Data Type Propagation in Simulink Models with Graph Transformation

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*—Embedded systems are usually modeled to simulate their behavior. Nowadays, this modeling is often implemented in the Simulink® environment, which offers strong support for modeling complex systems. Moreover, via modeling, various analyses can be applied to the systems at design time. An important analysis is of the data types assigned to signal variables. Such analysis enables identification of potential problems during model compilation and so prevent runtime surprises. To assist the system designer, Simulink supports which includes a step for automatic data type inferencing (e.g., based on designed signal value ranges and on the connection structure of design elements). In contrast to the Simulink implementation of the inferencing in a code base, which favors efficiency, the work presented here raises the level of abstraction by explicitly modeling the inferencing logic. This unlocks benefits such as the ability to (i) reason about the logic, (ii) implement different logic by advanced users, and (iii) experiment with different ordering of other logic in the model compilation.**

***Keywords*-Algorithms; Data type propagation; Graph transformation; MATLAB; Simulink**

I. Introduction

Today, computation as a technology is finding its way into almost any engineered system around. To a large extent, the choice to utilize computation is motivated by the unparallelled flexibility of software as an implementation vehicle for complex functionality. In embedded systems, computation is embedded in a physical environment such that it processes physical measurements and reacts with actuation in the physical world. Such systems are designed in an elaborate process where a range of domain experts are responsible for an increasingly detailed model of the functionality. For example, at a high level, a control expert may design a continuous-time feedback control law, in a next stage, this control law is then transformed into a digital representation, next effects of the computing platform may be included such as the task scheduler and data types, and ultimately an implementation in software may be automatically generated (e.g., [1] [2]).

In such an overall process there is a critical need to support domain experts in their design of what will ultimately be a computational solution. Such support can be substantially provided by selecting the most appropriate modeling framework given the task at hand. Here, domain specific modeling languages that correspondingly enable domain specific modeling [3] [4] are a powerful approach to successfully design complex systems. Because domain specific modeling languages are specialized for a certain application domain, they often are more efficient than general purpose languages.

In addition to domain specific modeling languages, deep knowledge about the application domains involved in the various stages of the system design process enables automatic model elaboration. For example, knowledge of a computing platform that is targeted for the software design provides information as to the data types that are available. This knowledge then enables an automatic choice of data types in the digital control design stage so that the control design domain expert is not confronted with such implementation details while still being able to produce an executable model.

A popular tool for model based software development in the context of embedded systems design is Simulink® [5] not only because Simulink allows simulation of a design throughout the design stages but also because of extensive automatic code generation capabilities (both for a software as well as a hardware implementation). For both of these technologies (simulation and code generation) it is essential that a computational representation of the model is generated. An inherent part of generating a computational representation is determining the data types of the variables in the code. In the earlier design stages, these data types are often not specified at all (simply considered to be Real) or only partly specified explicitly at key design elements where the remaining data types are then automatically inferred.

In Simulink, the data types of a model are a inferred in a compilation stage, for example by using forward and backward data type propagation rules [6]. To explicitly indicate that a data type should be inferred, it can be set to 'Inherit.' In addition to propagation rules, further information is utilized in the data type determination. For example, a default data type such as 'double' may be set in case no other explicit resolution is inferred. As another example, given a number of available bits per word on the computing platform, the design range of values for a variable may determine a representation in fixed point. Moreover,

Simulink also checks the types of signals and input ports in order to ensure that they do not conflict.

The combination of such domain specific elaboration with domain specific languages combined then accelerates system development, increases the quality of software products, and reduces time to market.

Currently, data type propagation is implemented in the Simulink code base. Though efficient, this renders it difficult or even prevents unlocking value for which a higher level of abstraction is more appropriate. Specifically, a higher level of abstraction enables:

- Reasoning about the inference procedure, for example to compare to and contrast with other approaches such as Hindley-Milner [7] [8],
- Designing different rules by advanced users, which may support further domain configurability,
- Experimenting with different permutations in which a number of elaboration steps (e.g., flattening [9], task scheduling, etc.) are applied.

Similarly to how domain specific modeling provides value at the application domain level, it also provides value when the application domain consists of computer aided design systems with extensive tool technology. Based on the fundamental premise of Computer Automated Multiparadigm Modeling [10] [11], the data type inferencing should be modeled by the most appropriate formalism while choosing the most appropriate level of abstraction to solve the problem at hand.

This paper focuses on a novel solution to realizing data type propagation for Simulink models. This approach is based on a model transformation-based solution, which is realized in the Visual Modeling and Transformation System (VMTS) framework [12]. The VMTS has been prepared to be able to communicate with the Simulink environment; therefore the model transformations designed in VMTS can be applied to different Simulink models. Using model transformation to solve the data type propagation issue helps to raise the abstraction level from the API programming to the level of software modeling. The solution possesses all advantageous characteristic of the model transformation, for example, it is transparent, easy to define and understand, reusable, and platform independent.

The rest of the paper is organized as follows. Section II presents the algorithm used for propagating and verifying data types. Next, Section III introduces the implemented transformation. A simple example for the transformation is presented in Section IV. Finally, concluding remarks are elaborated.

II. Propagating Data Types

Whenever a simulation is started, the Simulink software performs a processing step called data type propagation. This step involves determining the types of signals whose type is not otherwise specified and checking the types of signals and input ports to ensure that they do not conflict. If type conflicts arise, an error dialog is displayed that specifies the signal and port whose data types conflict. The signal path that creates the type conflict is also highlighted [13].

Blocks can inherit data types from a variety of sources, including signals to which they are connected and particular block parameters. In case the data type of a block is not set to a built-in data type, or to an expression that evaluates it, then the type is calculated by inheritance. In this manner, the data type of the block is marked as "Inherit: $<rule>$", where the $rule$ determines the mode of the inheritance. It can be, for example, via back propagation, or same as the input, etc. The aim of a data type propagation algorithm is to determine the data types of these blocks and revise the types based on the signal on the input port of the blocks.

