From UML to SIMULINK CAAM: Formal Specification and Transformation Analysis

Luciana Foss [1]
Simone André da Costa Cavalheiro [1]
Lisane Brisolara de Brisolara [1]
Nícolas Nogueira Bisi [1]
Vinícius Steffens Pazzini [1]
Flávio Rech Wagner [2]

Abstract: UML and Simulink are attractive languages for embedded systems design and modeling. An automatic mapping from UML models to Simulink would be an interesting resource in a seamless design flow, allowing designers to use UML as modeling language for the whole system and at same time to use facilities for code generation based on Simulink. In a previous work, a UML to Simulink translation was prototyped using a Java implementation. In this paper, we present the formal definition of this translation using graph grammars, as well as its automation, which is supported by the AGG system. With the formalization of the metamodels and translation rules, we can guarantee the correctness of the translation.

1 Introduction

The increased amount of software in embedded systems and tight time-to-market constraints have motivated the investigation of strategies to successfully manage embedded software design and its complexity. The complexity is usually managed using abstraction by means of models, and automation can be a solution for the pressure for fast product delivery. This scenario motivates the use of model-driven engineering approaches.

The growing interest for using UML in software projects has led to an increase in its popularity. UML [1] supports the whole software development process, starting with requirements analysis and supporting object-oriented system specification, design, and modeling, thus being considered the standard modeling language for the software community. Moreover, UML has been used in most of model-driven design approaches recently proposed, since it is an open, standard language supported by several tools. In the embedded system community, mainly due to its high abstraction, UML has been proposed as the system-level modeling language [2], and its use in hardware software codesign has been investigated [3].

[1]Centro de Desenvolvimento Tecnológico, UFPEL, Caixa Postal 354 - Pelotas - RS
{lfoss,simone.costa,nnbisi,vspazzini}@inf.ufpel.edu.br
[2]Instituto de Informática, UFRGS, Caixa Postal 15064 - Porto Alegre - RS
flavio@inf.ufrgs.br

Embedded systems are usually compositions of subsystems, each of which may be based on a different Model of Computation (MoC) [4]. Each MoC, with associated modeling notation, is chosen with respect to its adequacy to the subsystem domain. UML is widely used in the development of event-based systems in the software engineering domain, since the UML modeling style maps nicely to the underlying object-oriented paradigm. However, UML is not well suited to model dataflow or other MoCs required by heterogeneous systems, like those present in most current or emerging embedded systems (e.g. mobile phones).

Simulink [5] is another language currently used for embedded systems design, with a focus on control and signal processing. This language supports dataflow and continuous time models, but it lacks the desired high-level abstraction and support to other software-oriented models (object-orientation, for example). Furthermore, Simulink provides code generation and simulation features. A Simulink-based multiprocessor system-on-chip (MPSoC) design flow has been proposed previously [6], in order to support modeling of embedded multiprocessor systems, including facilities for hardware and software co-simulation and code generation. This design flow uses as input a Simulink CAAM model, which is an extension of the Simulink language to model multiprocessor systems, combining Algorithm and Architecture Models in a single model.

Recent efforts have shown that both UML and Simulink are considered attractive for embedded system-level design [7, 8, 9, 10], a fact that motivates researchers to aim at finding a way to simultaneously exploit the benefits of both languages. In this way, a mapping from UML to Simulink CAAM models was proposed by Brisolara [8] to allow designers to use UML as modeling language for the whole system and at same time to use facilities for simulation and code generation based on Simulink. However, the automation of this translation was just prototyped using a Java implementation, without any considerations regarding the correctness of the translation, nor any formalization of the metamodels or of the translation rules. One way of covering such gaps is the use of formal methods. Since UML diagrams constitute a visual language, it is natural to consider that UML translations are based on rules that transform graphs. Graph grammars are a formal language that follow such approach [11] and offer various results concerning different types of analysis (like termination and confluence) that are suitable for model transformations.

This paper is an extended version of [12, 13]. With respect to the original version, we included two relevant contributions:

1. Formal representation of UML specifications and Simulink block diagrams as graphs;
2. The proofs of termination and confluence of the translation.

The use of a formal language as graph grammars to define the translation avoids possible ambiguities from a natural language description, as well as supports its automation by the Attributed Graph Grammar system (AGG tool) [14, 15]. A design flow that starts with a UML model and provides a mapping to a Simulink block diagram is proposed. The process of translation from UML to Simulink CAAM is semi-automatic and starts by constructing the initial graph from UML deployment and sequence diagrams. Using the AGG tool and applying the rules defined in the graph grammar, the initial graph is automatically transformed into a graph representing the corresponding Simulink model. The precise definition of UML specifications and Simulink block diagrams as graphs formally define the complete process of transformation. It is important to note that the proposed translation is independent of specific application. Moreover, the designer does not need to know all formal definitions of graph grammars to proceed with a translation, but just to use the AGG tool.

The graph grammar specification also allows the formal analysis of the mapping. The proofs of termination and confluence guarantee the existence and uniqueness of the outcoming Simulink model. Termination ensures that the transformation process finishes, while confluence ensures that the transformation results in a unique target model. Using the AGG tool these proofs were automatically performed.

The remaining of this paper is organized as follows. Section 2 presents an introduction to the adopted specification language, which is an attributed graph grammar with negative application conditions [11]. Section 3 introduces UML sequence and deployment diagrams and defines a graph representation of a UML specification. Section 4 describes a brief presentation of Simulink block diagrams and presents the graph representation of a Simulink CAAM model. Section 5 formally defines the mapping from UML to Simulink CAAM, and Section 6 details the transformation analysis. Finally, Section 7 contains concluding remarks.

2 Graph Grammars

In this section we review the main concepts about typed attributed graph grammars with application condition and negative application conditions, based on the double pushout approach (DPO-approach) [11]. Basically, a graph is composed by a set of vertices and edges connecting them, but they can be enriched with other information, like labels and attributes. Graphs in which vertices (and edges) can be assigned to attributes of some data type are often called attributed graphs. Attributed graphs generally consist of two parts: a graph-part and a data-part. The data-part includes an algebra which defines values and algebraic operations over these values. An algebra is a semantical model of a signature [16]. As an analogy, we can see a signature as the interface of a program and an algebra as the implementation of this program. An algebra homomorphism relates two algebras over the same signature, identifying their values and operations.

Definition 1 [Signature] *A* signature $\Sigma = (S, OP)$ *consists of a set* S *of sorts and a family*

$OP = (OP_{w,s})_{(w,s)\in S^*\times S}$ of operation symbols. For an operation $op \in OP_{w,s}$, we can write $op : w \to s$ or $op : s_0, \dots, s_n \to s$, where $w = s_0, \dots, s_n$. If $w = \varepsilon$, then the operation $op :\to s$ is called constant.

Definition 2 [Algebra and homomorphism] *Given a signature $\Sigma = (S, OP)$, a $\Sigma-$*algebra *$A = (\{A_s | s \in S\}, \{op_A | op \in OP\})$ is defined by:*

- *a carrier set A_s for each sort $s \in S$;*
- *a constant $c_A \in A_s$ for each constant $c :\to s \in OP$;*
- *a function $op_A : A_{s_0} \times \cdots \times A_{s_n} \to A_s$ for each operation $op : s_0, \dots, s_n \to s$.*

The set obtained by the disjoint union of all carrier sets of a Σ-algebra A is denoted by $\mathcal{U}(A)$, i.e., $\mathcal{U}(A) = \biguplus_{s\in S} A_s$.

Given two algebras A and A' of the same signature $\Sigma = (S, OP)$, a (partial) homomorphism *$h : A \to A'$, also called* Σ-homomorphism, *is a family $h = (h_s)_{s\in S}$ of functions $h_s : A_s \to A'_s$ such that:*

- *for each $c :\to s \in OP$, we have $h_s(c_A) = c_{A'}$;*
- *for each $op : s_0, \dots, s_n \to s \in OP$, we have $h_s(op_A(x_0, \dots, x_n)) = op_{A'}(h_{s_0}(x_0), \dots, h_{s_n}(x_n))$, for all $x_i \in A_{s_i}$.*

A homomorphism is total or injective if all functions are total or injective, respectively, and if all functions are bijective, it is an isomorphism.

