CREATING SORTED LIST FOR SIMULINK MODELS WITH GRAPH TRANSFORMATION

Péter Fehér [(a)], Tamás Mészáros [(a)], Pieter J. Mosterman [(b)] and László Lengyel [(a)]

[(a)] Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Budapest, Hungary
[(b)] Research and Development
MathWorks
Natick, MA, USA

[(a)]{feher.peter, mesztam, lengyel}@aut.bme.hu, [(b)]pieter.mosterman@mathworks.com

ABSTRACT
Embedded systems are usually modeled to simulate their behavior and facilitate design space exploration. Nowadays, this modeling is often implemented in the Simulink® environment, which offers strong support for modeling complex systems. 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 semantically relevant model elements. Thus, to create an implementation or to execute a model, Simulink generates the dependency list of the model entities, which is referred to as the Sorted List. The work presented in this paper raises the level of abstraction of the model elaboration by modeling the Sorted List generation in order to unlock the potential for reuse, platform independence, etc. The transformation is implemented by applying graph transformation methods. Moreover, an analysis of the transformation is also provided.

Keywords: embedded systems, model-based design, model transformation, Simulink

1. INTRODUCTION

Nowadays a growing amount of software is modeled during the development phase. This is especially true with respect to the embedded systems. By modeling these systems, they can be examined based on, for example, their functionality, performance or robustness.

Selecting the most appropriate framework for modeling systems requires a remarkable amount of attention. Recently, domain-specific modeling has become a popular approach to describe complex systems. It is a powerful, but still understandable technique, the main strength of which lies in the application of domain-specific languages. Since domain-specific languages are specialized for a certain application domain, their application is more efficient than that of general purpose languages (Fowler 2010, Kelly and Tolvanen 2008).

MATLAB® (Matlab 2012) and Simulink® (Simulink 2012a) have undoubtedly become some of the leading tools for model-based system design and synthesis in the past years (Mosterman, Prabhu, and Erkkinen 2004, Mosterman and Vangheluwe 2002, Nicolescu and Mosterman 2010). As many other modeling tools, Simulink also offers the possibility to simulate the modeled system enabling thus to examine the behavior of the system before realizing it.

In order to simulate the system under design, Simulink must perform numerous preprocessing steps on the model. An important step of this preprocessing phase is inferring the execution order of the entities used in the model. This execution order is referred to as the *Execution List*. To establish the Execution List, Simulink must determine the relationships between the blocks. This is the aim of generating the so called *Sorted List* that constitutes a dependency list (Fehér et al. 2012, Simulink 2012b). That is, the Sorted List contains the elements of the modeled system in a specific order based on the control and data dependencies that determine how the different blocks can follow each other in the overall execution.

At present, generating the Sorted List is implemented in the Simulink code base. Though efficient, this makes difficult or even prevents unlocking value for which a higher level of abstraction is more appropriate (e.g., reasoning about the implementation and modularization of operations). Therefore, by implementing the Sorted List generation procedure at a higher level of abstraction, the advantageous properties of domain-specific modeling can be utilized. This is a fundamental premise of Computer Automated Multiparadigm Modeling; to use the most appropriate formalism for representing a problem at the most appropriate level of abstraction (Mosterman and Vangheluwe 2004, Mosterman, Sztipanovits, and Engell 2004).

This paper focuses on a novel solution to generate the Sorted List for Simulink models. This approach is based on a model transformation created in the Visual Modeling and Transformation System (VMTS) framework. VMTS has been prepared to communicate with the Simulink environment; therefore the model transformations designed in VMTS can be directly applied to the various Simulink models. Using model transformation to solve the Sorted List generation issue helps to raise the abstraction level from the API programming to the level of software modeling. The solution possesses all advantageous characteristics of model transformations, such as transparency, reusability and platform independence.

The rest of the paper is organized as follows. Section 2 presents the algorithm used for generating the Sorted List. Next, Section 3 introduces the implemented transformation. In Section 4, the properties of the transformation are examined. A simple example for the transformation is presented in Section 5. Finally, concluding remarks are elaborated.

2. CREATING A SORTED LIST

As it was previously described, Simulink creates the Sorted List based on the dependency of the elements. Previous work has captured in detail the dependencies that Simulink accounts for (Han and Mosterman 2010). A block *b* depends on the block *a* if the *direct feedthrough* (DF) property (Simulink 2012c) on its inport block obtaining the signal from *a* is set to *true*. In this case block *a* must appear before block *b* in the Sorted List. Else, there is no dependency, that is, the output of the block *b* with the input port can be computed without knowing the value on the input port. In this manner, there generally are many lists that satisfy the dependencies and it does not make any difference in semantics which list is selected.

At this phase of the processing, Simulink has already flattened the virtual subsystems of the model, therefore only nonvirtual subsystems left. Nonvirtual subsystems are, for example, the *Enabled Subsystem*, *Triggered Subsystem*, *Atomic Subsystem*, *Function-call Subsystem*, etc. These nonvirtual subsystems have their own Sorted List with the same principle discussed above.

The nonvirtual subsystems are treated as opaque blocks (in terms of execution) at the hierarchical level where they are used and so the input ports of a nonvirtual subsystem also have the DF attribute. The setting of this attribute is inferred from the content of the nonvirtual subsystem. Generally, for each input port of the nonvirtual subsystem the DF is set to be same as the DF attribute of the first block that the input connects to internally (to the nonvirtual subsystem). Note that a more sophisticated analysis may be applied to resolve some DF issues, as presented in other work (Mosterman and Ciolfi 2004, Denckla and Mosterman 2004).

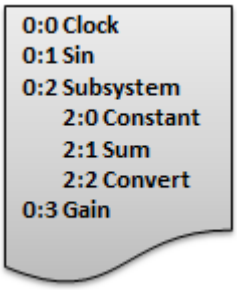


Figure 1: An example Sorted List

The Sorted Lists have hierarchical layering, as Figure 1 depicts. The $0 : x$ says that x is the position of the block in the Sorted List for the 0 hierarchical layer (with 0 being the top). Similarly, $2 : y$ says that y is the position of the block in the Sorted List for the 2 hierarchical layer, which is chosen the same as the position of the block with the hierarchical layering in its parent's Sorted List. For example, $2 : 0$ *Constant* says that the *Constant* block is at the first position for the 2 hierarchical layer, which relates to the *Subsystem* element.