The algorithm presented in this section performs these modifications. As the Algorithm 1 shows, the TypeProp algorithm can be split into two parts. The first part sets the data types of the blocks, where the value of the data type is inherited. Next, the second part of it verifies the data types, and if it is necessary, it changes the type based on the value of the connected signals.

Algorithm 1 The algorithm of the transformation TypeProp

```
1: procedure TypeProp()
2: SetTypes()
3: VerifyTypes()
4: return
```

The SetTypes algorithm (shown in Algorithm 2) is responsible for setting the data types of the blocks that have this attribute set to "Inherited" in the model. This procedure requires four different steps. First, it is possible to set the data types of the Constant blocks Next, the base method of the SetTypes algorithm sets the data type attribute of the appropriate blocks based on the data type of the previous blocks. Since the data type propagation through hierarchical levels may affect the applicability of this rule, a *repeat-until* block is implemented in the algorithm, and the propagation through the hierarchical levels is realized in the *until* block. Moreover, since there is no guarantee that data types of the blocks without incoming edges are set properly, the algorithm must set the data types of these blocks as well.

Throughout the design of these algorithms, the existences of certain sets have been assumed. These are the following:

- The $InheritConstants$ set contains all Constant blocks, which have their data types calculated by inheritance.
- The $InheritSimpleBlocks$ set contains all non-composite elements (i.e., no Subsystems), which have their data types calculated by inheritance.
- The $ConcreteSimpleBlocks$ set consists of the non-composite elements, which have their data types set to

Algorithm 2 The algorithm of the transformation SETTYPES

```
1: procedure SETTYPES()
2: SETCONSTANTTYPE()
3: repeat
4:    SETNEXTINHERIT()
5: until not SETFIRSTINSUBSYSTEM() and not SETTYPENEXTTOSUBSYSTEM() and not SETPREVIOUSINHERIT()
6: return
```

a built-in type.

- The $InheritInportBlocks$ set contains all Inport blocks, which have their data types calculated by inheritance.
- The $InheritOutportBlocks$ set contains all Outport blocks, which have their data types calculated by inheritance.
- The $Subsystems$ set consists of all Subsystem elements in the model.
- The $SimpleBlocks$ set contains all elements in the model but the Subsystems.

Based on these sets, the SETCONSTANTTYPE algorithm (depicted in Algorithm 3) iterates through the $InheritConstants$ set and calls the SETTYPEBASEDONVALUE method on each item. This method sets the data type attribute to the appropriate value based on the value of the constant that is output by the Constant block.

Algorithm 3 The algorithm of the transformation SETCONSTANTTYPE

```
1: procedure SETCONSTANTTYPE()
2: for all c|c ∈ InheritConstants do
3:    SETTYPEBASEDONVALUE(c)
4: return
```

After all Constant blocks are typed into a specific built-in type, the SETTYPES algorithm starts setting the data type of the non-composite elements. In the body of the aforementioned *repeat-until* block there is only one algorithm, the SETNEXTINHERIT, which is shown in Algorithm 4. This algorithm is responsible for the data type propagation on a given hierarchical level. As shown, the algorithm iterates over all block pairs, where both blocks are non-composite elements and they are connected. On the one hand, the data type of the source block must be set to a built-in data type, therefore this block is an element of the $ConcreteSimpleBlocks$ set. On the other hand, the data type of the target block must be calculated by inheritance (otherwise there is no data type to set), so it is part of the $InheritSimpleBlocks$ set. At each iteration step, the algorithm sets the $OutDataTypeStr$ attribute of the target block to the value of the same attribute of the source block. In Simulink, the value of data type of the actual element is stored in this attribute.

Algorithm 4 The algorithm of the transformation SETNEXTINHERIT

```
1: procedure SETNEXTINHERIT()
2: for all a|a ∈ ConcreteSimpleBlocks, b|b ∈ InheritSimpleBlocks ∧ b ∈ a.Targets do
3:    b.OutDataTypeStr = a.OutDataTypeStr
4: return
```

In case these block pairs are all set, the possibility of data type propagation over hierarchical levels has to be checked. The SETFIRSTINSUBSYSTEM algorithm, which is depicted in Algorithm 5, is responsible for this behavior. The algorithm looks for block pairs, where the source block is part of the $ConcreteSimpleBlocks$ and the target is a Subsystem, and iterates through these pairs. In the next step, the algorithm checks whether the data type of the appropriate Inport block is calculated by inheritance. The appropriate Inport block is selected based on the Port number. The edges have the information about which port of the blocks they are connected to. This information is stored in the $PortFrom$ and $PortTo$ properties. The Port properties of the Inport and Outport blocks represent the port number of the Subsystem where they receive or send their signals. In this manner, the connection of the blocks can be examined. In case the data type of the appropriate Inport block is calculated via inheritance, the algorithm set it to the value of the data type of the source block.

Algorithm 5 The algorithm of the transformation SETFIRSTINSUBSYSTEM

```
1: procedure SETFIRSTINSUBSYSTEM()
2: Boolean processedAny ← false
3: for all e|e.Source ∈ ConcreteSimpleBlocks, s|s ∈ Subsystems ∧ s = e.Target do
4:    if ∃i|i ∈ s.InheritInportBlocks ∧ i.Port = e.PortTo then
5:       a ← e.Source
6:       i.OutDataTypeStr = a.OutDataTypeStr
7:       if not processedAny then
8:          processedAny ← true
9: return processedAny
```

The data type propagation must be implemented in the opposite direction as well. In this propagation, the data type of the Outport block of a Subsystem is set, and the data type of the appropriate block connected to this port of the Subsystem is calculated. The SETTYPENEXTTOSUBSYSTEM algorithm (shown in Algorithm 6) handles these situations, and sets the $OutDataTypeStr$ attribute to the value of Outport block.