Example 1 [Signature and algebra] *The following signature defines sorts, constants and operations of $STRING$ data type. This signature is used in the graph grammar that defines the translation from UML to Simulink.*

```
Signature STRING =
sorts: string, char
opns:  a: -> char
       empty: -> string
       next: char -> char
       ladd: char string -> string
       concat: string string -> string
       first: string -> char
```

A $STRING$-algebra D can be defined as follows:

$$
\begin{array}{rcl}
D_{char} & = & \{a,\ldots,z,A,\ldots,Z,0,\ldots 9,\_\} \\
D_{string} & = & D^*_{char} \\
a_D & = & a \in D_{char} \\
empty_D & = & \varepsilon \in D_{string} \\
next_D & : & D_{char} \to D_{char} \\
 & & next_D(a) = b,\ldots,next_D(z) = A, next_D(A) = B,\ldots, \\
 & & next_D(Z) = 0, next_D(0) = 1,\ldots,next_D(9) = \_, next_D(\_) = a \\
concat_D & : & D_{string} \times D_{string} \to D_{string} \\
 & & \forall s,t \in D_{string} : concat_D(s,t) = st \\
ladd_D & : & D_{char} \times D_{string} \to D_{string} \\
 & & \forall x \in D_{char} : \forall s \in D_{string} : ladd_D(x,s) = xs \\
first_D & : & D_{string} \to D_{char} \\
 & & first_D(\varepsilon) = a, \forall s \in D_{string} : first_D(s) = x_1, \\
 & & \textit{where } s = x_1 \ldots x_n
\end{array}
$$

A graph is defined by sets of vertices and edges. The set of vertices is partitioned into two sets: a set of graph vertices and a set of data vertices. And the set of edges is also partitioned into two sets: the sets of graph edges and node attribute edges. The graph vertices and edges define the graphical part of a graph, while the data vertices and node attribute edges define the data structure of this graph. In this approach the edges are directed, therefore the source and target of each edge must be defined. There are two functions determining the source and target of graph edges and two functions defining the source and target of node attribute edges. Graph edges have source and target in graph vertices, and node attribute edges associate graph vertices to data vertices. In order to relate two graphs, a graph morphism is defined, mapping all elements of one graph into the corresponding elements of the other. This mapping must preserve the source and target of each edge, i.e., if an edge e_1 is mapped to an edge e_2, the source and target vertices of e_1 must be accordingly mapped to the source and target of e_2.

Definition 3 [Graphs and graph morphisms] *A* graph *G is a tuple $(V_G, V_D, E_G, E_{NA}, src_G, tgt_G, src_{NA}, tgt_{NA})$ defined as follows:*

- *V_G and V_D are sets of graph and data vertices, respectively;*
- *E_G is the set of graph edges, which connect graph vertices, and E_{NA} is the set of node attribute edges, which connect graph vertices to data vertices;*
- *$src_G, tgt_G : E_G \to V_G$ are total functions, defining source and target of graph edges, respectively;*

- *$src_{NA} : E_{NA} \to V_G$ and $tgt_{NA} : E_{NA} \to V_D$ are total functions, defining source and target of node attribute edges, respectively;*

Given two graphs $G = (V_G^G, V_D^G, E_G^G, E_{NA}^G, src_G^G, tgt_G^G, src_{NA}^G, tgt_{NA}^G)$ and $H = (V_G^H, V_D^H, E_G^H, E_{NA}^H, src_G^H, tgt_G^H, src_{NA}^H, tgt_{NA}^H)$, a (partial) graph morphism *$f : G \to H$ is a tuple $(f_{V_G}, f_{V_D}, f_{E_G}, f_{E_{NA}})$ such that f commutes with all source and target functions, for example $f_{V_G} \circ src_G^G = src_G^H \circ f_{E_G}$. If there is no confusion, the subscribed indexes can be omitted, e.g., $f_{V_G}(v)$ can be written $f(v)$. A graph morphism is said to be total or injective if all its components are total or injective functions, respectively.*

An attributed graph is a graph combined with a data algebra over a signature Σ. In the signature, a set of attribute value sorts is established and the corresponding carrier sets are used to define the data vertices. The relation between two attributed graphs is determined by a graph morphism and an algebra homomorphism, which must preserve the mapping between data vertices.

Definition 4 [Attributed graphs and attributed graph morphisms] *Given a signature Σ, called data signature, an* attributed graph *is a pair $AG = (G, A)$, where A is a $\Sigma-$algebra, called data algebra, and G is a graph, such that $V_D = \mathcal{U}(A)$.*

Given two attributed graphs $AG^1 = (G^1, A^1)$ and $AG^2 = (G^2, A^2)$, a (partial) attributed graph morphism *$f : AG^1 \to AG^2$ is a pair (f_G, f_D), with a graph morphism $f_G : G_1 \to G_2$ and an algebra homomorphism $f_D : A^1 \to A^2$, such that the following diagram commutes for all $s \in S$.*

$$\begin{tikzcd}
A_s^1 \arrow[r, "f_{D_s}"] \arrow[d, hook] & A_s^2 \arrow[d, hook] \\
V_D^1 \arrow[r, "f_{G_{V_D}}"'] & V_D^2
\end{tikzcd}$$

An attributed graph morphism f is said to be total or injective if f_G and f_D are total or injective, respectively. Moreover, f is a monomorphism if f_G is injective and f_D is an isomorphism of $\Sigma-$algebras.

Attributed graphs combined with the concept of typing lead to the notion of typed attributed graphs. For typing the attributed graphs, a type graph over the final Σ-algebra is used. The typing is given by an attributed graph morphism associating each element of a graph to elements of the type graph. Two attributed graphs typed over the same type graph can be related by an attributed graph morphism, which must preserve the types of each graph element.

Definition 5 [Typed attributed graphs and typed attributed graph morphisms] *Given a signature $\Sigma = (S, OP)$, an* attributed type graph *is an attributed graph $TG = (T, A)$, where A is the final $\Sigma-$algebra (where all carrier sets of A are singletons). A* typed attributed graph *(AG, t) over TG consists of an attributed graph AG and a total attributed graph morphism $t : AG \to TG$, called typing morphism.*

Given two typed attributed graphs (AG^1, t^1) and (AG^2, t^2), typed over TG, a (partial) typed attributed graph morphism *$f : (AG^1, t^1) \to (AG^2, t^2)$ is a (partial) attributed graph morphism $f : AG^1 \to AG^2$, such that $t^2 \circ f = t^1$. A typed attributed graph morphism $f : (AG^1, t^1) \to (AG^2, t^2)$ is said to be total, injective or a monomorphism if $f : AG^1 \to AG^2$ is total, injective or a monomorphism, respectively.*

Typed attributed graphs over an attributed type graph TG and typed attributed graph morphisms form the category $\mathbf{AGraphs_{TG}}$ [16].

Example 2 [Typed attributed graphs] *A typed attributed graph is shown in Figure 1. Vertices are depicted as rectangles, which are divided into two parts, and edges are shaped as arrows connecting their source and target vertices. This graph is attributed over the $STRING$-algebra defined in Example 1. The data vertices and the node attribute edges associating them to graph vertices are inscribed in the bottom part of rectangles. For example, the vertex* IO *(on top left of Figure 1) has an attribute named* id, *whose value is* "IODevice", *that is, there is a node attribute edge* id *connecting the graph vertex* IO *to the data vertex* "IODevice". *The type graph is depicted in Figure 2, and the typing information of G is given by the labels (*IO, SAengine, var, SFunction, SASchedRes *and* getIO*) in the top part of rectangles and the labels (*src, tgt, in, arg *and* result*) on the arrows. This graph is a small fragment of the graph illustrated in Figure 9, which represents a UML specification: thread* "T3" *of type* SASchedRes *(allocated in the processor -* SAengine *-* "CPU2"*) calls the* getIO *method from an input device (*"IODevice" *of type* IO*) and returns its result in variable (*var*)* "a"*; the* getIO *result is an argument of the* "calc" *method (of type* SFunction*) of thread* "T3"*; finally, the result of* "calc" *method is returned in variable* "x".

A production defines the transformation from one graph to another one, identifying which elements should be preserved, consumed, or created. In this work we use the double pushout approach (DPO), where a production is defined by two total typed attributed graph morphisms $l : K_p \to L_p$ and $r : K_p \to R_p$, one mapping elements from the interface (K_p) to the left-hand side (L_p) and another one mapping elements from the interface to the right-hand side (R_p). L_p defines the elements that must be present in the graph for the production to be applied; R_p defines the result of application of the production; and K_p defines the context

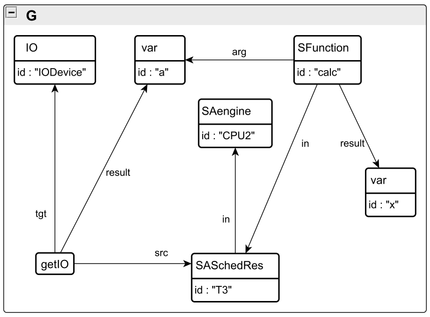


Figure 1. Attributed graph G typed over type graph TG in Figure 2.

