A Novel Algorithm for Flattening Virtual Subsystems in Simulink Models

Péter Fehér, Tamás Mészáros and László Lengyel
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Budapest, Hungary
{feher.peter, mesztam, lengyel}@aut.bme.hu

Pieter J. Mosterman
Research and Development
MathWorks
Natick, MA, USA
pieter.mosterman@mathworks.com

***Abstract*—Recently embedded systems are often modeled using Simulink® to simulate their behavior. In order to perform the simulation, the modeling tool has to process the model. An important processing step is to determine the execution order of the elements in a model. This execution order is based on a sorted list of all semantically relevant model elements. Therefore, before simulation, Simulink® removes all model elements that only have a syntactic implication. In Simulink, the virtual subsystems are composite elements with no semantic bearing. Thus, Simulink performs a flattening model transformation that eliminates virtual subsystems. The work presented in this paper provides a novel algorithm for flattening composite elements in hierarchical models. Moreover, an optimized algorithm is also presented for Simulink models. With the implementation of these algorithms the level of abstraction of the model transformation can be raised. In this manner, a reusable, platform independent solution can be achieved for flattening Simulink subsystems.**

***Keywords*-Algorithms, Cyber-Physical Systems, Embedded Systems, Hierarchical Models, Model-Based Design, Simulink, Subsystems**

I. Introduction

Nowadays modeling is a popular approach during the development of embedded systems. By modeling them, the different systems can be examined on a higher abstraction level. In this manner, they can be analyzed in terms of, for example, performance, robustness or correctness validation.

Domain-specific modeling is a poplar approach to describe complex systems. It is a powerful, but still understandable technique. Its main strength lies in the application of domain-specific languages. A domain-specific language is a language specialized for a certain application domain; therefore, it is more efficient than the general purpose languages [1] [2].

The systems are often modeled by a modeling tool. These tools are frequently used to simulate different behaviors. However, in order to simulate these behaviors, usually the system has to be processed in some way.

While operating at a given level of abstraction, two mechanisms are often employed to scale system complexity: partitioning and hierarchy [3] [4]. In modeling tools hierarchy can be supported by two different constructs: a purely syntactic construct, and a construct with semantic implications. The composite elements that only serve graphical purposes can be effectively flattened during the processing phase. This means, that the elements inside this constructs must be moved out to the next hierarchical level.

Today, Simulink® [5] is often used for modeling embedded system. Its modeling environment offers different composite elements, which are called Subsystems. On the one hand, there are Subsystems with purely syntactic purposes. These are called virtual Subsystems. On the other hand, there are nonvirtual Subsystems such as Atomic Subsystems, which have a semantic bearing as well.

The first step of processing Simulink models is to create the Sorted List. The Sorted List constitutes a dependency list [6] [7]. In order to create this list, it is effective to flatten virtual Subsystems. Currently, the flattening task is implemented as a procedure in the Simulink code base. Though efficient, this makes it difficult or even prevents unlocking value for which a higher level of abstraction is more appropriate (e.g., reasoning about the implementation, modularization of operations, and property proving). As such, by capturing the flattening procedure at a higher level of abstraction, the advantageous properties of domain-specific modeling can be utilized. This is the fundamental premise of Computer Automated Multi-paradigm Modeling; to use the most appropriate formalism for representing a problem at the most appropriate level of abstraction [8] [9].

To be able to flatten composite elements, such as the Subsystem elements in Simulink, there is a need for well-formed, effective algorithms. Eventually, the implementation of these algorithms —in the appropriate environment— makes the flattening process possible on a higher abstraction level. In this manner, a more optimized solution can be found for the given problem.

This paper focuses on a novel solution to flatten composite elements. Two algorithms are introduced. The first is a basic solution that can be used on hierarchical elements found in Simulink that have no restriction on their ports and blocks. However, the second, optimized algorithm utilizes the attributes of the virtual subsystems; in this manner better performance can be achieved.

The rest of the paper is organized as follows. Section II presents the algorithms to flatten composite elements. In Section III the complexity of the algorithms are examined. Next, in Section IV, the presented approaches are briefly

summarized. Finally, concluding remarks are elaborated.