Finally, as it was mentioned, there are cases, where the data type of a block cannot be calculated via propagation

Algorithm 6 The algorithm of the transformation SETTYPENEXTTOSUBSYSTEM

1: **procedure** SETTYPENEXTTOSUBSYSTEM()
2: **Boolean** $processedAny \leftarrow$ **false**
3: **for all** $e|e.Target \in InheritSimpleBlocks, s|s \in Subsystems \wedge s = e.Source$ **do**
4: **if** $\exists o|o \in s.ConcreteOutportBlocks \wedge o.Port = e.PortFrom$ **then**
5: $a \leftarrow e.Target$
6: $a.OutDataTypeStr = o.OutDataTypeStr$
7: **if not** $processedAny$ **then**
8: $processedAny \leftarrow$ **true**
9: **return** $processedAny$

in this forward direction. In these cases, the data type is set based on the value of the next block. The SETPREVIOUSINHERIT algorithm is responsible for this behavior. As it is shown in Algorithm 7, this algorithm differs from the SETNEXTINHERIT algorithm in only one aspect: in this case the source block is from the $InheritSimpleBlocks$ set. In this manner, the backpropagation can be accomplished as well.

Algorithm 7 The algorithm of the transformation SETPREVIOUSINHERIT

1: **procedure** SETPREVIOUSINHERIT()
2: **Boolean** $processedAny \leftarrow$ **false**
3: **if** $a|a \in ConcreteSimpleBlocks, b|b \in InheritSimpleBlocks \wedge b \in a.Sources$ **then**
4: $b.OutDataTypeStr = a.OutDataTypeStr$
5: **if not** $processedAny$ **then**
6: $processedAny \leftarrow$ **true**
7: **return** $processedAny$

In case any of the three algorithms in the *until* statement returns *true*, the SETTYPES algorithm applies the SETNEXTINHERIT again. Otherwise, the algorithm terminates, and the TYPEPROP algorithm moves on to the type verification part.

This verification part is contained by the VERIFYTYPES algorithm, shown in Algorithm 8. Its structure is very similar to the SETTYPES algorithm, that is, it uses a *repeat-until* block, in which it attempts to verify the data types of the connected block, and the *until* block contains the verification throughout the hierarchical levels.

The VERIFYSIMPLEBLOCK algorithm (shown in Algorithm 9) checks every edge that connects to non-composite elements. At each edge, the algorithm calls the COMPATIBLETYPES method with the source and target blocks as parameters. This method determines whether the data types of the blocks are compatible. In case they are, the algorithm does not modify them. Otherwise, the data type of the target block is set to the data type of the source block. The non-

Algorithm 8 The algorithm of the transformation VERIFYTYPES

1: **procedure** VERIFYTYPES()
2: **repeat**
3: VERIFYSIMPLEBLOCK()
4: **until not** VERIFYFIRSTINSUBSYSTEM() **and not** VERIFYNEXTTOSUBSYSTEM()
5: **return**

compatibility in this algorithm is interpreted to mean that there is arithmetic overflow. For example, if a block a has an incoming edge from a block, which has the *OutDataTypeStr* attribute set to *int32*, this a block cannot have its *OutDataTypeStr* set to, for example, *int16*. These conversion rules are implemented in the COMPATIBLETYPES method. If any of the rules is broken by the blocks passed as the parameters, the algorithm, as it was mentioned, modifies the data type of the target block to the data type of the source block. Since the algorithm should not overwrite the manually added data type conversions, the target block of the edge cannot be a Convert element.

Algorithm 9 The algorithm of the transformation VERIFYSIMPLEBLOCK

1: **procedure** VERIFYSIMPLEBLOCK()
2: **for all** $e|e.Source \in SimpleBlocks \wedge e.Target \in SimpleBlocks \wedge e.Target$ **is not** $Convert$ **do**
3: $a \leftarrow e.Source$
4: $b \leftarrow e.Target$
5: **if** $\neg$COMPATIBLETYPES(a, b) **then**
6: **print** $b.Name - a.OutDataTypeStr$
7: $b.OutDataTypeStr \leftarrow a.OutDataTypeStr$
8: **return**

As the data types are propagated across hierarchical levels, the verification performs verification across hierarchical levels as well. The VERIFYFIRSTINSUBSYSTEM algorithm, which is shown in Algorithm 10, is very similar to the already discussed SETFIRSTINSUBSYSTEM algorithm. However, instead of checking whether the data type of the appropriate Inport block is calculated or not, this time its compatibility is checked with respect to the block connecting to the Subsystem. In case there is a compatibility issue, the algorithm modifies the *OutDataTypeStr* attribute.

The last algorithm to be discussed is the VERIFYNEXTTOSUBSYSTEM, which is depicted in Algorithm 11. This algorithm is responsible for the verification in the other (i.e., bottom-up) hierarchical direction. This time the data type of the appropriate Outport block is examined. In case it is not compatible with the data type of the block b connecting to the Subsystem, then the *OutDataTypeStr* attribute of the b block is modified.