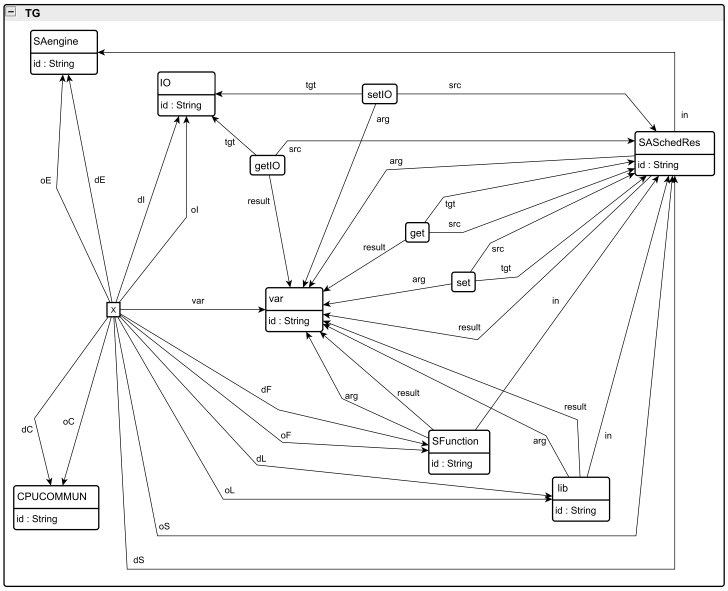


Figure 2. Type graph TG.

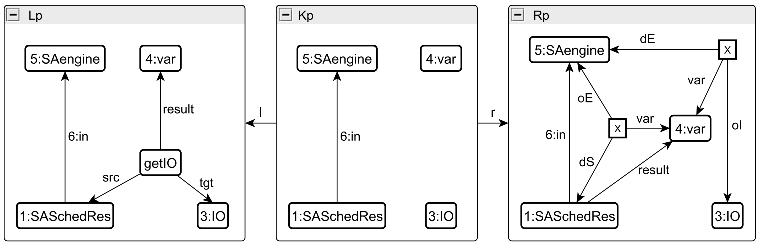
of the production application, i.e., elements that must be in the graph but are not deleted by application of the production. Elements of L_p that are not in the co-domain of l must be deleted, and elements in R_p that are not in the co-domain of r must be created. All graphs of p are typed over the same type graph T, with respect to a signature Σ, and the algebra associated to these graphs is the Σ-termalgebra [16] with the variables X used in p.

Definition 6 [Typed attributed graph productions] *Given an attributed type graph TG with data signature Σ, a* (typed attributed) graph production or rule $(p : L_p \overset{l}{\leftarrow} K_p \overset{r}{\rightarrow} R_p)$ *consists of three typed attributed graphs L_p (left-hand side), K_p (interface) and R_p (right-hand side), with a common $\Sigma-$algebra $T_\Sigma(X)$ (the $\Sigma-$termalgebra with variables X); and two typed attributed graph monomorphisms l and r.*

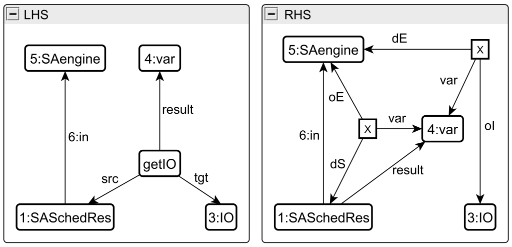
Example 3 [Graph production] *An example of production is showed in Figure 3(a). The morphisms l and r are defined by the numbers associated to each element of the graphs. The required elements to apply this production are those in graph L_p. Among these elements, those in K_p are preserved and the other ones are deleted. The created elements are those in R_p, but not in K_p. The AGG tool, which is used to support the analysis of the transformation, uses a more compact representation for a production. The AGG representation for the production p is showed in Figure 3(b). The graph K_p is omitted, but it is determined by the numbered elements. Elements in the left-hand side (LHS) or right-hand side (RHS) that have no associated number are deleted or created by the production, respectively. This production is used in the UML to Simulink mapping. It models the transformation of a* getIO *method call in the corresponding connections of Simulink blocks. The connections describe the data flow in the Simulink diagram: in this case, they are representing the flow of variable* “x” *from* IO *to* SASchedRes *passing by* SAengine *(vertices* X *specify these connections, whose source and target are indicated by edges with labels prefixed by* o *and* d, *respectively).*

The application of a production to a graph G is enabled if all elements in its left-hand side can be found in G , i.e., the left-hand side matches with a part of G. A match is defined as a total (typed attributed) graph morphism from the left-hand side of a production to a graph. It is total to ensure the presence of all needed elements in G. In this approach, if there is some edge connected to a deleted vertex or if there are two identified vertex, where one is preserved and the other one is deleted, the production cannot be applied.

Definition 7 [Match and gluing conditions] *Given a typed attributed graph production $(p : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, a typed attributed graph G and a total typed attributed graph morphism $m : L \rightarrow G$, with $X = (V_G^X, V_D^X, E_G^X, E_{NA}^X, src_G^X, tgt_G^X, src_{NA}^X, tgt_{NA}^X, D^X, t^X)$ for all $X \in \{L, K, R, G\}$, we can state the following definitions:*



(a) Morphisms l e r



(b) Production p in AGG tool

Figure 3. Graph production p.

- *the* identification points $IP = \{v \in V_G^L | \exists v' \in V_G^L : [v \neq v' \wedge m_{V_G}(v) = m_{V_G}(v')]\} \cup \{e \in E_i^L | \exists e' \in E_i^L : [e \neq e' \wedge m_{E_i}(e) = m_{E_i}(e')]\}$, *for all $i \in \{G, NA\}$ are graph elements in L that are identified by m;*
- *the* dangling points $DP = \{v \in V_G^L | \exists e \in (E_G^G - m_{E_G}(E_G^L)) : [m_{E_G}(v) = src_G^G(e) \vee m_{E_G}(v) = tgt_G^G(e)] \vee \exists e \in (E_{NA}^G - m_{E_{NA}}(E_{NA}^L)) : m_{E_{NA}}(v) = src_{NA}^G(e)\}$ *are graph vertices in L, whose image in G are source or target of an edge that are not in image of m.*

p and m satisfy the gluing condition if all identification and dangling points are elements preserved by p, i.e. they are in $l(K)$. In this case, m is called match *for p at G.*

The production application, or derivation, is defined as a double pushout in the category $\mathbf{AGraphs_{TG}}$ [16]. Intuitively, *G* can be transformed by removing the part matched by the production's left-hand side and adding its right-hand side.

Definition 8 [Typed attributed graph transformation] *Given a graph production ($p : L_p \overset{l}{\leftarrow} K_p \overset{r}{\rightarrow} R_p$), a typed attributed graph G and a match $m : L_p \rightarrow G$ for p at*

G. A direct (typed attributed) graph transformation *from G to the typed attributed graph H (with p at m), denoted by $G \overset{p,m}{\Rightarrow} H$, is given by the following double pushout (DPO) diagram in the category $\mathbf{AGraphs_{TG}}$, where (1) and (2) are pushouts:*

$$\begin{tikzcd}
L_q \arrow[d, "m"] & K_q \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[l, phantom, "(1)", shift right=4ex] \arrow[r, phantom, "(2)", shift right=4ex] & R_q \arrow[d, "n"] \\
G & D \arrow[l, "f"] \arrow[r, "g"'] & H
\end{tikzcd}$$

A (typed attributed) graph transformation from G_0 to G_k, *denoted by $G_0 \Rightarrow^* G_k$, is a sequence of direct graph transformations $G_0 \overset{p_1,m_1}{\Rightarrow} \cdots \overset{p_k,m_k}{\Rightarrow} G_k$.*

Example 4 [Direct graph transformation] *Figure 4 shows a direct derivation from G to H with the production p (see Figure 3). There is a match m of LHS in G, which is highlighted in the figure. In this case, G can be transformed into H by excluding the* getIO *vertex and its incident edges and adding two new vertices* X *and their incident edges, as well as the* result *edge between thread* T3 *and variable* a.

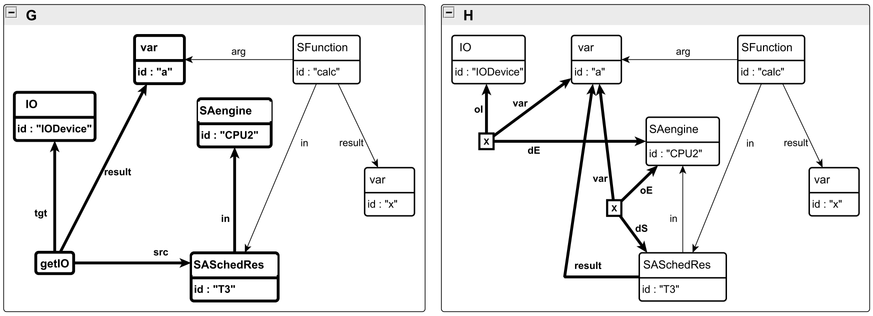


Figure 4. Direct graph transformation from G to H with production p in Figure 3.