II. The Algorithms

As it was previously described, Simulink flattens virtual Subsystems in order to create the Sorted List, which eventually leads to the Execution List. To support analysis, modularization, etc., it is crucial to specify at a sufficiently high level of abstraction. In this section an approach is proposed that can serve as the foundation of a solution based on model transformation.

First, a basic algorithm is presented. With the implementation of this algorithm arbitrary composite elements can be flattened, for example, the Subsystem block element of a Simulink model. During the presentation of this approach the main attributes of the Subsystem blocks are also introduced. However, this algorithm does not utilize all properties of the Subsystem blocks. Therefore, next, an optimized algorithm is presented in Section II-B.

A. The basic approach

Although the flattening approach can be applied to any composite block with the same structural elements, it is presented as processing virtual Subsystems. In this manner, a comparison with the optimized algorithm becomes easier.

The main part of the algorithm is shown in Algorithm 1. The TRANSFLATTENER algorithm contains a *while* block. The iteration condition of this block calls the ISTHEREVS function, whose return value determines whether a virtual Subsystem is present in the model. If it returns *true*, then the vs variable stores the virtual Subsystem to process and the algorithm steps into the iteration. Otherwise, since there is no virtual Subsystem to process, it terminates. Inside the iteration, the PROCESSVIRTUALSUBSYSTEM function is called, which is responsible for the processing.

Algorithm 1 The algorithm of the transformation TRANSFLATTENER

1: **procedure** TRANSFALTTENER(**Model** m)
2: **Subsystem** $vs \leftarrow$ *null*
3: **while** ISTHEREVS(m, vs) $=$ true **do**
4: PROCESSVIRTUALSUBSYSTEM(vs)
5: **return**

As mentioned, the ISTHEREVS function, depicted in Algorithm 2, is responsible for determining whether a model m contains a virtual Subsystem and it determines the particular Subsystem block to flatten. Since the other parts of TRANSFLATTENER process the given virtual Subsystem, the function also has to return the found subsystem in the parameter vs. Of course, this is just an implementation detail, the same can be achieved, for example, by a common storage or a global variable. Suppose that the model has a property named *Subsystems* that lists all contained Subsystems. In this case, the function simply has to check if there is any item in this list with the *IsVirtual* attribute set to *true*. This attribute is responsible for distinguishing between virtual and nonvirtual (e.g., Atomic) Subsystems. If the function finds such a Subsystem it stores it in the vs parameter and returns with *true*. Otherwise, the return value is *false*.

Algorithm 2 The algorithm of finding a virtual Subsystem in the model

1: **procedure** ISTHEREVS(**Model** m, **Subsystem** vs)
2: **if** $\exists s | s \in m.Subsystems, s.IsVirtual =$ true **then**
3: $vs \leftarrow s$
4: **return** true
5: **return** false

The PROCESSVIRTUALSUBSYSTEM function is called directly from the TRANSFLATTENER procedure and is presented in Algorithm 3. PROCESSVIRTUALSUBSYSTEM function is responsible for processing the given virtual Subsystem, which is its only parameter. The method processes the virtual subsystem in ten different steps. The first set of these steps concentrates on the edges between the contained blocks. The next step deals with the position of the blocks. Finally, the last set of steps deletes the superfluous elements and recreates the edges between the blocks. In the following, some of these functions will be presented.

Algorithm 3 The algorithm that flattens a virtual Subsystem

1: **procedure** PROCESSVIRTUALSUBSYSTEM(**Subsystem** vs)
2: GETOUTEDGEINFO(vs)
3: PROCESSOUTBLOCKS(vs)
4: PROCESSSIMPLEEDGES(vs)
5: PROCESSINBLOCKS(vs)
6: MOVETOPARENTLEVEL(vs, $prefix$)
7: MOVETOROOTLEVEL(vs, $prefix$)
8: DELETECONTAINMENTEDGES(vs)
9: DELETESUBSYSTEM(vs)
10: CONNECTBLOCKS()
11: **return**

Flattening a Subsystem means that the blocks within the Subsystem (except the *In* and *Out* elements) are moved to the next (upper) hierarchy level (either a Subsystem or the root model) and the flattened Subsystem is deleted. If a block is moved from a Subsystem to another hierarchy level, then the related edges are removed automatically by Simulink. It is because, in Simulink, edges that cross hierarchical levels are not allowed. This means that there is no edge to or from the moved block at its new location. However, there is no guarantee that there are no dangling edges left inside the Subsystem. Therefore, the edges of the block have to be deleted before the block is moved out of the Subsystem. Since the edges are lost at the moment when the block is moved, the algorithm must take care of the edges connected to this block as well.