As can be seen, the verification part of the algorithm

Algorithm 10 The algorithm of the transformation VERIFYFIRSTINSUBSYSTEM

1: **procedure** VERIFYFIRSTINSUBSYSTEM()
2: **Boolean** $processedAny \leftarrow$ **false**
3: **for all** $e|e.Source \in SimpleBlocks, s|s \in Subsystems \wedge s = e.Target$ **do**
4: $a \leftarrow e.Source$
5: $b \leftarrow i|i \in s.InportBlocks \wedge i.Port = e.PortTo$
6: **if** $\neg CompatibleTypes(a, b)$ **then**
7: **print** $b.Name - a.OutDataTypeStr$
8: $b.OutDataTypeStr \leftarrow a.OutDataTypeStr$
9: **if not** $processedAny$ **then**
10: $processedAny \leftarrow$ **true**
11: **return** $processedAny$

Algorithm 11 The algorithm of the transformation VERIFYNEXTTOSUBSYSTEM

1: **procedure** VERIFYNEXTTOSUBSYSTEM()
2: **Boolean** $processedAny \leftarrow$ **false**
3: **for all** $e|e.Target \in SimpleBlocks \wedge e.Target$ is **not** $Convert, s|s \in Subsystems \wedge s = e.Source$ **do**
4: $a \leftarrow o|o \in s.OutportBlocks \wedge o.Port = e.PortForm$
5: $b \leftarrow e.Source$
6: **if** $\neg CompatibleTypes(a, b)$ **then**
7: **print** $b.Name - a.OutDataTypeStr$
8: $b.OutDataTypeStr \leftarrow a.OutDataTypeStr$
9: **if not** $processedAny$ **then**
10: $processedAny \leftarrow$ **true**
11: **return** $processedAny$

relies heavily on the capability of the COMPATIBLETYPES method.

In case either the VERIFYFIRSTINSUBSYSTEM or the VERIFYNEXTTOSUBSYSTEM algorithm returns *true*, the VERIFYTYPES algorithm must step into the *repeat* block again, since there may be new edges that meet the condition in the VERIFYSIMPLEBLOCK algorithm. Otherwise, the two algorithms return *false*, the VERIFYTYPES algorithm terminates, and therefore, the TYPEPROP algorithm terminates as well.

III. IMPLEMENTING THE ALGORITHM

The previous section presented the algorithm, which provides a solution for propagating and verifying data types in Simulink models. This section presents a novel approach to realizing this algorithm by implementing it via graph transformations.

In order to realize the transformation rules, a modeling and transformation framework is necessary. This framework is introduced in Section III-A. The transformation itself, that is, the controlling structure and the details of the transformation rules, is discussed in Section III-B.

A. The Modeling Environment

To create transformations of the Simulink model, the Visual Modeling and Transformation System (VMTS) [12] [14] 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 easier to comprehensively express. VMTS applies different colors to distinguish the LHS from the RHS in the presentation layer. Imperative constraints can also be applied.

In VMTS the 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.

B. The Transformation

It was mentioned, in VMTS, the order of the transformation rules is determined in a separate model, the control flow. The control flow of the TRANS_TYPEPROP transformation, which implements the propagation algorithm, is shown in Figure 1.

The first thing one may recognize is the number of directed cycles in the model. These directed cycles implement the *repeat-until* blocks of the algorithms.

Since the data types of the Constant blocks do not depend on any other blocks, the transformation starts with setting the *OutDataTypeStr* attribute of the Constant blocks. This is achieved in the RW_MATLAB_SETCONSTANT transformation rule. The rule attempts to match the Constant blocks that have their *OutDataTypeStr* attributes start out with being set

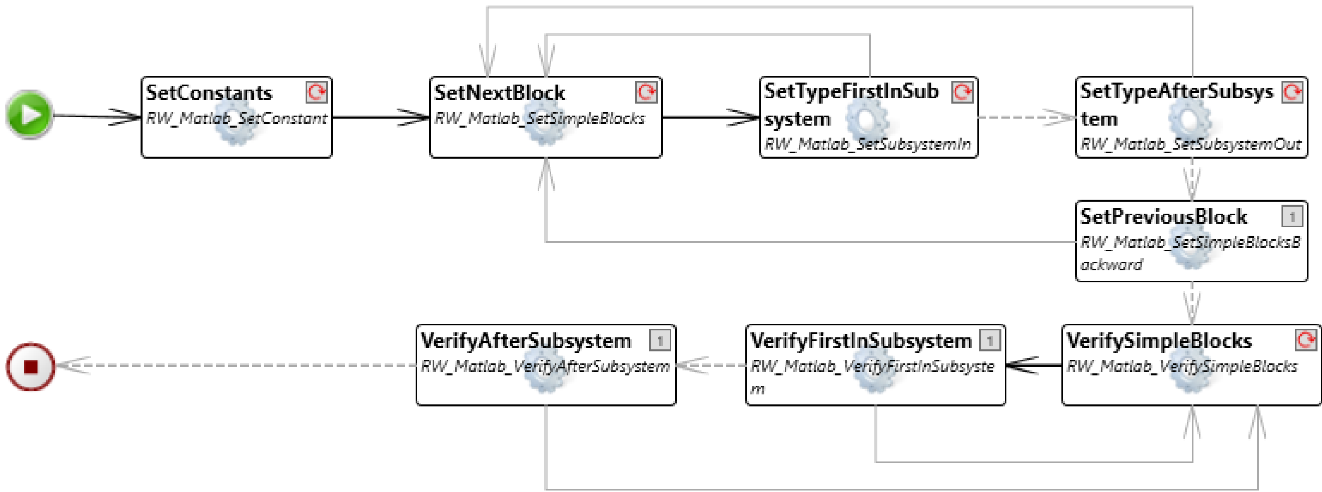


Figure 1. The control flow of the TRANS_TYPEPROP transformation

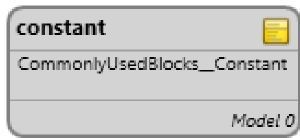


Figure 2. The transformation rule RW_MATLAB_SETCONSTANT

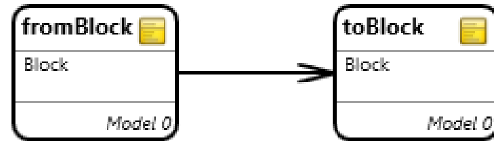


Figure 3. The transformation rule RW_MATLAB_SETSIMPLEBLOCKS

to "Inherit". The imperative code of the transformation finds the appropriate data type for the given value and assigns it to the attribute. The transformation rule is applied as long as there is a Constant block without explicitly set data type. The model of the transformation rule is shown in Figure 2.