In this section the algorithms for creating a Sorted List are presented as well as the complexity of the entire process.

2.1. The Algorithms

The main part of the Sorted List generation algorithm is shown in Algorithm 1. The SL algorithm contains a simple *Repeat-Until* block with only three algorithms. These three algorithms are responsible for processing the blocks.

Algorithm 1 The algorithm of the transformation SL

```
1: procedure SL()
2: repeat
3:     ProcessFirstBlocks(null)
4: until not ProcessSubsystem(null) and not ProcessNormalBlocks(null)
5: return
```

The three algorithms differ from each other in the type of block that they processing. First, the ProcessFirstBlocks algorithm processes only blocks without any incoming edge. Since without incoming edges a block cannot depend on any other block, the processing of these blocks does not need the examination of the DF properties. The ProcessFirstBlocks algorithm is presented in Algorithm 2.

Algorithm 2 The algorithm of processing blocks without incoming edges

```
1: procedure ProcessFirstBlocks(Subsystem sub)
2: while ProcessBlock(sub, true) do
3:     DeleteBlocksAndEdges()
4: return
```

The ProcessFirstBlocks algorithm obtains a parameter, which sets the actual subsystem the blocks are being processed within. In case the current

hierarchical level is the root of the model, then the parameter is *null*. The PROCESSFIRSTBLOCKS algorithm contains a *while* block. The condition of this loop is the return value of the PROCESSBLOCK algorithm, which is responsible for the processing itself. In case there was at least one block without incoming edges, the PROCESSBLOCK returns *true* and the DELETEBLOCKSANDEDGES algorithm is called. Since the DELETEBLOCKSANDEDGES deletes the processed blocks and their edges, the next time the PROCESSBLOCK algorithm is called, there may be new blocks without incoming edges. If there is no such a block, the algorithm terminates.

The PROCESSBLOCK algorithm is shown in Algorithm 3. It has two parameters: the *sub* sets the actual subsystem, while the *onlyFirsts* parameter determines whether only the blocks without incoming edges should be processed. Since each nonvirtual subsystem is basically an opaque block for execution purposes, it has its own Sorted List. Therefore, if this parameter is set to a subsystem, then only those blocks should be processed that are contained by this subsystem. This containment can be checked by the examination of the *Parent* property of the block to be processed. The algorithm only processes "simple" blocks, it is referred to blocks that are not composite elements. Moreover, the *Inport* and *Outport* blocks of the subsystems should not be processed, therefore, they are not part of the *SimpleBlock* set.

Algorithm 3 The algorithm of block processing

```
1: procedure ProcessBlock(Subsystem sub, Boolean onlyFirsts)
2: Boolean processedAny ← false
3: Block parent ← null
4: if sub not null then
5:    parent ← sub
6: for all b|b ∈ SimpleBlocks ∧ b.Parent = parent ∧ b.Tag ≠ "Processed" do
7:    if (onlyFirsts ∧ b.Sources = ∅) ∨ (¬onlyFirsts ∧ ∄e|e ∈ b.Source ∧ e.DF = true) then
8:       print CalculateIndent() + b.Name
9:       b.Tag = "Processed"
10:      if not processedAny then
11:         processedAny ← true
12: return processedAny
```

If the *onlyFirsts* parameter is set to *true*, the algorithm processes only the blocks without any incoming edge, that is, the *Sources* property is empty. Otherwise, if the *onlyFirsts* parameter is *false*, the algorithm must check if the DF properties on the port of the incoming edges are set to *false*.

The actual processing of a block is straightforward. The CALCULATEINDENT method determines the actual indent and position of the block. These values depend on the hierarchical level and the number of previously processed blocks. The name of the block is appended to the calculated value. After the list is maintained the algorithm depicts the block as "Processed". If the algorithm processed at least one block, then it returns *true*.

As it was mentioned before, the DELETEBLOCKSANDEDGES algorithm deletes the processed blocks and its edges. It is depicted in Algorithm 4.

Algorithm 4 The algorithm of deleting processed elements

```
1: procedure DeleteBlocksAndEdges()
2: for all b|b ∈ Blocks ∧ b.Tag = "Processed" do
3:    for all e ∈ b.InEdges do
4:       e.Delete()
5:    for all e ∈ b.OutEdges do
6:       e.Delete()
7:    b.Delete()
8: return
```

After there are no "simple" blocks left in the model without incoming edges, the SL algorithm moves on to process possible subsystems. This is achieved by the PROCESSSUBSYSTEM algorithm shown in Algorithm 5. Since it is possible in Simulink that a subsystem contains another subsystem, the algorithm obtains the current subsystem as a parameter. If it is set to *null* then the algorithm looks for a subsystem on the root level. The return value of the CHECKFORSUBSYSTEM algorithm determines if there is any processable subsystem on the given hierarchical level. If there is any, then it is stored in the *newSub* variable, and processed in a *repeat-until* loop.

Algorithm 5 The algorithm of processing a Subsystem

```
1: procedure ProcessSubsystem(Subsystem sub)
2: Subsystem actualSub ← sub
3: Subsystem newSub ← null
4: Boolean processedAny ← false
5: if CheckForSubsystem(actualSub, newSub) then
6:    repeat
7:       processedAny ← true
8:       DeleteBlocksAndEdges()
9:       ProcessInOutPorts(newSub)
10:      ProcessFirsts(newSub)
11:   until not ProcessSubsystem(newSub) and not ProcessNormals(newSub)
12:   newSub.Tag = "Processed"
13: return processedAny
```

This loop is the same as the one in the SL algorithm but it has three additional commands. First, this algorithm sets the *processedAny* variable to *true*, which will be the return value. Next, it calls the DELETEBLOCKSANDEDGES to delete the already processed elements if there are any. After this, the PROCESSINOUTPORTS algorithm deletes the *Inport* and *Outport* blocks and the edges connecting to them. This is necessary, since the Sorted Lists do not contain these blocks. After these steps, the PROCESSSUBSYSTEM algorithm processes the blocks in the same manner as the SL algorithm. As it can be seen in the Algorithm 5, the PROCESSSUBSYSTEM algorithm can be called recursively, in case the subsystem contains another subsystem. The algorithms are called with the *newSub* parameter as the current hierarchical level.