When a Subsystem is flattened, its *In* and *Out* ports are deleted. Therefore, the algorithm must know which blocks were connected to these ports in order to maintain the original

functionality. After the transformation, the blocks that were contained by the subsystem must be reachable by the exact same way (except the aforementioned ports) as they were before. The first two methods deal with the first part of this problem. In Simulink, each *In* and *Out* **port** is represented with an *Inport* and *Outport* block inside the subsystem. The elements connected to these **blocks** inside the subsystem are effectively connected to the elements connected to the appropriate **port** outside. At first, the algorithm concentrates on the *Out* ports of the subsystem. Since there are elements connected to the **ports** and elements connected to the **blocks**, it is convenient to handle the problem in two separate methods. Previous work [10] discusses how to deal with ports in composite elements in more detail.

The GETOUTEDGEINFO function, shown in Algorithm 4, stores for each *Outport* block all elements that are connected to the corresponding *Out* port. To achieve this, the algorithm iterates through the outgoing edges of the given virtual Subsystem. The *GetOutEdges* method of the Subsystem returns these edges. To be able to recreate the connection between the blocks, the port numbers also have to be stored for each edge. Moreover, the port number also identifies the corresponding *Outport* block. In this manner, the algorithm obtains the actual *Outport* block by calling the *GetOutBlock* method of the Subsystem with the appropriate port number. After the *Outport* block is found, the algorithm identifies the block at the other end of the actual edge. This block will be connected to the blocks linked to the actual *Outport* block. To this end, the algorithm stores this relevant information in the *Tag* attribute of the *Outport* block temporarily. At the end of the iteration the algorithm deletes the processed edge in order to avoid the appearance of dangling edges after the Subsystem is deleted. After the method iterates through all outgoing edges, all *Outport* blocks have the information about the edges going out from their corresponding *Out* ports.

Algorithm 4 The algorithm that obtains the information about the outgoing edges of the Subsystem

1: **procedure** GETOUTEDGEINFO(**Subsystem** vs)
2: **for all** e **in** $vs.GetOutEdges()$ **do**
3: $\quad outBlock \leftarrow vs.GetOutBlock(e.PortFrom)$
4: $\quad toBlock \leftarrow e.BlockTo$
5: $\quad info \leftarrow$ **new** EdgeInfo$(null, toBlock, null, e.PortTo)$
6: $\quad outBlock.Tag.Add(info)$
7: $\quad e.Delete()$
8: **return**

Since the *Outport* blocks now have the information about the outgoing edges, the PROCESSOUTBLOCKS (Algorithm 5) is only responsible for passing this information to the elements connected to the *Outport* blocks. To this end, the algorithm iterates through the edges connected to the *Outport* blocks. At each iteration the method determines the appropriate *Outport* block and the source of the edge. To maintain the functionality of the Simulink model, the blocks connected to the *Outport* block must be connected to all elements linked to the appropriate *Out* port. The information about these connections is stored in the *Tag* property of the *Outport* block. Since there can be multiple outgoing edges from each *Out* port, which means multiple item in the *Tag* property, the method has to iterate through this property. At each step of this iteration the algorithm stores the information in the *Tag* property of the source block. When this embedded iteration terminates, the algorithm also deletes the edge between the source block and the *Outport* block as the final step of the main iteration to avoid dangling edges. After these two methods have completed their execution all blocks connected to the *Outport* blocks contain all the necessary information to create edges connecting to all blocks linked to the corresponding *Out* ports.