Next, the transformation attempts to propagate the data types in case of blocks on the same hierarchical level. This is the responsibility of the RW_MATLAB_SETSIMPLEBLOCKS transformation rule, which is depicted in Figure 3. The matched blocks have the following constraints:

- The blocks cannot be composite elements.
- The data type of the source block, depicted as *fromBlock* in Figure 3, must be set explicitly.
- The *OutDataTypeStr* attribute of the target block, depicted as *toBlock* in Figure 3, must start out to be set to "Inherit". As was mentioned, this means that the data type of the block is calculated.

In case the rule finds a valid match, the *OutDataTypeStr* attribute of the target block is updated with the data type of the source block. The transformation applies this rule exhaustively, implementing the propagation as long as possible. As a result, this transformation rule realizes the SETNEXTINHERIT algorithm.

After there are no connected blocks where the data propagation can be applied, the transformation looks for data propagation through hierarchical levels. As such, it attempts to apply the RW_MATLAB_SETSUBSYSTEMIN transformation rule, which is shown in Figure 4. This rule attempts to find a match, where:

- A simple block is connected to a Subsystem element,
- The data type of the source block, depicted as *fromBlock*, is explicitly set,
- The Inport block (the *inBlock* in the Figure) of the Subsystem has its *OutDataTypeStr* attribute set to be based on a calculated value, that is, it starts off being set to "Inherited".

In case a match is found with this structure, the rule propagates the data type of the source block to the Inport block of the Subsystem, realizing the data type propagation across hierarchical levels.

This RW_MATLAB_SETSUBSYSTEMIN rule is the first rule in the transformation, where the applicability of the rule determines the next transformation rule. On the one hand, if the transformation rule was applied successfully (i.e., at least once), there might be connected simple blocks, where hereby the data type propagation can be applied. Therefore, the transformation steps back to the RW_MATLAB_SETSIMPLEBLOCKS rule. On the other hand, in case the transformation does not find any match with the desired structure, there cannot be new block pairs where the data

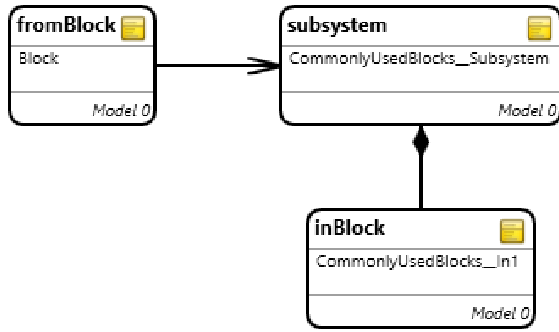


Figure 4. The transformation rule RW_MATLAB_SETSUBSYSTEMIN

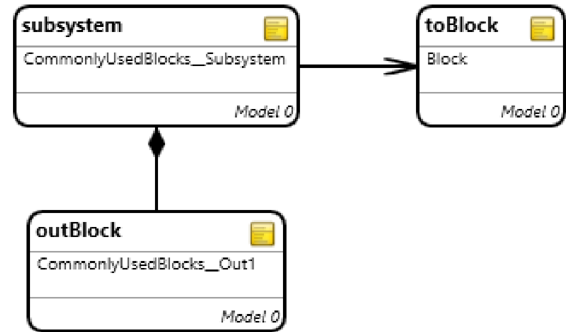


Figure 5. The transformation rule RW_MATLAB_SETSUBSYSTEMOUT

type propagation on the same level would be possible. Therefore, the transformation moves on to the RW_MATLAB)SETSUBSYSTEM_OUT rule.

Note, that this is the implementation of the lazy evaluation of conditional structures. In case the SETFIRSTINSUBSYSTEM algorithm returns *true* in the *until* block of the SETTYPES algorithm, then the condition cannot be evaluated as *true*, therefore, the execution returns to the body of the *repeat-until* block, that is, to the SETNEXTINHERIT algorithm, without checking the other conditions. This is implemented in the graph transformation-based solution as well. If the RW_MATLAB_SETSUBSYSTEMIN rule finds at least one match, then the control flow of the transformation navigates back to the RW_MATLAB_SETSIMPLEBLOCKS rule.

The bottom-up direction of the hierarchical data type propagation is implemented in the RW_MATLAB_SETSUBSYSTEMOUT transformation rule. This rule is shown in Figure 5. This rule is similar to the one presented before. However, in this case, the rule attempts to find a graph with this structure:

- The direction of the edge between the Subsystem and the simple block is reversed, it leads from the Subsystem to the block.
- The data type of the Outport block of the Subsystem, depicted as *outBlock*, is explicitly set.
- The data type of the target block, depicted as *toBlock* in Figure 5 starts out being set to "Inherited", which means it is calculated.

If a match is found, then the *OutDataTypeStr* attribute of the target block is set to the same value as the data type of the Outport block. Note, that the applicability of the transformation rule determines the direction of the control flow as well. In case the rule was applied at least once, the transformation returns to the RW_MATLAB_SETSIMPLEBLOCKS rule, otherwise, since there was no propagation, the transformation moves on to the RW_MATLAB_SETSIMPLEBLOCKBACKWARD transformation rule.