Now we can define typed attributed graph grammars.

Definition 9 [Typed attributed graph grammar] *A* (typed attributed) graph grammar $GG = (\Sigma, TG, G_0, P)$ *consists of a data signature Σ, which defines the data values and operations used in all graphs of GG; a type graph TG attributed over the final Σ-algebra; an initial attributed graph G_0 typed over TG; and a set of graph productions P typed over TG.*

3 UML Sequence and Deployment Diagrams

A UML Sequence Diagram (SD) represents the interaction between objects. In the object-oriented approach, on which the UML language is based, objects interact through the exchange of messages. Thus, the main elements on this diagram are the objects and the messages, and, for each element, UML provides a graphical notation. The objects are instances of classes, while a message represents an invocation of an operation (or method).

From a Sequence Diagram a sequence of method invocations performed by the objects collaborating in a given scenario can be captured. This can be useful to represent the behavior of object-oriented code. Usually, several Sequence Diagrams are used to build dynamic views of the software system, one for each use case.

An object is represented by a rectangle with the names of the instance and of its class. Besides this information, stereotypes can be added to give a special meaning to an object. The UML standard defines some stereotypes, and the UML profiles (eg. MARTE and UML-SPT) use stereotypes to extend UML standard elements.

In UML2, the last version of the standard, Sequence Diagrams support the representation of iterations (loop), conditionals (alt or opt), parallelism (par), among other new syntax elements. For example, one can use a fragment with the operator loop to indicate that messages inside this fragment must be repeated.

In this work, in fact, the UML Sequence Diagram must be defined with some restrictions. For example, at the moment, loop, alt, opt or parallel operators are not supported. Besides, three stereotypes ($\langle\langle SASchedRes\rangle\rangle$, $\langle\langle SAScheduler\rangle\rangle$, $\langle\langle SAengine\rangle\rangle$) from the UML-SPT profile, and two new stereotypes are used to define an object type. Objects must be of one of four types: processor (denoted by $\langle\langle SAengine\rangle\rangle$), thread (denoted by $\langle\langle SASchedRes\rangle\rangle$), platform (denoted by $\langle\langle lib\rangle\rangle$) or IO (denoted by $\langle\langle IO\rangle\rangle$).

The $\langle\langle SASchedRes\rangle\rangle$ stereotype is used to identify objects representing threads or schedulable objects. The $\langle\langle SAScheduler\rangle\rangle$ stereotype can represent the scheduler, which is responsible for initializing the execution of a thread. $\langle\langle SAengine\rangle\rangle$ is used to represent a processing element (processor). Besides threads and processors, Platform and IO are other special objects. Platform represents a pre-defined library of functions, while IO objects model external systems (sensors, actuators, and etc). Objects can call its own methods or methods of the object Platform, and communication between threads must be done through gets and sets, to indicate receive or send operations, respectively. Besides, the method invocations from the object IO represent operations for reading input ports and writing output ports.

UML Deployment Diagrams can have nodes, representing processing elements, and artifacts or components, representing software elements. In this work, a Deployment Diagram is just used to define the number of processors and capture the mapping of threads to processors, so as to support the modeling of an MPSoC.

3.1 Syntax

In this section, we formally define the syntax of restricted sequence and deployment UML diagrams, which are used in the translation. In the following definitions, we denote the projection of the i-th element of a tuple x by $Pj_i(x)$ and define the set of stereotypes by $ST = \{SASchedRes, SAScheduler, SAengine, lib, IO\}$ and the alphabet by $N = N_{Ob} \cup N_{Ms} \cup N_X$, where N_{Ob} is a set of object names, N_{Ms} is a set of message names and N_X is a set of variable names. Moreover, there is a subset of N_{Ob} that defines the set of threads, i.e., $N_{Th} \subseteq N_{Ob}$ is the set of thread names. Let l be a list, $a \in l$ says that a is in some position of l, $l(i) = a$ says that a is the i-th element of l and $|l|$ denotes the length of l.

We first define the concepts of objects and messages and then use these elements to define the diagrams. Each diagram has an associated vocabulary, which defines the names of objects, messages and variables that are used as arguments or results of messages. An object has a stereotype, a name and a list of attributes.

Definition 10 [Object] *An* object *ob is defined by a tuple* (s, n, A_{ob})*, where* $s \in ST$ *is the stereotype of ob,* $n \in N_{Ob}$ *is the name of ob and* A_{ob} *is the set of attributes of ob.*

A message specifies a communication between two objects. It represents the invocation of an operation, which has a list of arguments and a result. Moreover, in a sequence diagram, the messages are partially ordered. This order relation defines the sequence in which messages are sent. In this way, a message is defined by its order of transmission, an operation name, a list of arguments, and a result.

Definition 11 [Message] *A* message m *is defined by a tuple* (i, n, l, r)*, with* $i \in \{1, 2, \ldots, k\}$ *and* $k \in \mathbb{N}$*, denotes the message order,* $n \in N_{Ms}$ *is the name of* m*,* $l \in N_X^*$ *is the list of arguments of* m *and* $r \in (N_X \cup \varepsilon)$ *is the result of* m*.*

In this work, each SD has a vocabulary and a thread associated to it and specifies the exchange of messages from the associated thread to other ones. An SD for a thread thr over a vocabulary N is defined by a set of objects, a set of messages, and two total functions defining the sender and the receiver of each message (which can be the same). These elements must satisfy some restrictions in order to specify a well-defined sequence diagram in our approach: (1) the associated thread thr must be in the set of objects; (2) all messages from an object to another one must be a get or set message, except messages sent to lib objects; (3) all messages, except the $main\_task$ message, must have their source in thr; (4) the order of transmission of each message must be unique in the set of messages; (5) there is exactly one object scheduler in the set of objects; (6) there is exactly one message $main\_task$; (7) the $main\_task$ is the first task in the order of transmission, and it is sent from the scheduler object to thr; (8) all messages to IO objects are either $getIO$ or $setIO$ messages; (9) get messages must be sent at the beginning of the thread execution; and (10) set messages must be sent at the end of the thread execution. A sequence diagram specification $\mathcal{SD}$ is the set of sequence diagrams associated to all thread names in the vocabulary (N_{Th}).

Definition 12 [Sequence Diagram] *A* sequence diagram *for a thread thr over the alphabet N, with $thr \in N_{Th}$, is defined by a tuple $SD_N^{thr} = (O, M, to, from, ord)$, where:*

- *O is a set of objects;*
- *M is a set of messages;*
- *$to : M \to O$ is a function defining for each message the object which receives it;*
- *$from : M \to O$ is a function defining for each message the object which sends it;*
- *$ord : M \to \mathbb{N}$ is a function defining an order for the messages.*

such that:

1. *$(SASchedRes, thr, A) \in O$;*
2. *for all $m \in M$, $from(m) \neq to(m)$ and $Pj_1(to(m)) \neq lib$ iff $Pj_2(m) = set\_x$ or $Pj_2(m) = get\_x$, with $x \in Pj_3(to(m))$;*
3. *for all $m \in M$, if $Pj_2(m) \neq main\_task$, then $Pj_2(from(m)) = thr$;*
4. *$m_1, m_2 \in M$, if $Pj_1(m_1) = Pj_1(m_2)$, then $m_1 = m_2$;*
5. *$\#\{o \in O \mid Pj_1(o) = SAScheduler\} = 1$;*
6. *$\#\{m \in M \mid Pj_2(m) = main\_task\} = 1$;*
7. *if $Pj_2(m) = main\_task$, then $Pj_1(m) = 1$ and $Pj_1(from(m)) = SAScheduler$ and $Pj_2(to(m)) = thr$;*
8. *if $Pj_1(to(m)) = IO$, then $Pj_2(m) = getIO$ or $Pj_2(m) = setIO$;*
9. *for all $m_1 \in M$, if $Pj_2(m_1) = get\_x$, then there not exist $m_2 \in M$ such that $Pj_1(m_2) < Pj_1(m_1)$ and $Pj_2(m_2) \neq get\_x$ or $Pj_2(m_2) \neq main\_task$;*
10. *for all $m_1 \in M$, if $Pj_2(m_1) = set\_x$, then there not exist $m_2 \in M$ such that $Pj_1(m_2) > Pj_1(m_1)$ and $Pj_2(m_2) \neq set\_x$ or $Pj_2(m_2) \neq main\_task$.*

A sequence diagram specification *over the alphabet* N *is the set of sequence diagrams over* N*, defined by* $\mathcal{SD} = \{SD_N^{thr} | thr \in N_{Th}\}$*. We denote by* $O_{\mathcal{SD}} = \{Pj_1(s) | s \in \mathcal{SD}\}$ *and* $M_{\mathcal{SD}} = \{Pj_2(s)_{thr} | s = SD_N^{thr} \in \mathcal{SD}\}$ *the sets of objects and messages of* $\mathcal{SD}$*, respectively.*

A deployment diagram is defined by two sets of objects: one defining the processors of the system and another one defining the threads of the system. Moreover, there is a function in defining where each thread will be executed.