In order to determine if there is any processable Subsystem on the current hierarchical level, the CHECKFORSUBSYSTEM algorithm is used. The

algorithm is shown in Algorithm 6. A Subsystem is processable, if it is contained by the appropriate parent element and does not have any input port with the DF property set to *true*. In case the algorithm finds such a Subsystem, then it assigns the Subsystem to the *newSub* variable and returns *true*. Otherwise, the return value is *false*.

Algorithm 6 The algorithm of checking processable Subsystem

```
1: procedure CHECKFORSUBSYSTEM(Subsystem actSub, Subsystem newSub)
2: if ∃s|s ∈ Subsystems ∧ s.Parent = actSub ∧ ∄e|e ∈ s.Sources ∧ e.DF = true then
3:     print CALCULATEINDENT() + s.Name
4:     newSub ← s
5:     return true
6: return false
```

As it was mentioned before, the *Inport* and *Outport* blocks of the Subsystem are not part of the Subsystem. Therefore, their outgoing edges should be deleted without processing the blocks, as it is depicted in Algorithm 7. Note, that a nonvirtual Subsystem is only processable if none of its *In* ports has its DF property set to *true*. In case of virtual Subsystems this restriction does not hold, but the virtual Subsystems must be flattened before the processing of the Sorted List begins. This makes it possible to delete all edges connected to the *Inport* blocks.

Algorithm 7 The algorithm of processing edges related to *In*- and *Outport* blocks

```
1: procedure PROCESSINOUTPORTS(Subsystem sub)
2: for all b|b ∈ InBlocks ∧ b.Parent = sub do
3:     for all e ∈ b.OutEdges do
4:         e.Delete()
5: for all b|b ∈ OutBlocks ∧ b.Parent = sub do
6:     for all e ∈ b.InEdges do
7:         e.Delete()
8: return
```

The PROCESSNORMALS algorithm (Algorithm 8) is called from both *repeat-until* blocks. This algorithm is responsible for processing those blocks that have an arbitrary number of incoming edges but none of its input ports has a DF property with a value of *true*. It is similar to the PROCESSFIRSTBLOCKS algorithm but calls the PROCESSBLOCK algorithm with different parameter.

Algorithm 8 The algorithm of processing blocks with incoming edges

```
1: procedure PROCESSNORMALS(Subsystem sub)
2: if PROCESSBLOCK(sub, false) then
3:     DELETEBLOCKSANDEDGES()
4:     return true
5: return false
```

Note that both *until* blocks (in SL and PROCESSSUBSYSTEM) have the same condition. That is, the algorithm stays in the loop if either the PROCESSSUBSYSTEM or the PROCESSNORMALS algorithm returns *true*. In other words this means that either of these two algorithms processes at least one element. In this case, after the deletion of the related edges, there may be new, processable blocks. Otherwise, if there are no Subsystems and no "simple" blocks to process, then either all elements have been processed or there is an algebraic loop in the current hierarchical level. The SL algorithm terminates if this condition is met in the root level, that is, no elements are left in the model or there is an algebraic loop.

This section has presented the SL algorithm, which was designed to create a Sorted List from the input model. Next, the complexity of the algorithm will be determined.

2.2. Complexity Analysis

In order to use an algorithm in production applications its complexity must be established. In this section the algorithms are examined based on their execution time. Therefore, the attributes that determine their computational complexity must be determined.

To determine the complexity of the SL algorithm, the following algorithms must be examined (the rest of the algorithms call these ones):

- PROCESSBLOCK
- DELETEBLOCKSANDEDGES
- CHECKFORSUBSYSTEM
- PROCESSINOUTPORTS

The PROCESSBLOCK algorithm is called from the PROCESSFIRSTBLOCKS and the PROCESSNORMALS algorithms. This is the one and only algorithm that processes "simple" blocks. On the one hand, since the Sorted List must contain all of these blocks, the body of the PROCESSBLOCK algorithm is executed at least n_{sb} times, where n_{sb} means the number of "simple" block on the current hierarchical level. On the other hand, since each processed block is deleted by the DELETEBLOCKSANDEDGES algorithm, the PROCESSBLOCK algorithm is executed at most n_{sb} times. Assume, that the complexity of the CALCULATEINDENT method is O(*1*). In this manner, the complexity of the PROCESSBLOCK algorithm is O(n_{sb}).

The DELETEBLOCKSANDEDGES algorithm deletes all processed elements and their edges. Based on the previous reasoning, all "simple" blocks are processed, therefore the *for* loop in the algorithm is executed at least *sb* times. However, the algorithm is also called after a Subsystem is processed. Let *n* denote the sum of the "simple" blocks and the Subsystem blocks. In this case the aforementioned *for* loop is run exactly *n* times. At each iteration the algorithm deletes the related edges as well. Assume, that the complexity of a deletion is O(1). In this manner, the complexity of the algorithm is O($n*(e_i+e_o)/2$), where e_i represents the average number of incoming edges to a processed element and e_o represents the average number of outgoing edges from a processed element. Let e_s denote all edges connected to either a "simple" block or a Subsystem. In this manner, the complexity of the DELETEBLOCKSANDEDGES algorithm is O($n+e_s$), every element and edge is deleted exactly once.

The CHECKFORSUBSYSTEM algorithm simply examines whether there is a processable Subsystem. If there is, then the algorithm assigns it to the *newSub* variable, and adds it to the Sorted List with the help of the CALCULATEINDENT method. Therefore, the complexity of the algorithm is $O(n_s)$, where n_s means the number of Subsystems in the model.

Finally, the PROCESSINOUTPORTS algorithm deletes all edges connected to the *Inport* or *Outport* blocks. In this case the complexity of the algorithm is $O(e_{io})$, where e_{io} represents the edges connected to the *Inport* or *Outport* blocks.