The transformation focuses on the type of data type propagation, where the direction of the propagation coincides with the direction of the connecting edge. This is called forward propagation. However, in some cases it is unavoidable to change the direction in order to specify the data type of a block. For this reason, the transformation implements the RW_MATLAB_SETSIMPLEBLOCKBACKWARD transformation rule, which realizes the backward propagation of data types. The structure of the rule corresponds with the RW_MATLAB_SETSIMPLEBLOCKS rule (Figure 3), but this time the *toBlock* must have its *OutDataTypeStr* attribute set explicitly, and the data type of the *fromBlock* is calculated.

Note however, that this rule is applied just once. If the match was successful, then the transformation returns to the RW_MATLAB_SETSIMPLEBLOCKS transformation rule, no matter how many more times the backward propagation could be applied. This behavior supports the forward propagation as opposed to the backward propagation.

In case the transformation does not find any match for the RW_MATLAB_SETSIMPLEBLOCKS, the transformation finishes by specifying the data type of the blocks. However, this does not mean that every block in the model has its data type set explicitly. In case there was not any block with set data type and there was no Constant block in the model, then the transformation could not propagate any information. Moreover, since the transformation prefers the forward data type propagation, the backward hierarchical propagation would not be invoked, which might lead to blocks with unspecified data type.

As the TYPEPROP algorithm suggests, after specifying the types of the blocks, their compatibility should be checked. This is the responsibility of the next three rules of the TRANS_TYPEPROP transformation. The structure of these rules (RW_MATLAB_VERIFYSIMPLEBLOCKS, RW_MATLAB_VERIFYFIRSTINSUBSYSTEM and RW_MATLAB_VERIFYAFTERSUBSYSTEM) and the rules themselves are very similar to the ones described before. As for the structure, these three rules realize a lazy evaluated *repeat-until* block, where the RW_MATLAB_VEREIFYSIMPLEBLOCKS is the only rule in the body of the block.

As for the rules, each of the three rules corresponds to one of the already discussed rules, but with different imperative code. These pairs are the following:

- The RW_MATLAB_VERIFYSIMPLEBLOCK transformation rule corresponds to the RW_MATLAB_SETSIMPLEBLOCKS rule,
- The RW_MATLAB_VERIFYFIRSTINSUBSYSTEM rule corresponds to the RW_MATLAB_SETSUBSYSTEMIN rule,
- The RW_MATLAB_VERIFYAFTERSUBSYSTEM rule corresponds to the RW_MATLAB_SETSUBSYSTEMOUT transformation rule.

The only differences between the setting and verifying rules are the constraints and the imperative code.

The constraints have to be different, since at this stage of the transformation it does not look for inherited data types. Moreover, the data types must be explicitly set to examine their compatibility. This compatibility check is implemented via imperative code. If the two data types are not compatible, for example, the data type of the source block is "int32" and the data type of the target block is "int16", there may be a possible arithmetic overflow. In these cases the transformation modifies the *OutDataTypeStr* attribute of the target block.

As their names suggest, the RW_MATLAB_VERIFYSIMPLEBLOCK transformation rule is responsible for the compatibility check of connected blocks in the same hierarchical level, and the RW_MATLAB_VERIFYFIRSTINSUBSYSTEM and RW_MATLAB_VERIFYAFTERSUBSYSTEM rules implement this check across hierarchical levels.

Since the direction of the signals correspond to the direction of the edges, there is no need for backward checking.

It is worth mentioning that a verifying transformation rule is successful if there is at least one block whose data type is changed because of compatibility issues. The application of the RW_MATLAB_VERIFYFIRSTINSUBSYSTEM and RW_MATLAB_VERIFYAFTERSUBSYSTEM rules is not exhaustive. If either of these rules is applied successfully, the transformation returns to the RW_MATLAB_VERIFYSIMPLEBLOCKS to perform compatibility checks on the affected hierarchical level.

C. The Termination of the Transformation

Before applying a graph transformation on a model, some of its properties must be examined. The termination of a transformation is arguably one of the most important properties.

Proposition 1. *The transformation* TRANS_TYPEPROP *always terminates.*

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

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

In VMTS, the application mode of a rule container, which contains the applicable rule, can be set to either "Once" or "Exhaustive". In case this attribute is set to "Once", then the VMTS attempts to apply the rule exactly once and moves on in the control flow based on the result of the matching. Therefore, the transformation cannot apply the rule indefinitely.

However, in case the transformation attempts to match a rule exhaustively, an nonterminating loop might arise. Therefore, these rules must be examined, in order to avoid their indefinite application. In the TRANS_TYPEPROP transformation the following rules are applied exhaustively:

- The RW_MATLAB_SETCONSTANT rule sets the *OutDataTypeStr* attribute based on the value of the Constant block. Since the constraint of the LHS states, that the *OutDataTypeStr* attribute must start being set to "Inherit", but the rule sets this property to a valid data type, therefore the rule cannot be applied on the matched block again. Considering that a Simulink model always contains only finite number of elements, the rule can be applied only a finite number of times.
- The RW_MATLAB_SETSIMPLEBLOCKS transformation rule sets the *OutDataTypeStr* attribute of a block based on one of its predecessor. In the LHS graph, the *OutDataTypeStr* attribute of the target block must start being set to "Inherited". However, the transformation modifies the value of this attribute to a valid data type. Therefore, this block cannot be matched as target block in the LHS graph. In this manner, a block can be matched as target block at most once. Since a Simulink model contains finite number of blocks, the rule cannot be applied infinite number of times.
- The RW_MATLAB_SETSUBSYSTEMIN rule is responsible for the data type propagation across hierarchical levels. The rule matches Inport blocks, which have their *OutDataTypeStr* attribute set to "Inherit". After the rule application, the value of the attribute is set based on the data type of the block connecting to the appropriate port of the Subsystem. Since the rule modifies the value of the attribute, which is the applicability condition, an Inport block can be matched by this rule at most once. Considering the number of block in the Simulink model is always finite, the rule cannot be applied indefinite number of times.
- The RW_MATLAB_SETSUBSYSTEMOUT rule implements the data type propagation across hierarchical levels as well, but this rule propagates the data type of the Outport block to the block connecting to the Subsystem on a higher level. The reasoning behind the bounded applicability of the rule is the same as before: each block can satisfy the constraint of the LHS graph

at most once, and since there exists only a finite number of blocks in the model, the rule is applied a bounded number of times.