Algorithm 5 The algorithm that constructs the information about the outgoing edges in the nodes connected to the *Outport* blocks of the Subsystem

1: **procedure** PROCESSOUTBLOCKS(**Subsystem** vs)
2: **for all** l **in** $vs.GetLastEdges()$ **do**
3: $\quad outPortB \leftarrow l.BlockTo$
4: $\quad fromBlock \leftarrow l.BlockFrom$
5: $\quad$ **for all** $info$ **in** $outPortB.Tag$ **do**
6: $\quad\quad info \leftarrow$ **new** EdgeInfo$(fromBlock, info.ToBlock, l.PortFrom, info.PortTo)$
7: $\quad\quad fromBlock.Tag.Add(info)$
8: $\quad l.Delete()$
9: **return**

At this point, the algorithm must store the information about the remaining edges in the Subsystem. The PROCESSSIMPLEEDGES function, shown in Algorithm 6, is responsible for this behavior. Note, that the edges connected to the *Outport* blocks are already deleted. PROCESSSIMPLEEDGES simply iterates through every outgoing edge of each block inside the Subsystem. The *Blocks* property of the Subsystem returns the list of the blocks contained by the Subsystem, while the *GetOutEdges* property returns the outgoing edges from the given block. Inside the embedded iteration the function simply identifies the block the edge is connected to and then stores the necessary information in the *Tag* property of that block. Finally, the processed edge is deleted. After the algorithm terminates, each block within the Subsystem stores all necessary information about the edges going out from them. In this manner, it is possible to correctly recreate these edges after the blocks are moved out of the subsystem.

Finally, the algorithm must ensure that the functionality does not change after the *In* ports and *Inport* blocks are deleted. This is achieved by the PROCESSINBLOCKS method (Algorithm 7). Since after the PROCESSSIMPLEEDGES the *Inport* blocks know about their outgoing edges, this method only has to copy and extend these items to the *Tag* property of the appropriate blocks. The main principle of this algorithm is the same as the aforementioned GETOUTEDGEINFO method. The algorithm iterates over the incoming edges of the Subsystem. The *GetInEdges* method of the Subsystem returns these items. At each iteration the algorithm identifies the source

Algorithm 6 The algorithm that constructs the information about the outgoing edges in all nodes

```
1: procedure ProcessSimpleEdges(Subsystem vs)
2: for all actualBlock in vs.Blocks do
3:     for all e in actualBlock.GetOutEdges do
4:         toBlock ← e.BlockTo
5:         info ← new EdgeInfo(actualBlock, toBlock,
           e.PortFrom, e.PortTo)
6:         actualBlock.Tag.Add(info)
7:         e.Delete()
8: return
```

block of the edge, this block must store the edge information. Moreover, the method also determines the actual port the edge is connected to. In this manner the corresponding *Inport* block can be determined. At this point the algorithm just iterates through the *Tag* property of this *Inport* block and stores the edge information in the source block based on the information found. Before the main iteration terminates, the method also deletes the current edge.

Algorithm 7 The algorithm that gets the information about the outgoing edges of the Subsystem

```
1: procedure ProcessInBlocks(Subsystem vs)
2: for all e in vs.GetInEdges() do
3:     fromBlock ← e.BlockFrom
4:     inBlock ← vs.GetInBlock(e.PortTo)
5:     for all info in inBlock.Tag do
6:         info ← new EdgeInfo(fromBlock, info.ToBlock,
           e.PortFrom, info.PortTo)
7:         fromBlock.Tag.Add(info)
8:     e.Delete()
9: return
```

The algorithms described in this section so far are responsible for saving the necessary information about the edges related to the Subsystem. In this manner the connection between its elements can be reestablished after the blocks are moved to a higher level.

The next set of algorithms deals with the actual flattening. These algorithms are responsible for replacing the elements of a Subsystem, deleting its edges and finally the Subsystem as well. There are two separate methods responsible for moving the elements inside the Subsystem to a higher hierarchy level. The first one is the MoveToParentLevel method, which is shown in Algorithm 8. This method replaces the elements only if the subsystem is a child element, that is, the Subsystem is an element of another Subsystem. The parent element is returned by the *Parent* property of the elements. If this property of a Subsystem is *null*, then it means the subsystem is not contained by anything, it is at the root level, thus the method terminates. Otherwise, it iterates through all blocks and resets their *Parent* and *Name* properties. The *Inport* and *Outport* blocks are excluded from the iteration since they are related to the Subsystem element. With the reset of the *Parent* property the block is moved out of the Subsystem.

Simulink does not allow two different blocks at the same hierarchy level to have the same name, therefore, all blocks are prefixed with a Subsystem-specific name.