To summarize, the complexity of the SL algorithm is $O(n_{sb}+n+e_s+n_s+e_{io})$, which is equal to $O(2*n+e_s+e_{io})$. Let e denote all edges in the model, that is, the sum of e_s and e_{io}. In this manner, the complexity of the algorithm is $O(2*n+e)$, that is, each block ("simple" or Subsystem) is processed exactly once and all edges with the processed blocks are deleted.

3. IMPLEMENTING THE ALGORITHM

The previous section presented an algorithm that creates a Sorted List from a Simulink model. This section provides a novel approach to realize this algorithm: the algorithm is implemented via graph transformation.

With this approach, the solution utilizes the strong mathematical background of the graph rewriting-based model transformation. Moreover, the result possesses all the advantageous properties of the model transformation, for example, it is reusable, transparent and platform independent.

3.1. The Modeling Environment

To create transformations of a Simulink model, the Visual Modeling and Transformation System (VMTS) (Angyal, et al. 2009, VMTS 2012) was used. The 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. VMTS is also a transformation system. It utilizes a graph rewriting-based model transformation approach or a template-based text generation. Whereas templates are used mainly to produce textual output from model definitions in an efficient way, 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 depicted together. In this manner, the process of the transformation is more expressive. VMTS applies different colors to distinguish the LHS from the RHS in the presentation layer. Imperative constraints can also be applied.

In VMTS a control flow determines the order of the transformation rules. Each controls flow has exactly one start state and one or more end states. The applicable rules are defined in the rule containers. This means that exactly one rule belongs to each rule container. The application number of the rule can be defined here as well. By default, the VMTS attempts to locate 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 exists within the model.

The edges are used to determine the sequence of the rule containers. The control flow follows an edge, which corresponds to the result of the rule application. In VMTS, the edge to be followed in case of a successful rule application is depicted with a solid gray flow edge, 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.

3.2. The Transformation

As it was mentioned, a control flow determines the application order of the rules. The control flow of the TRANS_SL transformation is shown in Figure 2.

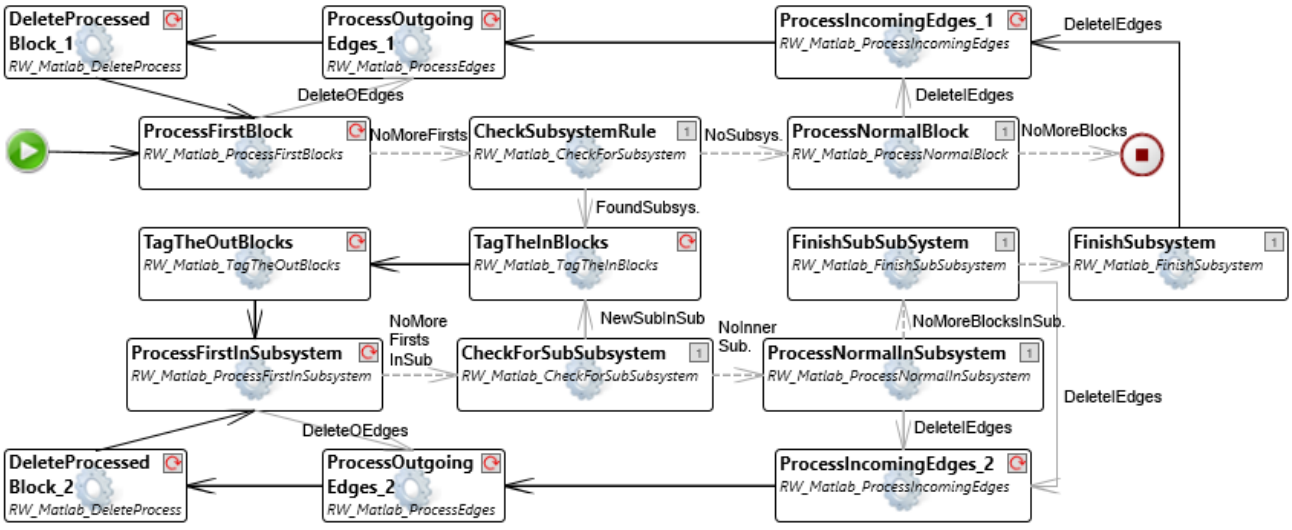


Figure 2: The control flow of the TRANS_SL transformation

At first, the transformation attempts to apply the RW_MATLAB_PROCESSFIRSTBLOCKS rule. This transformation rule matches for "simple" blocks, which have no incoming edges, therefore they do not depend on any other block. If a match is found, then the imperative code part of the rule writes out the name of the block with the necessary indentation, hierarchical level, and positioning. In this manner, the CALCULATEINDENT method of the PROCESSBLOCK algorithm is implemented via imperative code.

The rule is applied exhaustively, that is, as long the transformation finds an unprocessed "simple" block without incoming edges, and each processed block is tagged as "Processed". In case there was at least one successful match, the transformation moves to PROCESSOUTGOINGEDGES_1 rule container, which applies the RW_MATLAB_PROCESSEDGES rule (depicted in Figure 3). This rule attempts to find matches of the outgoing edges from the already processed elements and deletes them.

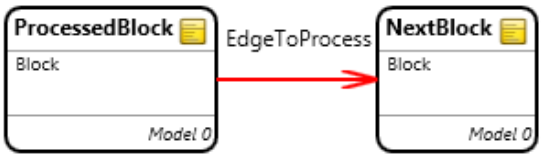


Figure 3: The transformation rule RW_MATLAB_-PROCESSEDGES

After there are no edges left to delete, the RW_MATLAB_DELETEPROCESSED transformation rule (shown in Figure 4), which is contained by the DELETEPROCESSEDBLOCK_1 rule container, deletes the appropriate elements.