- The RW_Matlab_VerifySimpleBlocks transformation rule is very similar to the RW_Matlab_SetSimpleBlocks rule, but it looks for incompatible data types. In case the rule finds a match, the data type of the target block is modified. However, there may be loops in the Simulink model, which makes it possible to apply the rule on the very same S (Source) and T (Target) blocks. This occurs, when the *OutDataTypeStr* of the S blocks is set to a type with broader range after the first application. In this case, the *OutDataTypeStr* of the T block is modified again. However, this cannot be repeated indefinitely since there is only a finite number of data types in Simulink, and the transformation rule is applied only when the data type of S is broader than the data type of T.

After examining the exhaustively applied transformation rules, the loops must be checked. Both the first and second part of the transformation implement loops. However, the rules in the first part, which explicitly set the data type of the blocks, cannot form an nonterminating loop. The reason for this behavior is the following: Each rule attempts to match exactly one block, whose *OutDataTypeStr* attribute starts off being set to "Inherit". This value is then modified by the rule. However, there is no rule in the transformation that sets the *OutDataTypeStr* attribute to a value that starts off being set to "Inherit". Therefore, the rules cannot be applied more often than there are blocks in the Simulink model.

This reasoning can be applied to the verification part of the transformation as well, but it must be extended. The successful application of either the RW_Matlab_VerifyFirstInSubsystem or the RW_Matlab_VerifyAfterSubsystem may cause the already matched S-T block pair to be matched again by the RW_Matlab_VerifySimpleBlocks rule. This happens when the application of the rules changes the data type of the S block to a data type with broader range. This way the data type of the T block is going to change again. However, there are only finite number of data types, and the types are never narrowed by any of the transformation. In this manner, the rules cannot form an nonterminating loop, at some point there will be no incompatible block pairs.

Since none of the transformation rules can be applied indefinitely and the transformation does not contain any nonterminating loop, it is proven that the transformation always terminates. ■

IV. Experimental Results

After presenting the algorithm and the transformation, this section shows a simple example to demonstrate the transformation functionality.

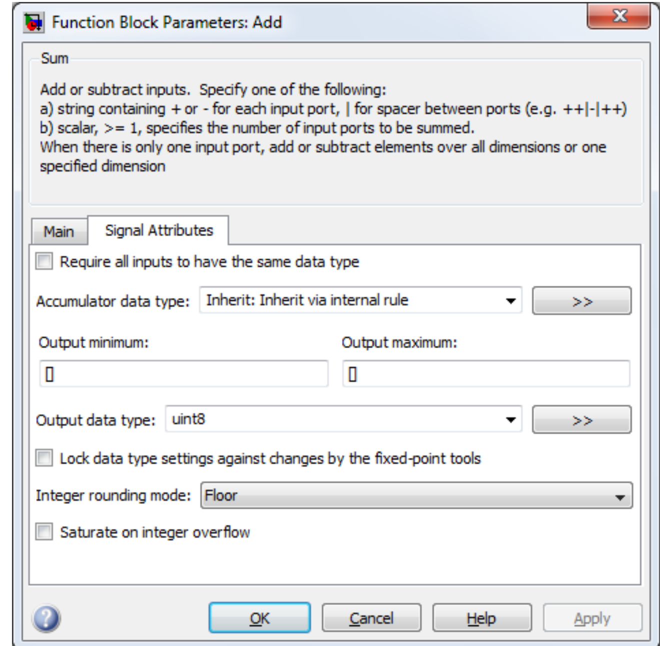


Figure 6. The properties of the Add block before the transformation

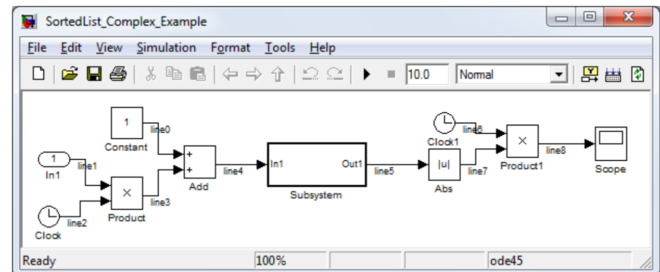


Figure 7. The root level of the example Simulink model

Figure 6 shows the properties dialog of an Add block. This Add block is the Add block from the model depicted in Figure 7. It can be seen that one of its sources is a Constant block and the other is a Product block. Both of these blocks have their data type set as calculated. Moreover, the data type of the In block, which is connected to the Product block, is *int16*. As Figure 6 shows, the data type of the Add block is set to *uint8*.

First, the transformation calculates the data types of the Constant blocks. In this implementation, the data type of the Constant block is set to *int8*.

Next, the data type propagation begins. In this case, the data type of the Product block is set to *int16* because one of its source blocks, that is, the In block, has this data type.

