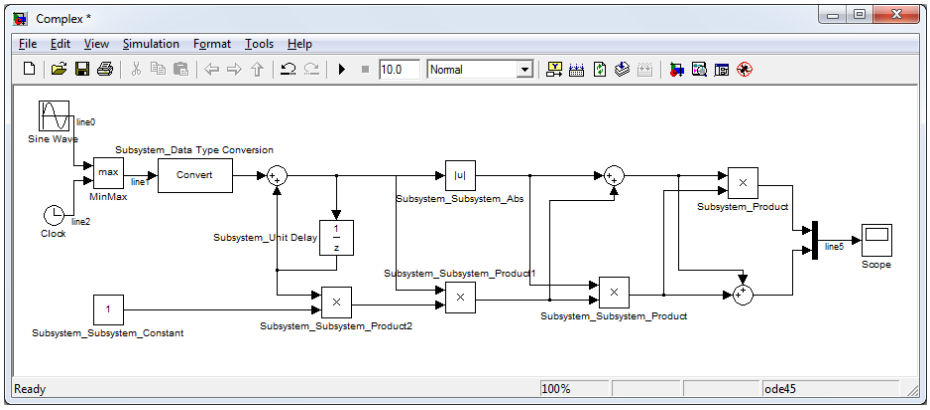


Fig. 10: The model after the TRANSFLATTENER transformation

to multiple blocks. In this manner, the transformation did not change the functionality of the source model.

The transformation was examined on simpler and more complex models as well, and the results always were found to be correct by inspection.

8 Conclusions and Future Works

As a popular tool for the design of embedded control systems, industry relies on Simulink models to support a level of abstraction much above the embedded implementation in, for example, C code. Design relies heavily on model elaboration to increasingly add detail to the design models. Such elaboration is a form of model transformation that currently is implemented in software as part of the Simulink code base or as external functionality based on the Simulink model API.

Part of the elaboration is removing hierarchical structures that have only a syntactic effect such as flattening of syntactic hierarchical layers. In this paper a detailed model transformation-based solution has been presented for flattening virtual subsystems in Simulink models. The approach enables taking advantage of benefits of modeled model transformation such as reusability and platform independence. In this manner, the abstraction level of the model transformation problem can be raised. Besides the transformation details, its formal analysis has been also discussed.

The transformation was implemented in the Visual Modeling and Transformation System. Therefore the modeling framework and its communication with the Simulink environment were briefly introduced as well.

Future work intends to study whether with the help of this transformation, the sorted list and the execution list can also be implemented via model transformation. In this manner the abstraction level could be raised even further and more benefits unlocked.

9 Acknowledgments

This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TÁMOP-4.2.2.C-11/1/KONV-2012-0013).

References

1. PCAST document. http://varma.ece.cmu.edu/InfoCPS/Readings.html, 2007.
2. An introduction to P/Invoke and marshaling on the Microsoft .NET compact framework. http://msdn.microsoft.com/en-us/library/aa446536.aspx, 2012.
3. MATLAB® user's guide. http://www.mathworks.com/help/matlab/index.html, Sep 2012.
4. Simulink®. http://www.mathworks.com/simulink/, 2012.
5. Simulink® user's manual. http://www.mathworks.com/help/simulink/index.html, 2012.
6. Stateflow® user's guide. www.imec.be/elela/HK19/background/stateflow_users_guide.pdf, 2012.
7. VMTS website. http://vmts.aut.bme.hu/, 2012.
8. A. Agrawal, G. Simon, and G. Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. *Electron. Notes Theor. Comput. Sci.*, 109:43–56, dec 2004.
9. R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *THEORETICAL COMPUTER SCIENCE*, 138:3–34, 1995.
10. L. Angyal, M. Asztalos, L. Lengyel, T. Levendovszky, I. Madari, G. Mezei, T. Mészáros, L. Siroki, and T. Vajk. Towards a fast, efficient and customizable domain-specific modeling framework. In *Software Engineering*. ACTA Press, 2009.
11. R. C. Ansgar Bredenfeld. Tool integration and construction using generated graph-based design representations. In *Design Automation, 1995. DAC '95. 32nd Conference on*, pages 94–99, 1995.
12. M. Asztalos and I. Madari. An improved model transformation language. In *Automation and Applied Computer Science Workshop 2009*, 2009.
13. U. Aßmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In *Compiler Construction (CC)*, pages 121–135. Springer, 1996.
14. P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating discrete-time simulink to lustre. In *In: Third International ACM Conference on Embedded Software, Lecture Notes in Computer Science*, pages 84–99. Springer, 2003.
15. F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 603–612, New York, NY, USA, 2008. ACM.
16. P. Fehér, P. J. Mosterman, T. Mészáros, and L. Lengyel. Processing simulink models with graph rewriting-based model transformation. Model Driven Engineering Languages and Systems (MODELS '12) - Tutorials, 2012.
17. M. Fowler. *Domain Specific Languages*. The Addison-Wesley Signature Series. Addison-Wesley, 2010.
18. S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008.
19. T. Mens, K. Czarnecki, and P. V. Gorp. A taxonomy of model transformation. In *Proc. Dagstuhl Seminar on "Language Engineering for Model-Driven Software Development". Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl*. Electronic, 2005.

20. P. J. Mosterman, J. Ghidella, and J. Friedman. Model-based design for system integration. In *The Second CDEN International Conference on Design Education, Innovation, and Practice*, pages TB3–1–TB3–10, 2005.
21. P. J. Mosterman, S. Prabhu, and T. Erkkinen. An industrial embedded control system design process. In *Proceedings of The Inaugural CDEN Design Conference (CDEN'04)*, pages 02B6–1–02B6–11, 2004.
22. P. J. Mosterman, J. Sztipanovits, and S. Engell. Computer-automated multiparadigm modeling in control systems technology. *Control Systems Technology, IEEE Transactions on*, 12(2):223–234, march 2004.
23. P. J. Mosterman and H. Vangheluwe. Computer automated multi-paradigm modeling in control system design. *IEEE Transactions on Control System Technology*, 12:65–70, 2000.
24. P. J. Mosterman and H. Vangheluwe. An introduction to computer automated multi-paradigm modeling, 2004.
25. T. Mészáros, G. Mezei, and T. Levendovszky. A flexible, declarative presentation framework for domain-specific modeling. In *Proceedings of the working conference on Advanced visual interfaces*, AVI '08, pages 309–312, New York, NY, USA, 2008. ACM.
26. H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 440–455, Berlin, Heidelberg, 2009. Springer-Verlag.
27. G. Nicolescu and P. Mosterman. *Model-Based Design for Embedded Systems*. Computational Analysis, Synthesis, and Design of Dynamic Models Series. CRC Press, 2010.
28. A. Radermacher. Support for design patterns through graph transformation tools. In *In Applications of Graph Transformation with Industrial Relevance (Intl. Workshop AGTIVE'99, Proceedings), LNCS 1779*, pages 111–126. Springer, 1998.