Algorithm 8 The algorithm that moves the inner blocks of the Subsystem to the parent level

```
1: procedure MoveToParentLevel(Subsystem vs
   String prefix)
2: parent ← vs.Parent
3: if parent = null then
4:     return
5: for all actualBlock in vs.Blocks do
6:     if actualBlock.GetType() ∉ {InBlock, OutBlock}
       then
7:         actualBlock.Parent ← parent
8:         actualBlock.Name ← prefix +
           actualBlock.Name
9: return
```

After the MoveToParentLevel method the algorithm calls the MoveToRootLevel. This method moves the contained elements to the root level, that is, set their *Parent* property to *null*. Other than setting the *Parent* attribute to *null*, the iteration part is the same as it was presented above. Note, that the *Blocks* property of the Subsystem returns an empty list if the elements were already replaced by the MoveToParentLevel method.

The DeleteContainmentEdges method sets the *Parent* property of the Subsystem to *null*. In case the connection between the Subsystem and its parent is represented by a containment edge, it is important to delete this edge here.

The last method of this set is the DeleteSubsystem, which simply deletes the empty Subsystem.

Finally, the algorithm must reestablish the connection between the blocks. The ConnectBlocks method, depicted in Algorithm 9, is responsible for this. It iterates through the blocks that have edge information stored in their *Tag* property. Inside this iteration the method simply creates a new iteration over these items. In this embedded iteration the algorithm creates a new edge based on the actual edge information. After the edge is created, the edge information is removed from the *Tag* property. In this manner every edge will be recreated that has information stored in the blocks.

Algorithm 9 The algorithm that moves the inner blocks of the Subsystem to the parent level

```
1: procedure ConnectBlocks()
2: taggedNodes ← ∀n|n.Tag.Count > 0
3: for all actualBlock in taggedNodes do
4:     for all info in actualBlock.Tag do
5:         actualBlock.CreateEdge(info.ToBlock,
           info.PortFrom, info.PortTo)
6:         actualBlock.Tag.Remove(info)
7: return
```

Once there are no further edges to create, the algorithm has completed the flattening of the Subsystem. Next, the algorithm steps back to the main loop condition to determine whether there are any further Subsystems to process. If there are, the algorithm steps back into the iteration. Otherwise, the algorithm terminates.

B. *Algorithms optimized for Simulink®*

The basic algorithm can be applied to arbitrary composite elements with the same structure as a virtual Subsystem (i.e., it has *In* and *Out* ports and corresponding *Inport* and *Outport* blocks). However, the virtual Subsystems in Simulink have some additional properties this algorithm can take advantage of.

The first is the fact that in Simulink each *Outport* block has exactly one block that is connected to it. The basic algorithm is prepared to deal with multiple blocks connected to the *Outport* block and this is the reason why the *Out* port and *Outport* block cannot be processed in a single step. In case the *Out* port has three outgoing edges and the corresponding *Outport* block has four incoming edges, the algorithm creates twelve edges to ensure that the functionality is maintained after the flattening. Contrary to this, the optimized algorithm knows that exactly one block is connected to the *Outport* block, thus the processing of the *Out* ports and *Outport* blocks can be solved in one step. The algorithm operates in the same manner as GETOUTEDGEINFO (i.e., it iterates through the edges going out from the *Out* port) but instead of storing the information in the *Tag* property of the *Outport* block, it simply saves it into the block connected to the *Outport* block.

The other property to take advantage of is similar. The *In* port of the virtual subsystem also has this characteristic, that is, it has exactly one incoming edge. The basic algorithm handles the *In* port the same way as the *Out* port. In this manner the algorithm is designed to be able to flatten elements that have more blocks connected to the *In* port. However, in Simulink this is superfluous as the processing of the *In* ports and *Inport* block can be achieved in a single step. Note that this affects the PROCESSSIMPLEEDGES method as well, since this processes the outgoing edges from the *Outport* blocks too. In this manner, in the optimized algorithm the PROCESSSIMPLEEDGES does not process the edges connected to the *Outport* block. These edges are processed only once, when the *Out* ports are handled.

The main algorithm of the optimized approach is depicted in Algorithm 10.

The next section analyses these optimized algorithms to flatten composite elements.