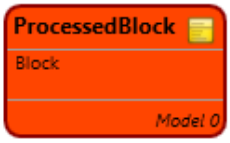


Figure 4: The transformation rule RW_Matlab_-DeleteProcessed

These three transformation rules are applied exhaustively, this way imitating the behavior of a *foreach* or *while* loop. The RW_MATLAB_PROCESSFIRSTBLOCKS implements the PROCESSBLOCK algorithm with the parameter list of (*null*, *true*); and the RW_MATLAB_PROCESSEDGES with the RW_MATLAB_DELETEPROCESSED correspond to the DELETEBLOCKSANDEDGES algorithm. Although the transformation does not delete any incoming edges related to the processed elements unlike the DELETEBLOCKSANDEDGES algorithm, this is not necessary here, since these blocks do not have any incoming edges. Moreover, since the transformation moves to the RW_MATLAB_PROCESSFIRSTBLOCKS after deleting the already processed elements, the *while* loop of the PROCESSFIRSTBLOCKS algorithm is implemented as well.

When the application of the RW_MATLAB_PROCESSFIRSTBLOCKS transformation rule was unsuccessful, the transformation moves along the dashed grey line of the control flow, which leads to the RW_MATLAB_CHECKFORSUBSYSTEM rule. As it is shown in Figure 5, the rule attempts to find a match for a Subsystem, an Inport block and an ordinary Block. In Simulink, the DF property is an attribute of the In port of the blocks. However, in VMTS, this property is moved, and is a characteristic of the edges. In this manner, a Subsystem is processable, if the DF property of the *FirstEdge* edge (the edge starting from the Inport block) is set to *false*. In case such a Subsystem is found, it is written out with the help of the imperative code, and the transformation starts processing its elements.

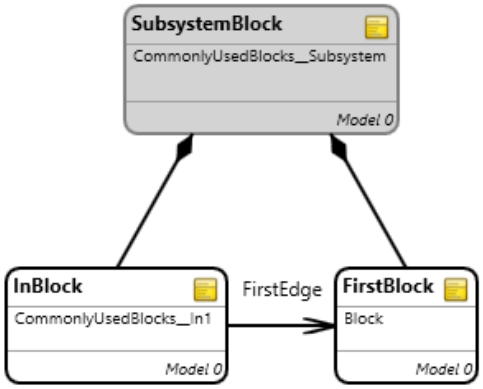


Figure 5: The transformation rule RW_MATLAB_-CHECKFORSUBSYSTEM

This processing starts with deleting the edges connected to the Inport and Outport blocks. These deletions are accomplished in the RW_MATLAB_TAGTHEINBLOCKS (depicted in Figure 6) and RW_MATLAB_TAGTHEOUTBLOCKS, respectively. The first rule may create blocks without any incoming edge, which means they are independent from any other block. The latter is only necessary to avoid any dangling edges after the transformation terminates.

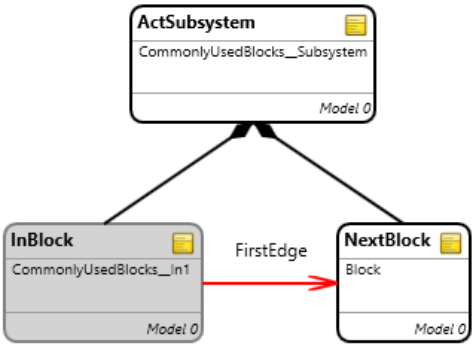


Figure 6: The transformation rule RW_MATLAB_-TAGTHEINBLOCKS

The next three transformations are similar to the first three. The RW_MATLAB_PROCESSFIRSTINSUBSYSTEM rule (contained by the PROCESSFIRSTINSUBSYSTEM rule container), which is shown in Figure 7, processes the blocks without incoming edges. These blocks must be contained by the found Subsystem element, which is the only difference between this rule and the RW_MATLAB_PROCESSFIRSTBLOCKS. In case the RW_MATLAB_PROCESSFIRSTSINSUBSYSTEM rule is applied at least once, the transformation executes the same rules to delete the edges and blocks as in the beginning of the process. The application of a new rule container is necessary though, since the transformation now must move to the RW_MATLAB_PROCESSFIRSTSINSUBSYSTEM instead of the RW_MATLAB_PROCESSFIRSTBLOCKS.

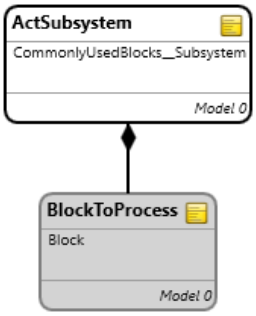


Figure 7: The transformation rule RW_MATLAB_PROCESSFIRSTSINSUBSYSTEM

When there is no other processable block without incoming edges, the transformation attempts to find a match for the RW_MATLAB_CHECKFORSUBSUBSYSTEM (presented in Figure 8) transformation rule. This rule basically attempts to find a Subsystem element inside the current one. Therefore, the applied rule is very similar to the already described one, the only difference is the presence of a parent Subsystem, which needs to be the current one.

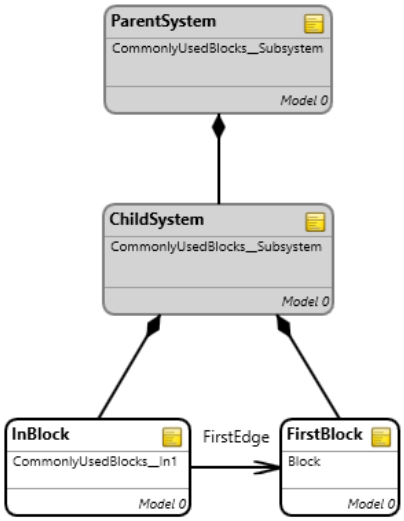


Figure 8: The transformation rule RW_MATLAB_CHECKFORSUBSUBSYSTEM

In case the transformation found a processable inner Subsystem, the transformation then starts to process with the already presented RW_MATLAB_TAGTHEINBLOCKS rule. This is essentially the implementation of the recursive CHECKFORSUBSYSTEM algorithms in a model transformation environment.

However, if there is no processable inner Subsystem, the transformation attempts to process the remaining blocks with the help of the RW_MATLAB_PROCESSNORMALINSUBSYSTEM rule. This rule is exactly the same as the RW_MATLAB_PROCESSFIRSTSINSUBSYSTEM but it allows the matched blocks to have incoming edges, if their DF properties are set to *false*. In case the transformation processed any block here, then it moves to the RW_MATLAB_PROCESSINCOMINGEDGES rule, which deletes the incoming edges of the processed elements. Next, the transformation returns to the RW_MATLAB_PROCESSEDGES rule. This structure corresponds to the *repeat-until* block of the CHECKFORSUBSYSTEM algorithm.