Finally, the verification part of the transformation changes the data type of the Add block from *uint8* to *int16*, as it is depicted in Figure 8. This modification is based on the data type of the Product block, which is one of the sources of the Add block. Note, that the other source of the Add

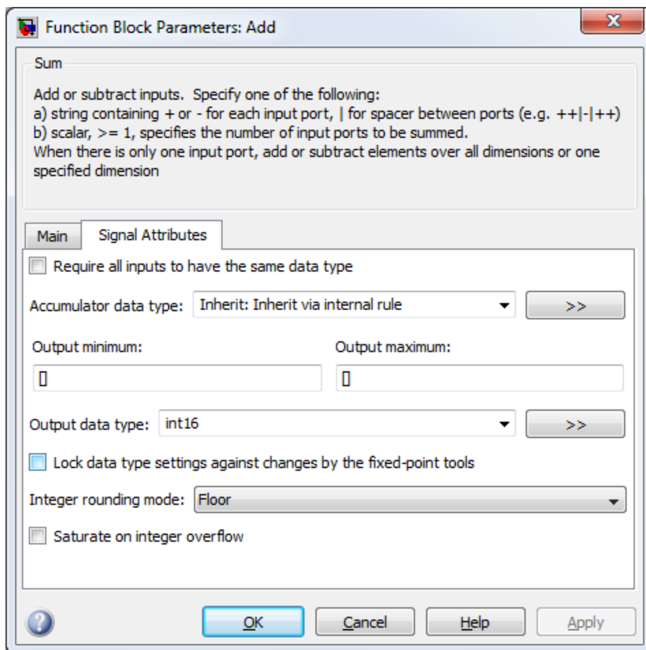
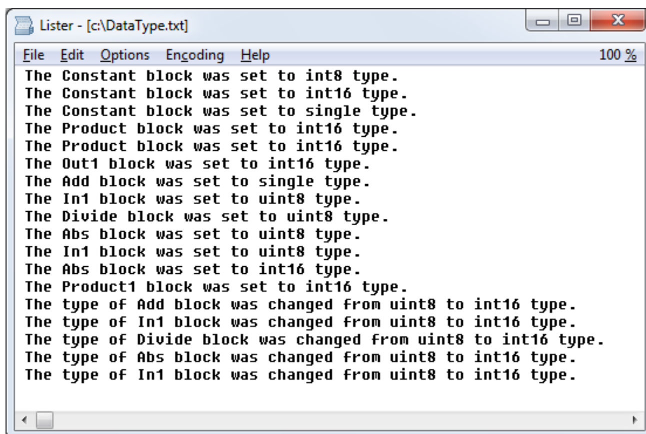


Figure 8. The properties of the Add block after the transformation




```
The Constant block was set to int8 type.
The Constant block was set to int16 type.
The Constant block was set to single type.
The Product block was set to int16 type.
The Product block was set to int16 type.
The Out1 block was set to int16 type.
The Add block was set to single type.
The In1 block was set to uint8 type.
The Divide block was set to uint8 type.
The Abs block was set to uint8 type.
The In1 block was set to uint8 type.
The Abs block was set to int16 type.
The Product1 block was set to int16 type.
The type of Add block was changed from uint8 to int16 type.
The type of In1 block was changed from uint8 to int16 type.
The type of Divide block was changed from uint8 to int16 type.
The type of Abs block was changed from uint8 to int16 type.
The type of In1 block was changed from uint8 to int16 type.
```

Figure 9. The change log of the transformation

block, that is, the Constant block, would also imply a data type change. The order of the matches in case of the same rule is not deterministic in the VMTS, so the data type of the Add block might have changed to *int8* first, and then to *int16*.

Since logging functionality is also implemented in the transformation, the data type changes can be conveniently tracked in a clear listing. This logging is shown in Figure 9.

Note, that the effectiveness of the transformation depends heavily on the implementation of the SETTYPEBASEDONVALUE and the COMPATIBLETYPES algorithms. Nevertheless, the presented example conveys the value and potential of graph transformations in software and system modeling.

V. CONCLUSION

Nowadays, Simulink is a popular tool in industry for modeling embedded systems. In order to precisely model the functionality of the modeled system, Simulink elaborates the source model. This elaboration is practically a form of model transformation, which is implemented in the Simulink code base.

Part of the elaboration process is specifying the data types of the blocks explicitly. In this paper, an algorithm is presented in detail, which is suitable for defining and verifying the data types of the different model elements by using data propagation.

Moreover, the algorithm implemented via graph transformation is also presented in this paper. This novel approach enables taking advantage of the benefits of model transformation such as platform independence and reusability. With this solution, the abstraction level of the data type propagation problem can be also raised. In order to use this graph transformation in practice, its termination was examined as well.

Finally, a simple example was given to illustrate the potential of the transformation.

Future work intends to study whether with the help of the graph transformation the other model elaboration steps can be implemented as well. These other steps include flatting of the source models and creating the execution list. In this manner, the abstraction level could be raised even further and additional benefits unlocked.

ACKNOWLEDGMENT

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 has been supported by the project "Talent care and cultivation in the scientific workshops of BME" financed by the grant TÁMOP - 4.2.2.B-10/1–2010-0009.

REFERENCES

[1] P. J. Mosterman, J. Zander, G. Hamon, and B. Denckla, "A computational model of time for stiff hybrid systems applied to control synthesis," *Control Engineering Practice*, vol. 20, no. 1, pp. 2–13, January 2012.

[2] P. J. Mosterman and J. Zander, "Advancing model-based design by modeling approximations of computational semantics," in *Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT 2011)*, September 2011, pp. 3–7.

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

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

[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] R. Hindley, "The principal type-scheme of an object in combinatory logic," *Transactions of the american mathematical society*, vol. 146, pp. 29–60, 1969.

[8] R. Milner, "A theory of type polymorphism in programming," *Journal of computer and system sciences*, vol. 17, no. 3, pp. 348–375, 1978.

[9] 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.

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

[11] 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.

[12] 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.

[13] "Simulink® 2012b - working with data types," http://www.mathworks.com/help/simulink/ug/working-with-data-types.html, 2012.

[14] "VMTS website," http://vmts.aut.bme.hu/, 2012.