Algorithm 10 The algorithm that flattens a virtual Subsystem

```
1:  procedure PROCESSVIRTUALSUBSYSTEM(
        Subsystem vs)
2:  PROCESSOUTPORTSANDBLOCKS(vs)
3:  PROCESSINNEREDGES(vs)
4:  PROCESSINPORTSANDBLOCKS(vs)
5:  MOVETOPARENTLEVEL(vs, prefix)
6:  MOVETOROOTLEVEL(vs, prefix)
7:  DELETECONTAINMENTEDGES(vs)
8:  DELETESUBSYSTEM(vs)
9:  CONNECTBLOCKS()
10: return
```

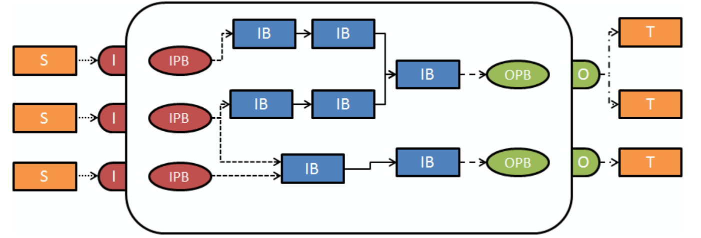

Fig. 1. The structure of a Subsystem

To distinguish the different types of elements contained by or connected to a Subsystem, new sets are introduced. These sets are depicted in Fig. 1. The different edge and block types are the following:

- i = {The edges connected to the *In* ports (the round dotted edges in Fig. 1).}
- f = {The edges going from the *Inport* blocks out (the square dotted edges).}
- l = {The edges going into the *Outport* blocks (the dashed edges).}
- e = {The edges contained by the Subsystem that are not part of the f or l sets (the solid edges).}
- o = {The edges connected to the *Out* ports (the dash dotted edges).}
- S = {The blocks connected to the *In* ports of the Subsystem.}
- I = {The *In* ports of the Subsystem.}
- IPB = {The *Inport* blocks of the Subsystem.}
- OPB = {The *Outport* blocks of the Subsystem.}
- O = {The *Out* ports of the Subsystem.}
- T = {The blocks connected to the *Out* ports of the Subsystem.}
- IB = {The other blocks contained by the Subsystem.}

The complexity of the basic algorithm is examined first, and then in Section III-B the optimized algorithm is investigated.

III. COMPLEXITY ANALYSIS

In order to use an algorithm in production applications its complexity has to be determined. In this section the algorithms are examined based on how many times their main parts are called, that is, which attributes determine their computational complexity, and, thereby, their execution time.

A. *The Basic Algorithm*

The complexity of the TRANSFLATTENER algorithm depends on the complexity of the ISTHEREVS and PROCESSVIRTUALSUBSYSTEM algorithms. The ISTHEREVS algorithm returns *true* as long as there is a virtual Subsystem in

the model. However, the complexity of the PROCESSVIRTUALSUBSYSTEM is more complicated. It contains nine different algorithms, so these algorithms have to be examined as well.

The first of these algorithms is GETOUTEDGEINFO. The body of this algorithm is a *for* loop. The algorithm terminates as soon as there is no unprocessed edge from any *Out* port related to the given Subsystem. Since the edges connected to the *Out* ports are depicted with o, the complexity of the GETOUTEDGEINFO algorithm is $\mathcal{O}(o)$.

When GETOUTEDGEINFO terminates, the algorithm calls the PROCESSOUTBLOCKS algorithm. This latter algorithm has embedded *for* blocks. The outer one iterates through the edges connected to the *Outport* blocks, while the inner one creates a new object for each object stored in the given *Outport* block. These objects are created by the GETOUTEDGEINFO. In this manner the complexity of the algorithm in a worst case scenario is $\mathcal{O}(l * o)$.

The next algorithm is PROCESSSIMPLEEDGES. This algorithm processes all edges in the subsystem. Since GETOUTEDGEINFO deleted the edges connected to the *Outport* blocks (l), the body of the method is called $f+e$ times. In this manner the complexity is $\mathcal{O}(f + e)$.

The PROCESSINBLOCKS algorithm contains the *for* loops, similarly to the PROCESSOUTBLOCKS algorithm. The method iterates through the edges connected to the *In* ports. At each iteration PROCESSINBLOCKS creates a new object for every edge stored in the related *Outport* blocks. In this manner the complexity of this algorithm is $\mathcal{O}(i * f)$.