In case there are no blocks to process in the actual Subsystem, the transformation applies the RW_MATLAB_FINISHSUBSUBSYSTEM (shown in Figure 9) and RW_MATLAB_FINISHSUBSYSTEM rules. These are the exit points of the recursion and tag the found Subsystem as "Processed". If the first rule can be matched, then it means that there was an inner Subsystem and the current Subsystem must be set back to its parent. After this, the edges connected to this processed Subsystem and then the Subsystem itself is deleted. The transformation continues with the processing of the parent Subsystem. In case the second rule is matched, then it means the transformation returns to the root level again. The applied transformation rules are the same, but the transformation returns to the RW_MATLAB_PROCESSFIRSTBLOCKS instead of its equivalent rule in the Subsystem level.

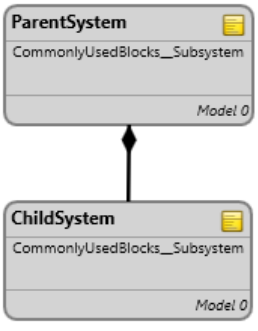


Figure 9: The transformation rule RW_MATLAB_FINISHSUBSUBSYSTEM

The transformation terminates when there is no processable element in the root level. This means that the RW_MATLAB_PROCESSNORMALBLOCK transformation rule does not find any match.

In this section the VMTS framework was briefly introduced. Moreover, a model transformation, which

implements the algorithms defined in Section 2.1, was also presented. During the presentation, the mechanisms of defining *foreach*, *while* loops, *repeat-until* blocks and recursion were introduced. The transformation is also a good example of how well the declarative and imperative approaches can complement and extend each other.

4. THE ANALYSIS OF THE TRANSFORMATION

As complexity analysis is essential for applying a new algorithm, the analysis of a model transformation is necessary before the transformation is included in a robust engine, which is the subject of this section. First, the functionality of the transformation is examined and then its further attributes, such as termination and correctness, are verified.

Definition 1. *The **simple elements** of a Simulink model are all elements, that are neither a composite element (e.g. Subsystems) nor a mandatory element of a composite element (e.g. Inport and Outport block of a Subsystem).*

Before the transformation is examined in detail, note, that the transformation rules can be divided into two parts. The first part consists of the following:

- RW_MATLAB_PROCESSFIRSTBLOCKS
- RW_MATLAB_PROCESSEDGES
- RW_MATLAB_DELETEPROCESSED
- RW_MATLAB_PROCESSINCOMINGEDGES
- RW_MATLAB_CHECKFORSUBSYSTEM
- RW_MATLAB_FINISHSUBSYSTEM
- RW_MATLAB_PROCESSNORMALBLOCK

These transformation rules process the simple elements and find Subsystems on the root level. The second group contains the following:

- RW_MATLAB_PROCESSFIRSTSINSUBSYSTEM
- RW_MATLAB_PROCESSEDGES
- RW_MATLAB_DELETEPROCESSED
- RW_MATLAB_PROCESSINCOMINGEDGES
- RW_MATLAB_CHECKFORSUBSUBSYSTEM
- RW_MATLAB_FINISHSUBSUBSYSTEM
- RW_MATLAB_PROCESSNORMALIN-SUBSYSTEM

The rules contained by the second group behave exactly the same way as the ones in the first, but they processes the elements in deeper levels. The transformation rules match the same pattern with the exception of matching a parent Subsystem as well. So the reason behind the existence of the rules in the second group is the need of parent matching.

By the examination of the transformation we assume there is no algebraic loop in the model.

Proposition 1. *After the transformation TRANS_SL, all simple elements and Subsystems of the Simulink model are processed, therefore they are contained by the Sorted List. These elements are processed exactly once.*

Proof: The RW_MATLAB_PROCESSFIRSTBLOCKS transformation rule processes all simple elements without incoming edges. Throughout the process, its imperative code implements the CALCULATEINDENT method, which inserts the block into the Sorted List in the appropriate format. Each processed element is marked as "Processed". Both the RW_MATLAB_PROCESSEDGES and RW_MATLAB_-DELETEPROCESSED rules match these marked blocks and delete their outgoing edges, moreover, the rules delete the blocks themselves. This may results in other blocks without incoming edges, therefore the RW_MATLAB_PROCESSFIRSTBLOCKS may be applicable again.

When these rules cannot be applied anymore, it means, that a nonvirtual Subsystem is in the way (which require special treatment) or there is a directed cycle in the model. If a Subsystem is found, the RW_MATLAB_CHECKFORSUBSYSTEM rule puts the element into the list and the transformation starts processing the elements of the Subsystem. This is achieved by the rules corresponding to the ones on the root level, therefore they are not discussed in more detail. After a Subsystem is processed, the transformation marks it as "Processed" and moves on to delete their edges, and finally the marked Subsystem as well. The processing of the model continues with the RW_MATLAB_PROCESSFIRSTBLOCKS again.

In case there is a directed cycle, the RW_MATLAB_PROCESSNORMALBLOCK looks for blocks which have none of their incoming edges marked with a DF property set to *true*. If the application of the rules was successful, then the rule inserts the found elements into the Sorted List and marks them as "Processed". In this case, the transformation moves on to the rules responsible for deleting the related edges. In case, however, the application of the rule is unsuccessful, then the transformation terminates. This means that either there are no elements left in the model, or there are no elements left without having at least one incoming edges with the DF property set to *true*. The first case means that the transformation successfully processed all simple elements of the model. However, the latter case means that there is an algebraic loop in the Simulink model. Since it was assumed, there is no algebraic loop in the source model, the transformation cannot come to this result. In this manner, the transformation terminates if and only if there is no element left to process. ■

Proposition 2. *The transformation TRANS_SL processes the elements of the Simulink model in an appropriate order, that is, a block **a** is always processed later, than a block **b**, on which **a** is depend.*

Proof: There are three rules on the root level, which insert elements into the Sorted List:

- RW_MATLAB_PROCESSFIRSTBLOCKS, which processes only simple blocks without any incoming edges. A block without incoming edges does not depend on any other blocks, therefore it can be placed into the Sorted List. If a block *a* had an incoming edge *e* with the DF property set to *true*, but this edge *e* has been deleted after processing its source block *b*, then it means, *a* is now processable, since its dependency has been processed already. In this manner, this rule cannot insert any block into the Sorted List, that has any unprocessed dependency.
- RW_MATLAB_CHECKFORSUBSYSTEM, which processes Subsystem elements without incoming edges, or Subsystem elements with incoming edges with DF property set to *false*. These conditions ensure that the Subsystem is processable at the moment, it has no unprocessed dependency.
- RW_MATLAB_PROCESSNORMALBLOCK, which has the same conditions:

1. The processed block has no incoming edges,
2. The processed block has incoming edges, but none of them has its DF property set to *true*.