Definition 13 [Deployment diagram] *A* deployment diagram *over the alphabet* N *is defined by a tuple* $DD = (P_{DD}, T_{DD}, in_{DD})$*, where* P_{DD} *is a set of objects defining the processors of the system,* T_{DD} *is a set of objects defining the threads of the system and* $in_{DD} : T_{DD} \to P_{DD}$ *is a total function mapping each thread to a processor where it will run.*

A UML specification is defined by a sequence diagram specification and a deployment diagram, such that all threads defined in the sequence diagrams are in the set of threads of the deployment diagram.

Definition 14 [UML specification] *A* UML specification *over the alphabet* N *is defined by a pair* $(DD, \mathcal{SD})$*, where* $DD = (P_{DD}, T_{DD}, in_{DD})$ *is a deployment diagram and* $\mathcal{SD}$ *is a sequence diagram specification, both over* N*, such that for all* $o \in O_{\mathcal{SD}}$*, if* $Pj_1(o) = SASchedRes$*, then* $o \in T_{DD}$*.*

From a UML specification, defined as above, we can obtain a typed attributed graph. The set of graph vertices V_G is composed by: a vertex for each processor object in the deployment diagram; a vertex for each object in the sequence diagram specification, except the scheduler; a vertex for each message in the sequence diagram specification; and a vertex for each variable name used as argument or result of each message. The set of data vertices V_D contains all strings defining the names of all objects (except for the scheduler object) and messages and all variables in N_X (set of variables in the vocabulary).

The set of graph edges E_G is defined by: one edge in_o for each thread in DD; one edge in_m for each message different from gets and sets; one source edge src_m and one target edge tgt_m for each get and set; one arg_{mj} edge for the j-th argument of each message; one $result_m$ for each message that has a result. The set of node attribute edges E_{NA} is formed by one edge id_v for all graph vertices, except those representing gets and sets messages.

The source of graph and node attribute edges are determined by the own index of each edge. The target of graph edges are specified according to its type: if it is an in_o edge, the target is the object associated to o by function in of DD; if it is an in_m or src_m edge, the target is the object associated to m by function $from$ of $\mathcal{SD}$; if it is a $result_m$ edge, the target is the vertex associated to the variable which corresponds to the message result; if it is an arg_{mj} edge, the target is the vertex associated to the variable that corresponds to the j-th argument of message m. The target of node attribute edges are determined according to its index and type: for all id_v edge, if v is a vertex associated to a variable, the target is the name of the variable; otherwise it is the name of the object or the message.

The algebra Alg is the STRING-algebra defined in Example 1, while the type graph is detailed in Figure 2. The graph vertex typing function is specified as follows: if the vertex represents an object, its type is given by the object stereotype; if the vertex represents a message to an IO object, its type is the name of the message; if the vertex represents a get or set message, its type is get or set, respectively; if the vertex corresponds to a message with the same source and target, its type is $SFunction$; if the vertex is a message to a lib object, its type is lib; if the vertex is a variable, its type is var. For all data vertices, its type is $string$. For each graph edge $type_m$, its type is given by $type$. For all node attribute edges, its type is id. And the algebra typing function associates each element of the algebra to its sort.

Definition 15 [Graph representation of a UML specification] *Given a UML specification $US = (DD, \mathcal{SD})$, over the alphabet N, and the signature $STRING$ (see Example 1). The graph representation of US is the typed attributed graph $UG = (AG, t : AG \to TG)$, defined by:*

- $AG = ((V_G, V_D, E_G, E_{NA}, src_G, tgt_G, src_{NA}, tgt_{NA}), Alg)$, *where:*

$$\begin{aligned}
V_G \quad &= \quad P_{DD} \cup O_{\mathcal{SD}} - \{o \in O_{\mathcal{SD}} | Pj_1(o) = Scheduler\} \cup M_{\mathcal{SD}} \cup V \\
&\quad\quad V = \{vert_x | x \in N_X \wedge (i,n,l,r) \in M_{\mathcal{SD}} \wedge (x = r \text{ } or \text{ } x \in l)\} \\
V_D \quad &= \quad \{n | (s,n,A) \in P_{DD} \vee (s,n,A) \in O_{\mathcal{SD}} \vee (i,n,l,r) \in M_{\mathcal{SD}}\} \cup \\
&\quad\quad \{x | x \in N_X \wedge (i,n,l,r) \in M_{\mathcal{SD}} \wedge (x = r \text{ } or \text{ } x \in l)\} \\
E_G \quad &= \quad In_O \cup In_M \cup Tgt \cup Src \cup Arg \cup Res \\
&\quad\quad In_O = \{in_o | o \in T_{DD}\} \\
&\quad\quad In_M = \{in_m | m \in M_{\mathcal{SD}} \wedge \\
&\quad\quad\quad\quad (from(m) = to(m) \vee Pj_1(to(m)) = lib)\} \\
&\quad\quad Tgt = \{tgt_m | m \in M_{\mathcal{SD}} \wedge from(m) \neq to(m)\} \\
&\quad\quad Src = \{src_m | m \in M_{\mathcal{SD}} \wedge from(m) \neq to(m)\} \\
&\quad\quad Arg = \{arg_{mj} | m = (i,n,l,r) \in M_{\mathcal{SD}} \wedge l \neq \varepsilon \wedge j \in \{1, \ldots, |l|\}\} \\
&\quad\quad Res = \{result_m | m \in M_{\mathcal{SD}} \wedge Pj_4(m) \neq \varepsilon\} \\
E_{NA} \quad &= \quad \{id_v | v \in V_G - \{m \in M_{\mathcal{SD}} | from(m) \neq to(m)\}\}
\end{aligned}$$

$$\forall e \in E_G : \quad src_G(e) = \begin{cases} o & \text{if } e = in_o \in In_O \\ m & \text{if } e = x_m \in (In_M \cup Tgt \cup Src \cup Res) \\ m & \text{if } e = arg_{mj} \in Arg \end{cases}$$

$$\forall e \in E_G : \quad tgt_G(e) = \begin{cases} in_{DD}(o) & \text{if } e = in_o \in In_O \\ from(m) & \text{if } e = x_m \in (In_M \cup Src) \\ to(m) & \text{if } e = x_m \in Tgt \\ vert_x & \text{if } e = result_m \in Res, \\ & \quad \text{where } Pj_4(m) = x \\ vert_{l(j)} & \text{if } e = arg_{mj} \in Arg, \\ & \quad \text{where } Pj_3(m) = l \end{cases}$$

$$\forall id_v \in E_{NA} : \quad src_{NA}(id_v) = v$$

$$\forall id_v \in E_{NA} : \quad tgt_{NA}(id_v) = \begin{cases} Pj_2(v) & \text{if } v \in (P_{DD} \cup O_{\mathcal{SD}} \cup M_{\mathcal{SD}}) \\ x & \text{if } v = vert_x \wedge x \in N_X \end{cases}$$

Alg is the $STRING$-algebra D, defined in Example 1.

- *TG is depicted in Figure 2*
- $t = (t_{V_G}, t_{V_D}, t_{E_G}, t_{E_{NA}}, t_D)$ *is defined by:*

$$\forall v \in V_G : t_{V_G}(v) = \begin{cases} Pj_1(v) & \text{if } v \in (P_{DD} \cup O_{\mathcal{SD}}) \\ Pj_2(v) & \text{if } v \in M_{\mathcal{SD}} \text{ and } Pj_1(to(v)) = IO \\ get & \text{if } v \in M_{\mathcal{SD}} \text{ and } Pj_2(v) = get\_x \\ set & \text{if } v \in M_{\mathcal{SD}} \text{ and } Pj_2(v) = set\_x \\ SFunction & \text{if } v \in M_{\mathcal{SD}} \text{ and } to(v) = from(v) \\ lib & \text{if } v \in M_{\mathcal{SD}} \text{ and } Pj_1(to(v)) = lib \\ var & \text{if } v = vert_x \in N_X \end{cases}$$

$$\forall v \in V_D : t_{V_D}(v) = string$$

$$\forall e \in E_G : t_{E_G}(e) = type, \text{ where } e = type_m$$

$$\forall e \in E_{NA} : t_{E_{NA}}(e) = id$$

$$\forall x \in Alg_s : t_{Ds}(x) = s, \text{ for all } s \in S_{STRING}$$

4 Simulink Block Diagrams

Traditionally, the functional block (FB) modeling approach has been used by the signal processing, industrial automation, and control engineering communities for the development of embedded systems. These models are widely accepted in industrial design, driven by an extensive set of design tools, as for instance, Matlab/Simulink from MathWorks. Features like modularity, abstraction level, and reusability contributed to the popularity of this modeling approach. In the functional block (FB) approach, applications are designed by connecting several FBs. Each FB output must be connected with an appropriate input, coming from an FB or another model element.