After the edges are processed, the PROCESSVIRTUALSUBSYSTEM algorithm moves to the MOVETOPARENTLEVEL algorithm. In case the given Subsystem is a child element of another Subsystem then it moves each block (except the *Inport* and *Outport* ones) to a higher level. This means that every block in the IB set is moved out, which results in a complexity of $\mathcal{O}(IB)$. In case the Subsystem is at the root level then the replacing is done by the MOVETOROOTLEVEL algorithm. Since MOVETOROOTLEVEL moves the same blocks its complexity is the same as MOVETOPARENTLEVEL. Note, that only one of the algorithms moves the blocks, so the complexity of this part is $\mathcal{O}(IB)$ in both cases.

The next problem is the deletion of the Subsystem. This is done by two algorithms, but both the DELETECONTAINMENTEDGES and DELETESUBSYSTEM algorithm delete at most one element, so their complexity can be approximated with $\mathcal{O}(1)$.

Finally, the edges have to be recreated. This is the responsibility of the CONNECTBLOCKS algorithm. Since the functionality of the model has to be maintained, the blocks connected to the *In* ports have to have an edge to all blocks that were connected to the appropriate *Inport* blocks. This reasoning is true in case of the blocks connected to the *Out* ports. In this manner, the number of edges to be created is $i * f + e + l * o$. The method deals with only these edges, so the complexity of it is exactly $\mathcal{O}(i * f + e + l * o)$.

This means that the complexity of processing a virtual subsystem in case of the basic algorithm is $\mathcal{O}(o + (l * o) + (f + e) + (i * f) + IB + 2 + (i * f + e + l * o))$, which is equal to $\mathcal{O}(o + 2e + f + 2(i * f + l * o) + IB)$. Since the complexity of the ISTHEREVS algorithm is $\mathcal{O}(1)$ the complexity of the basic algorithm is $\mathcal{O}(n * (o + 2e + f + 2(i * f + l * o) + IB))$, where n represents the number of virtual subsystem contained by the model.

B. The Optimized Algorithm

The complexity derived in Section III-A is reduced in case of the optimized algorithm. As it is discussed in Section II-B, the optimized algorithm differs from the basic approach in three methods.

First of all, the GETOUTEDGEINFO and PROCESSOUTBLOCKS methods are merged into the PROCESSOUTPORTSANDBLOCKS method. Taking advantage of the fact that the *Outport* blocks have exactly one incoming edge, the algorithm only iterates through the edges connected to the *Out* ports. In this manner, the complexity of this algorithm is $\mathcal{O}(o)$.

In the optimized algorithm the PROCESSSIMPLEEDGES method is changed to the PROCESSINNEREDGES method. This method only processes the edges that are connected to inner elements. This means that the edges connected to the *Inport* blocks are not processed in this step. In this manner, the complexity of the method is $\mathcal{O}(e)$.

Finally, the PROCESSINBLOCKS method is changed to the PROCESSINPORTSANDBLOCKS. Similarly to the PROCESSOUTPORTSANDBLOCKS method, this function simplifies the processing of the *In* ports. Since it only has to iterate through the edges connected to the *Inport* blocks, the complexity of it is $\mathcal{O}(f)$.

Moreover, these restrictions (i.e., there is exactly one edge connection to each *In* ports and *Outport* block) have an effect on the CONNECTBLOCKS method as well. In case of a Simulink model, there are only $f + e + o$ edges to create after the blocks are moved out, which means $\mathcal{O}(f + e + o)$ complexity.

To summarize, considering only Simulink models and exploiting their restrictions the complexity is $\mathcal{O}(o+e+f+IB+2+(f+e+o))$. This equals to $\mathcal{O}(2(f+e+o)+IB+2)$. In other words, it is equal to deleting and recreating the relevant edges and moving out the relevant blocks.

Since the complexity of the ISTHEREVS method is $\mathcal{O}(1)$ the complexity of the optimized algorithm in case of Simulink models can be reduced to $\mathcal{O}(n * (2(f + e + o) + IB))$, where n represents the number of virtual Subsystems contained by the model.