In this manner, these rules never insert any block into the Sorted List, that has any unprocessed dependency. These rules have their pairs in the Subsystem level, which behave the same in this regard. This means, the transformation TRANS_SL processes the elements of the Simulink model in an appropriate order. ■

Proposition 3. *The Sorted List created by the transformation TRANS_SL is a valid Sorted List for the input Simulink model.*

Proof: Proposition 1 states that every simple element is processed by the transformation, and Proposition 2 state that the elements are processed in the right order. This means that the Sorted List created by the transformation TRANS_SL is a valid Sorted List of the model, since it contains all relevant elements in an appropriate order. ■

Proposition 4. *The transformation TRANS_SL always terminates.*

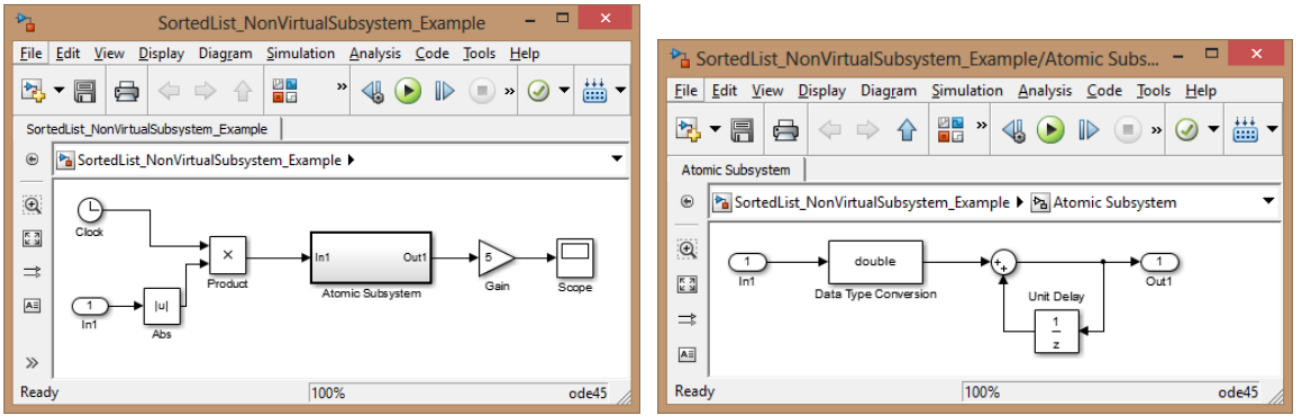
Proof: In order to prove the transformation always terminates, the following two statements have to be proved:

1. Each transformation rule is applied only a bounded number of times,
2. The transformation does not contain any infinite loop.

In VMTS the application mode of a transformation rule is set to either "Once" or "IsExhaustive". In case the rule is applied "Once" then after the transformation attempts to apply the rule, it moves on to the next rule. The result of the application defines only the direction of the movement. Therefore, these rules are only applied a bounded number of times.

However, this is not the case when the application mode is set to "IsExhaustive". In this case the transformation attempts to apply the LHS of the rule as long as there is a corresponding pattern in the host graph. In this manner, it has to be checked whether the rules applied in this way terminates. These rules are the following:

- RW_MATLAB_PROCESSFIRSTBLOCKS, where the rule attempts to match unprocessed simple elements. Since it marks the elements as "Processed" after each application, the rule is applied at most *n* times, where *n* means the number of simple elements in the Simulink model. The Simulink models contain only a bounded number of blocks, therefore the number of application of the rule is always bounded.
- The RW_MATLAB_PROCESSEDGES rule attempts to match processed simple elements with at least one outgoing edge, and deletes the matched edge. A block must have a bounded number of edges, therefore the rule cannot be applied indefinitely.
- The RW_MATLAB_DELETEPROCESSED rule is applied to process simple elements. Since the rule deletes the matched block, and a Simulink model has a bounded number of blocks, the rule is applied only bounded number of times.
- The RW_MATLAB_PROCESSINCOMINGEDGES rule attempts to match processed elements with at least one incoming edge. Since the rule is the same as the RW_MATLAB_PROCESS-EDGES, but with incoming edges, the reasoning is analogous.
- The RW_MATLAB_TAGTHEINBLOCKS, where the rule attempts to match Inport blocks and deletes their outgoing edges. A Subsystem must have a bounded number of Inport blocks and an Inport block must have a bounded number of outgoing edges, the rule is applied a bounded number of times.
- The RW_MATLAB_TAGTHEOUTBLOCKS is similar to the RW_MATLAB_TAGTHEIN-BLOCKS, but it attempts to match Outport blocks with incoming edges. The same reasoning can be applied here as well, that is, the Subsystem must have a bounded number of Outport blocks and an Outport block must have a bounded number of incoming edges. Therefore, the rule is applied a bounded number of times.
- The RW_MATLAB_PROCESSFIRSTSIN-SUBSYSTEM rule is the pair of the RW_MATLAB_PROCESSFIRSTBLOCKS rule at a deeper level. Since it looks for the same pattern with the extension of a parent element, the same reasoning can be applied here as well.



(a) The root level of the model (b) The Atomic Subsystem

Figure 10: The example Simulink model

Based on the previous, none of the rules in the TRANS_SL can be applied an indefinite number of times.