Simulink supports several models of computation, which means we can find two Simulink models with different semantics. For that reason, in this work, we focus on the discrete-time Simulink models. In this context, a Simulink block represents a function that takes inputs and produces outputs. Examples of Simulink blocks include User-defined (SFunction), discrete delay, and pre-defined blocks, such as mathematical operations. Using Simulink, pre-defined components from libraries can be reused to build a solution (adders, multipliers, switches, etc), and, when this is not possible, an SFunction block can be used. SFunction blocks allow the user to link a C-code to a block to represent the desired behavior for a block.

Other elements in a Simulink model are the connections or links. The Simulink link connects one output port of a block (or subsystem) to one or more input ports of one or more blocks (or subsystems), representing a data flow in the system. A connection can start in an output port and arrive at several input ports from different components (named a Branched-Line), which means that the same data can be sent to several blocks. However, an input port of a block cannot be connected to several outputs of other blocks.

An interesting feature from Simulink is the hierarchy of composition. This feature allows a complex system to be represented at several levels. At the high level, the system is composed of interconnected subsystems. After that, each subsystem can be decomposed in several interconnected blocks (or yet subsystems). Thus, the Simulink Subsystem is a special modeling element, which can contain blocks (pre-defined or user-defined blocks), links, gates, and other subsystems, to represent hierarchical composition, and conditionals, such as the for-loop iteration or the if-then-else structure. A subsystem is defined with its input ports and output ports. Using the Simulink GUI, a double-click in the subsystem graphical representation allows one to see its decomposition at the next hierarchical level, where the subsystem is represented by interconnected blocks. At this level, the input and output ports are represented by Input and Output Gates, respectively.

In this work, we are using an extension of Simulink, where the use of subsystems is also used to represent CPUs, threads, and communication resources, following the Simulink CAAM [17]. Similarly to the stereotypes used in UML, parameters defined in the subsystem description are used to indicate when a special semantics should be considered for this subsystem. Since we have UML-SPT stereotypes to describe some of these aspects we keep them in the Simulink graph representation, using the stereotype $\langle\langle SAEngine\rangle\rangle$ for CPU subsystems, and $\langle\langle SASchedRes\rangle\rangle$ for Thread subsystems. However, for communication subsystems (intra-SS and inter-SS subsystems) we use $CPUCOMM$, changing the id to indicate communication between threads allocated to the same CPU ($id =$ "$intra$") and to different CPUs ($id =$ "$inter$").

4.1 Syntax

In this section, the syntax of Simulink block diagrams and their graph representations are determined. Basically, a Simulink block diagram is composed by blocks (or subsystems) and connections. Therefore, a block diagram is defined by a set of blocks; a set of links (connections); a function defining the subsystems; two sets determining the input and output ports; and two functions defining the source and target ports of each link. All elements (blocks, links and ports) have a unique identifier, allowing the system to have different instances of the same element. Moreover, blocks and links have names; and blocks have also a type identifying the parameters for each component. The hierarchy of subsystems is defined by a function that associates each block to a set of its component blocks. This means that, if the associated set is not empty, the block is a subsystem composed of blocks in the set, otherwise it is a simple block. Each port is associated to a block, and the links associate one source port to a list (not empty) of target ports.

A Simulink block diagram is well defined if some restrictions are satisfied: (1) the identifier of each block is unique in the set of blocks; (2) the identifier of each link is also unique in the set of links; (3) a block cannot be a component of itself; (4) the subsystem hierarchy is not symmetric, i.e., if a block is a component of another one, the latter cannot be a component of the former; (5) each block can be a component of only one subsystem; (6) links must connect ports according to a given set of restrictions. Components of a subsystem can only be connected to the subsystem or to other components in this subsystem. In this way, a link can connect blocks if its source is a subsystem and its target is a component of this subsystem, or vice-versa ($P_1 - P_2$); or if its target and source are the same block (P_3); or if both target and source blocks are components of the same subsystem (P_4); or if both target and source blocks are not components of any subsystem (P_5). In addition, depending on the type of link ($P_1 - P_5$), the connected ports must be either an input or an output port. Links of type P_1 (or P_2) connect input to input ports (or output to output ports); links of type P_3 connect input to output ports; and, finally, links of types P_4 and P_5 connect output to input ports.

Definition 16 [Simulink block diagram] *Given a set ID of identifiers and a set NM of names (strings), a* Simulink block diagram *is defined by a tuple $SB = (B, L, sub, P_{in}, P_{out}, sl, tl)$ where:*

- $B = \{(id, name, type) | id \in ID \wedge name \in NM \wedge type \in \{SAengine, SASchedRes, SFunction, lib, CPUCCOMUN, IO\}\}$ *is the set of blocks;*
- $L = \{(id, name) | id \in ID \wedge name \in NM\}$ *is the set of links;*
- $sub : B \to \mathcal{P}(B)$ *is a total function associating each block to the set of its components (blocks).*

- $P_{in} = \{(id, b)_{in} | id \in ID \wedge b \in B\}$*, is the set of all input ports;*
- $P_{out} = \{(id, b)_{out} | id \in ID \wedge b \in B\}$*, is the set of all output ports;*
- $sl : L \to (P_{in} \cup P_{out})$ *is a total function defining the source port of each link;*
- $tl : L \to (P_{in}^{+} \cup P_{out}^{+})$ *is a total function defining the list of target ports of each link;*

such that:

1. $\forall b_1, b_2 \in B : \text{if } Pj_1(b_1) = Pj_1(b_2) \text{ then } b_1 = b_2$*;*
2. $\forall l_1, l_2 \in L : \text{if } Pj_1(l_1) = Pj_1(l_2) \text{ then } l_1 = l_2$*;*
3. $\forall b \in B : b \notin sub(b)$*;*
4. $\forall b_1, b_2 \in B : \text{if } b_1 \in sub(b_2) \text{ then } b_2 \notin sub(b_1)$*;*
5. $\forall b_1, b_2, b_3 \in B : \text{if } b_1 \in sub(b_2) \wedge b_1 \in sub(b_3) \text{ then } b_2 = b_3$*;*
6. $\forall l \in L :$

$$\begin{aligned}
[\forall pd \in tl(l) : \quad & \textstyle\bigvee_{i \in \{1,\ldots,5\}} P_i(pd, sl(l)) \wedge \\
& (\text{if } P_1(pd, sl(l)) \text{ then } pd, sl(l) \in P_{in})) \wedge \\
& (\text{if } P_2(pd, sl(l)) \text{ then } pd, sl(l) \in P_{out}) \wedge \\
& (\text{if } P_3(pd, sl(l)) \text{ then } pd \in P_{out} \wedge sl(l) \in P_{in}) \wedge \\
& (\text{if } P_4(pd, sl(l)) \vee P_5(pd, sl(l)) \text{ then } pd \in P_{in} \wedge sl(l) \in P_{out})]
\end{aligned}$$

where

$$\begin{aligned}
P_1(pd, po) &\overset{def}{=} Pj_2(pd) \in sub(Pj_2(po)) \\
P_2(pd, po) &\overset{def}{=} Pj_2(po) \in sub(Pj_2(pd)) \\
P_3(pd, po) &\overset{def}{=} Pj_2(pd) = Pj_2(po) \\
P_4(pd, po) &\overset{def}{=} \exists b \in B : Pj_2(pd), Pj_2(po) \in sub(b) \\
P_5(pd, po) &\overset{def}{=} \nexists b \in B : Pj_2(pd) \in sub(b) \wedge \nexists b \in B : Pj_2(po) \in sub(b)
\end{aligned}$$

In the following, we have $oX = \{oE, oI, oC, oS, oL, oF\}$ and $dX = \{dE, dI, dC, dS, dL, dF\}$. Moreover, we say that an edge e has type oX or dX, if $t_{E_G}(e) \in oX$ or $t_{E_G}(e) \in dX$, respectively.