IV. COMPARISION OF THE ALGORITHMS

Both presented algorithms are capable of flattening composite elements. After the algorithms are executed, the contained elements are placed at the next hierarchy level while the composite elements are deleted from the model.

The basic algorithm can be applied in any modeling environment where the composite elements consist of *In* and *Out* ports, *Inport* and *Outport* blocks, and inner elements. If these structural restrictions are met, the algorithm can be used with

the presented complexity. Moreover, the algorithm can be used as a base algorithm and tailored to more specific platforms.

The optimized algorithm is designed for Simulink models. It can be used for flattening virtual Subsystems. In this manner, it can be applied to raise the abstraction level of the preprocessing of Simulink models. Since the algorithm is optimized for Simulink, it takes advantage of the restrictions implemented in the environment. Based on the fact that in Simulink the *Outport* blocks have exactly one incoming edge, the algorithm does not have to store any information in the *Outport* blocks and then iterate through them. Instead the algorithm can directly store the information in the block connected to the *Outport* blocks. In this manner, the complexity can be reduced from $\mathcal{O}(o + l * o)$ to $\mathcal{O}(o)$. In Simulink, the *In* ports have exactly one incoming edge, so with similar reasoning, the algorithm can be transformed to process each block connected to the *Inport* blocks just once. This results in decreasing the complexity from $\mathcal{O}(f + e + i * f)$ to $\mathcal{O}(e + f)$. Finally, these optimizations can be applied during the recreation of the edges, which leads to $\mathcal{O}(f + e + o)$ instead of the $\mathcal{O}(i * f + e + l * o)$. In this manner, the optimized approach always performs better than the basic algorithm.

V. Conclusions

Nowadays Simulink is a popular tool for modeling and design of embedded control systems. The digital representation of models allows automatic generation of an implementation (such as software code) from a model. However, the design relies heavily on model elaboration to increasingly add detail to the design models.

Part of the implementation generation is removing hierarchical structure that has only a syntactic effect which includes flattening of syntactic hierarchical levels represented by composite elements. In this paper a basic algorithm is presented to process such composite elements. Moreover, an optimized algorithm is also introduced that takes advantage of the restrictions that can be found in a Simulink model with respect to virtual Subsystems. The computational complexity of these two algorithms is determined and compared. With the help of these algorithms the abstraction level of the processing can be increased to allow the benefits of abstraction such as more efficient reasoning, reuse, analysis, etc.

Future work intends to study the application of these algorithms, how they can be applied, for example in a graph transformation. Once virtual Subsystems have been removed, a Sorted List of semantic elements must be created from which an Execution List is derived. Obtaining these lists is part of future work with a focus on further increasing the abstraction level.

VI. Acknowledgments

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.

References

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

[2] M. Fowler, *Domain Specific Languages*, ser. The Addison-Wesley Signature Series. Addison-Wesley, 2010.

[3] K. C. J. Wijbrans, “Twente hierarchical embedded systems implementation by simulation: A structured method for controller realization,” Ph.D. dissertation, University Twente, Enschede, The Netherlands, 1993.

[4] P. Mosterman, J. Sztipanovits, and S. Engell, “Computer-automated multiparadigm modeling in control systems technology,” *Control Systems Technology, IEEE Transactions on*, vol. 12, no. 2, pp. 223–234, march 2004.

[5] “Simulink® 2012b,” http://www.mathworks.com/simulink/, 2012.

[6] “Simulink® 2012b user’s manual,” http://www.mathworks.com/help/simulink/index.html, 2012.

[7] 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.

[8] P. J. Mosterman and H. Vangheluwe, “Computer automated multiparadigm modeling in control system design,” *IEEE Transactions on Control System Technology*, vol. 12, pp. 65–70, 2000.

[9] P. J. Mosterman and H. Vangheluwe, “An introduction to computer automated multi-paradigm modeling,” 2004.

[10] M. Wimmer, W. Retschitzegger, G. Kappel, J. Schoenboeck, A. Kusel, and W. Schwinger, “Plug & play model transformations: a dsl for resolving structural metamodel heterogeneities,” in *Proceedings of the 10th Workshop on Domain-Specific Modeling*, ser. DSM ’10, 2010, pp. 7:1–7:6.