Now, it has to be checked, that the transformation does not contain infinite loops. It can be stated, that none of the transformation rules create any new element, that is no new edges or blocks are created throughout the process. However, every processed element (and its edges) is deleted by the appropriate rule. Since the Simulink model contains a bounded number of elements, the processing rules, and the related rules deleting elements, can be applied only a bounded number of types.

Furthermore, in Simulink models the number of Subsystems that can be used is limited and hierarchies of Subsystems cannot be created in a recursive manner. Moreover, it is not possible to create a hierarchy, where Subsystem S_A contains Subsystem S_B and S_B also contains S_A. With these restrictions it is ensured that the application of the RW_MATLAB_CHECKFOR-SUBSYSTEM transformation rule cannot lead to an infinite loop because each found Subsystem will be processed and deleted.

In this manner, since none of the transformation rules can be applied indefinitely and the transformation does not contain an infinite loop, it is proven that the transformation always terminates. ■

5. EXPERIMENTAL RESULTS

After introducing and analyzing the transformation, this section presents a simple example to demonstrate its functionality.

Figure 10 shows an example Simulink model. The top level of the model is depicted in Figure 10a, and the elements contained by the nonvirtual Subsystem are shown in Figure 10b. It is a simple example, which contains only one nonvirtual Subsystem, and a couple of simple elements. The transformation was examined on more complex models as well, and produced the expected results.

After the transformation for Figure 10b finished its execution, it resulted in the Sorted List depicted in Figure 11.

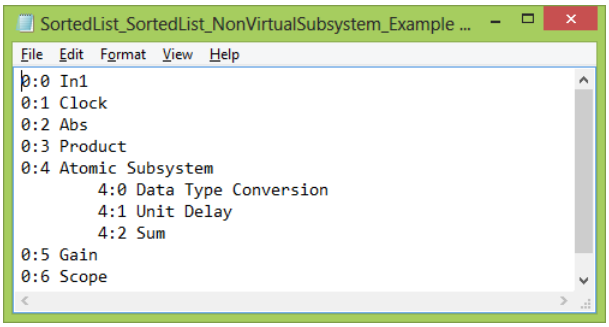


Figure 11: The resulting Sorted List

6. CONCLUSIONS

Nowadays Simulink is a popular tool for modeling embedded system. Simulink, in order to precisely model the functionality of the modeled system, can automatically elaborate a source model. Such elaboration is a form of model transformation process that is currently implemented in software as part of the Simulink code base.

Part of the elaboration is creating a Sorted List, which represents the dependency between the elements in the source model. In this paper, an algorithm is presented in detail, which is suitable for creating such a list. This algorithm is examined in terms of complexity.

Moreover, a detailed model transformation-based solution is also presented for creating Sorted Lists. This approach enables taking advantage of the benefits of 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, the analysis of the transformation is also discussed and a simple example is given to presents its applicability.

Future work intends to study whether with the help of this transformation, 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.

ACKNOWLEDGEMENTS

This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013) organized by VIKING Zrt. Balatonfüred. This work was partially supported by the Hungarian Government, managed by the National Development Agency, and financed by the Research and Technology Innovation Fund (grant no.: KMR_12-1-2012-0441).

REFERENCES

Angyal, L., Asztalos, M., Lengyel, L., Levendovszky, T., Madari, I., Mezei, G., Mészáros, T., Siroki, L., and Vajk, T., 2009, Towards a fast, efficient and customizable domain-specific modeling framework, *Software Engineering*.

Denckla, B. and Mosterman, P. J., 2004, An intermediate representation and its application to the analysis of block diagram execution, *Proceedings of the 2004 Summer Computer Simulation Conference (SCSC'04)*, pp. 167–172, July 2004.

Fehér, P., Mosterman, P. J., Mészáros, T., and Lengyel, L., 2012, Processing Simulink models with graph rewriting-based model transformation, *Model Driven Engineering Languages and Systems (MODELS '12) – Tutorials*.

Fowler, M., 2010, *Domain Specific Languages*, Addison-Wesley.

Han, Z. and Mosterman, P. J., 2010, Detecting data store access conflict in Simulink by solving boolean satisfiability problems, *Proceedings of the 2010 American Control Conference (ACC'10)*, pp. 5702–5707, June 2010.

Kelly, S. and Tolvanen, J.-P., 2008, *Domain-Specific Modeling: Enabling Full Code Generation*, Wiley.

Matlab® 2012b, http://www.mathworks.com/, 2012.

Mosterman, P. J. and Ciolfi, J. E., 2004, Using interleaved execution to resolve cyclic dependencies in time-based block diagrams, *Proceedings of 43rd IEEE Conference on Decision and Control (CDC'04)*, pp. 4057–4062, December 2004.

Mosterman, P. J., Prabhu, S., and Erkkinen, T., 2004, An industrial embedded control system design process, *Proceedings of The Inaugural CDEN Design Conference (CDEN'04)*, pp. 02B6–1–02B6–11, 2004.

Mosterman, P. J. and Vangheluwe, H., 2002, Computer automated multi-paradigm modeling, *ACM Transactions on Modeling and Computer Simulation*, vol. 12, no. 4, pp. 249–255.

Mosterman, P. J. and Vangheluwe, 2004, H., Computer automated multi-paradigm modeling: An introduction, *SIMULATION: Transactions of The Society for Modeling and Simulation International*, vol. 80, no. 9, pp. 433–450.

Mosterman, P. J., Sztipanovits, J., and Engell, S., 2004, Computer automated multi-paradigm modeling in control systems technology, *IEEE Transactions on Control Systems Technology*, vol. 12, no. 2, pp. 223–234.

Nicolescu, G. and Mosterman, P. J., 2010, Model-Based Design for Embedded Systems, *Computational Analysis, Synthesis, and Design of Dynamic Models Series*. CRC Press.

Simulink® 2012b, http://www.mathworks.com/simulink/, 2012.

Simulink® 2012b user's manual, http://www.mathworks.com/help/simulink/index.html, 2012.

Simulink® 2012b - direct feedthrough, http://www.mathworks.com/help/simulink/sfg/s-functionconcepts.html, 2012.

VMTS website, http://vmts.aut.bme.hu/, 2012.