A Simulink CAAM block diagram can be represented by a typed attributed graph. This graph must be typed over the type graph T depicted in Figure 5. Vertices of graph T represent a block ($SFuncion$, lib, $CPUCOMMUN$, $SAengine$, $SASchedRes$ or IO), or a variable (var) or links (X). In fact, a link is represented by a vertex of type X; one edge of type oX, indicating the source block of the link; one edge of type dX, defining the target block of the link; and one edge of type var, determining the data which are transmitted through the link. A graph can represent a block diagram if it satisfies the following restrictions: (1) the graph must be typed over the type graph T depicted in Figure 5; $(2-4)$ a vertex can be the source of at most one edge of each type in, $result$ and id; (5) two edges of type in cannot connect the same vertices in opposite directions; (6) each vertex of type X must have exactly one outgoing edge of each type var, oX and dX; (7) if there is a link (a vertex of type X and edges of types oX and dX with source in X) connecting two blocks, then either one of the blocks is a component of the other or both are components of the same subsystem.

Definition 17 [Graph representation of a Simulink block diagram] *A typed attributed graph $GT = (G, t)$ is a* graph representation of a Simulink block diagram *(gSD) if it is typed over T, such that:*

1. *T is the type graph depicted in Figure 5;*

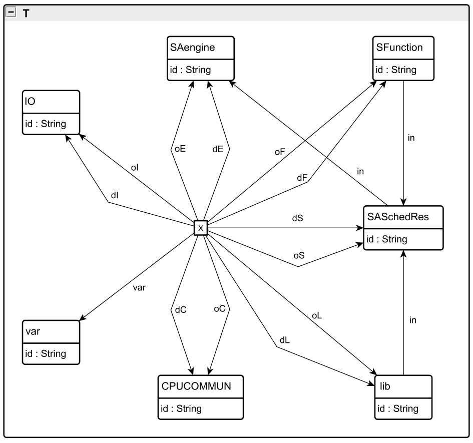


Figure 5. Type graph for gSDs.

2. $\forall e_1, e_2 \in E_G$: if $t(e_1) = t(e_2) = in \wedge src(e_1) = src(e_2)$ then $e_1 = e_2$*;*

3. $\forall e_1, e_2 \in E_G$: if $t(e_1) = t(e_2) = result \wedge src(e_1) = src(e_2)$ then $e_1 = e_2$*;*

4. $\forall e_1, e_2 \in E_{NA}$: if $t(e_1) = t(e_2) = id \wedge src(e_1) = src(e_2)$ then $e_1 = e_2$*;*

5. $\nexists e_1, e_2 \in E_G : [t(e_1) = t(e_2) = in \wedge src(e_1) = tgt(e_2) \wedge src(e_2) = tgt(e_1)]$*;*

6. $\forall v \in V_G$: if $t(v) = X$ then $\#Var(v) = \#Orig(v) = \#Dest(v) = 1$, *where:*

- $Var(v) = \{e \in E_G | t(e) = var \wedge src(e) = v\}$
- $Orig(v) = \{e \in E_G | t(e) \in oX \wedge src(e) = v\}$
- $Dest(v) = \{e \in E_G | t(e) \in dX \wedge src(e) = v\}$

7. $\forall x \in V_G$:

```
if      $t(x) = X$
then    $\forall e \in E_G$ :
        if      $t(e) = in \wedge (oX_V(x) \in src(e) \vee dX_V(x) \in src(e))$
        then    $P(e, oX_V(x), dX_V(x)) \wedge P(e, dX_V(x), oX_V(x))$

where   $oX_V(x) \overset{def}{=} v$ such that $e \in E_G : [src(e) = x \wedge t(e) \in oX \wedge tgt(e) = v]$
        $dX_V(x) \overset{def}{=} v$ such that $e \in E_G : [src(e) = x \wedge t(e) \in dX \wedge tgt(e) = v]$
        $P(e, v_o, v_d) \overset{def}{=}$   if $src(e) = v_o$,
                          then $(tgt(e) = v_d) \vee$
                              $(\exists e' \in E_G : [t(e') = in \wedge src(e') = v_d \wedge$
                                          $tgt(e) = tgt(e')])$
```

Since each vertex v of type X has exactly one outgoing edge of each type var, oX or dX, we use $var(v)$, $oX(v)$ and $dX(v)$ to denote them, respectively. Moreover, for a vertex $v \in V_G$ with an outgoing edge $e \in E_{NA}$ of type id, we denote the $tgt_{NA}(e)$ by $id(v)$.

Given a graph representing a Simulink model, we can obtain the Simulink block diagram corresponding to this graph.

Each vertex v of type $SFuncion$, lib, $CPUCOMMUN$, $SAengine$, $SASchedRes$ or IO defines one block b of the set of blocks of the Simulink diagram. For each block b, its identifier is defined by v, its name by the value of attribute id of v and its type by the v type.

A set of vertices of type X defines a link l associated to a variable v. The identifier of l is determined by the set of X vertices and its name by the value of attribute id of v. Each vertex X in the set that defines an identifier L_o^v must have the same source block o (that is, X is connected to o by an edge of type oX) and the same associated variable v. Besides, all vertices X in L_o^v must satisfy one of the following restrictions: each X must have its source in a component of a subsystem and its target either in a block inside the same subsystem or in the subsystem itself (that is, X belongs to Int_o^v and l is of type Int_o^v); X must have source and target in blocks that are not components of any subsystem (that is, X belongs to Ext_o^v); X must have source in a subsystem and target either in a component of this subsystem or in the subsystem itself (that is, X belongs to Mix_o^v).

The edges of type in determine the hierarchy of subsystems. Each in edge with source in block b' and target in block b establishes that b' is a component of subsystem b.

The set of links (created from X vertices together with its outgoing edges oX, dX and var) specifies the set of input and output ports of each block. Each link l of type Mix_o^v defines an input port of block o identified by v. In case that l has an X vertex with target in some component d of block o, then it defines an input port in d. If l has an X vertex with both source and target in block o, then it also specifies an output port in o. Each link l of type Ext_o^v defines input ports in the target of each X in Ext_o^v and specifies an output port in block o. Each link l of type Int_o^v defines an input port in a block d if there is an X in Int_o^v with target in d such that o and d are components of the same subsystem. Besides that, each link l of type Int_o^v determines an output port in o. If l has an X vertex with target in a subsystem d which contains o, then it specifies an output port in d.

The source of each link of type Mix_o^v is the input port of o identified by v. The source of each link of type Int_o^v or Ext_o^v is the output port of o identified by v. The target d of each X in a link l defines a target port p of link l in block d identified by v . If l is of type Ext_o^v, p is an input port. If l is of type Int_o^v and d and o are in the same subsystem, then p is an input port. If l is of type Mix_o^v and d is a component of subsystem o, then p is also an input port. Otherwise, p is an output port.

Definition 18 [From a gSD to a Simulink block diagram] *Given a graph gSD $GT = ((V_G, V_D, E_G, E_{NA}, src_G, tgt_G, src_{NA}, tgt_{NA}, Alg), t : G \to T)$, a Simulink block diagram equivalent to GT is defined by $SB = (B, L, sub, P_{in}, P_{out}, sl, tl)$, where:*

- $B = \{(v, id(v), t(v)) \mid v \in V_G \wedge t(v) \notin \{X, var\}\}$;
- $L = \{(L_o^v, id(v)) \mid L_o^v \neq \emptyset \wedge o \in B \wedge v \in V_G \wedge t(v) = var\}$, *where* $L_o^v = Int_o^v \vee L_o^v = Ext_o^v \vee L_o^v = Mix_o^v$, with

$$\begin{aligned}
Int_o^v = \quad & \{x \in V_G \mid P(x, o, v) \wedge v_d = tgt(dX(x)) \wedge \\
& \qquad \exists w \in B : [o \in sub(w) \wedge (v_d = Pj_1(w) \vee SB(v_d, w))]\} \\
Ext_o^v = \quad & \{x \in V_G \mid P(x, o, v) \wedge v_d = tgt(dX(x)) \wedge \\
& \qquad \nexists w \in B : o \in sub(w) \wedge \nexists w \in B : SB(v_d, w)\} \\
Mix_o^v = \quad & \{x \in V_G \mid P(x, o, v) \wedge (tgt(dX(x)) = Pj_1(o) \vee SB(tgt(dX(x)), o))\}
\end{aligned}$$

$$\begin{aligned}
\textit{with} \quad P(x, o, v) &\overset{def}{=} t(x) = X \wedge tgt(oX(x)) = Pj_1(o) \wedge tgt(var(x)) = v \\
SB(v, o) &\overset{def}{=} (v, id(v), t(v)) \in sub(o)
\end{aligned}$$

- $\forall b \in B : sub(b) = \{b' \in B \mid e \in E_G \wedge t_{E_G}(e) = in \wedge src_G(e) = Pj_1(b') \wedge tgt(e) = Pj_1(b)\}$

- $P_{in} = M_1 \cup M_2 \cup E \cup I$, *where*
$$\begin{array}{rcl} M_1 = \{(v,o)_{in} & | & (Mix_o^v, n) \in L\} \\ M_2 = \{(v,d)_{in} & | & (Mix_o^v, n) \in L \wedge x \in Mix_o^v \wedge tgt(dX(x)) = v_d \wedge \\ & & d = (v_d, id(v_d), t(v_d)) \in sub(o)\} \\ E = \{(v,d)_{in} & | & (Ext_o^v, n) \in L \wedge x \in Ext_o^v \wedge tgt(dX(x)) = v_d \wedge \\ & & d = (v_d, id(v_d), t(v_d))\} \\ I = \{(v,d)_{in} & | & (Int_o^v, n) \in L \wedge x \in Int_o^v \wedge tgt(dX(x)) = v_d \wedge \\ & & d = (v_d, id(v_d), t(v_d)) \wedge o \notin sub(d)\} \end{array}$$

- $P_{out} = M \cup EI \cup I$
$$\begin{array}{rcl} M = \{(v,o)_{out} & | & (Mix_o^v, n) \in L \wedge x \in Mix_o^v \wedge tgt(dX(x)) = Pj_a(o)\} \\ EI = \{(v,o)_{out} & | & (Ext_o^v, n) \in L \vee (Int_o^v, n) \in L\} \\ I = \{(v,d)_{out} & | & (Int_o^v, n) \in L \wedge x \in Int_o^v \wedge tgt(dX(x)) = v_d \wedge \\ & & d = (v_d, id(v_d), t(v_d)) \wedge o \in sub(d)\} \end{array}$$

- $\forall l \in L : sl(l) =$
$$\begin{cases} (v,o)_{in} & \text{if } l = (Mix_o^v, n) \\ (v,o)_{out} & \text{if } l = (Int_o^v, n) \vee l = (Ext_o^v, n) \end{cases}$$

- $\forall l \in L : tl((L_o^v, n)) = ld$, *where*
$$\begin{cases} (v,d)_{in} \in ld & \text{if } x \in L_o^v \ \wedge v_d = tgt(dX(x)) \wedge d = (v_d, id(v_d), t(v_d)) \wedge \\ & ((L_o^v = Mix_o^v \Rightarrow d \in sub(o)) \vee (L_o^v = Int_o^v \Rightarrow o \notin sub(d))) \\ (v,d)_{out} \in ld & \text{if } x \in L_o^v \ \wedge v_d = tgt(dX(x)) \wedge \\ & d = (v_d, id(v_d), t(v_d)) \wedge L_o^v \neq Ext_o^v \wedge \\ & ((L_o^v = Mix_o^v \Rightarrow d = o) \vee (L_o^v = Int_o^v \Rightarrow o \in sub(d))) \end{cases}$$

5 Specifying the Mapping from UML to Simulink

In this section, we propose a design flow to use UML as modeling language of a system and provide the generation of a Simulink model. A graph grammar specification defines the mapping from a graph representation of a UML specification to a graph representation of a Simulink block diagram. In this way, designers have to know a single modeling language with a high-level abstraction for systems specification. The translation also handles the allocation of processors, the mapping of threads to processors, and the insertion of required Simulink ports and dataflow connections, in order to generate a synthesizable Simulink CAAM model according to the design flow proposed in [6].

The process of translation is illustrated in Figure 6. It starts by constructing the initial graph from sequence and deployment diagrams (which must respect Definitions 12, 13 and 14). The specification of this graph is elaborated in the AGG tool [15] according to Definition 15. Using the AGG simulator and applying the defined rules, the initial graph is transformed into a graph representing the corresponding Simulink CAAM model. The final graph is then described in the input language of Simulink. The process is still semi-automatic, where only the transformation between the initial and final graphs is automated, using the AGG tool.

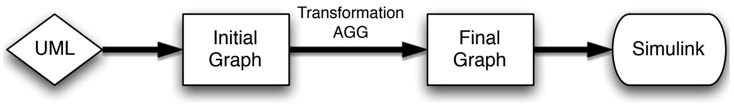


Figure 6. Process of transformation of UML diagrams into Simulink models.

In the following, we present the process of translating UML diagrams into Simulink models.

5.1 Initial graph

In order to translate a set of UML diagrams into a Simulink diagram, a graph grammar that models such transformation must be defined. The translation rules used in this grammar are those defined in Section 5.2, but the initial graph changes according to the diagrams to be translated. For instance, in order to illustrate the graph grammar specification, we translate the UML sequence and deployment diagrams detailed in Figure 7 into the Simulink block diagram depicted in Figure 8. This example describes a hypothetical system.

According to the mapping proposed in [8], the deployment diagram from Figure 7 defines the processing elements used in the multiprocessor system (nodes decorated with the $\langle\langle SAEngine\rangle\rangle$ stereotype), which are $CPU1$ and $CPU2$, besides the threads (indicated by the $\langle\langle SASchedRes\rangle\rangle$ stereotype) allocated for each CPU. The sequence diagram represents the behavior of threads ($T3$, $T1$ and $T2$ in this order) using message exchanges according to the object-oriented approach. In this diagram, rectangles represent objects, as for instance, the threads $T1$, $T2$ and $T3$. Communications between threads are represented by set and get messages, as well as data input and output are represented respectively by get and set messages sent to $\langle\langle IO\rangle\rangle$ objects. Other messages can represent invocations of methods from a library (e.g. mult) or user-defined methods (e.g. calc and dec), which represent parts of tasks behavior.

The initial graph (initial state of the grammar) that represents the UML specification is illustrated in Figure 9. The construction of this graph is straightforward. All objects, all called methods and all communications between threads are represented by vertices. The arguments and results of method calls are also represented by vertices. The mapping of threads to processors is captured from the deployment diagram and represented by in edges in the corresponding graph. The direction of communications between threads is represented by

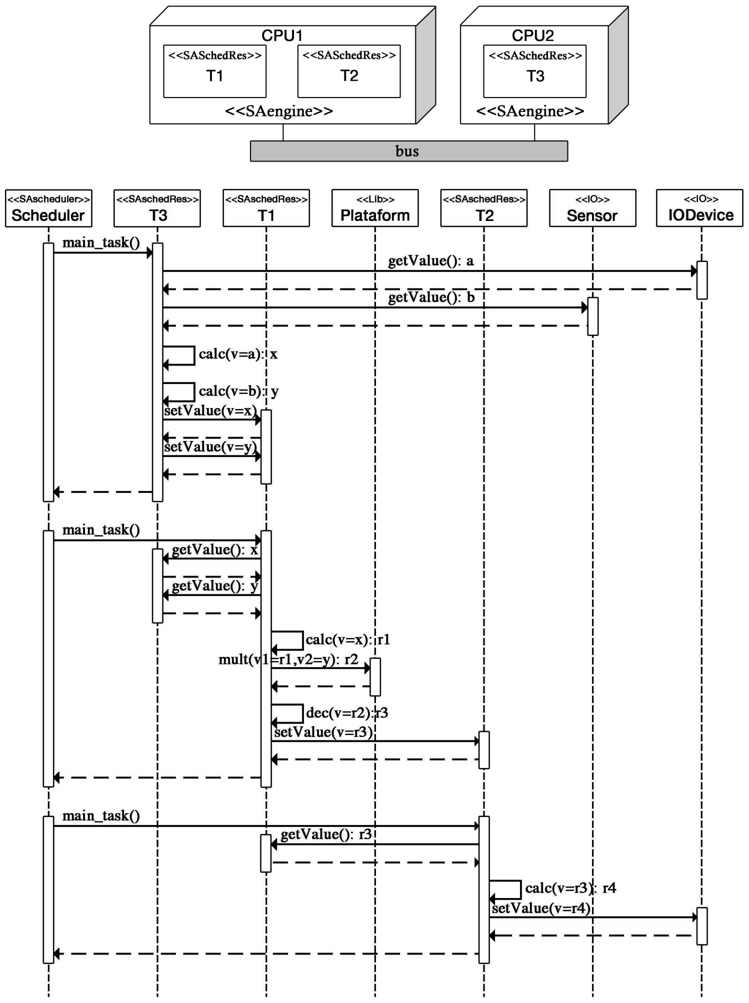


Figure 7. Deployment and Sequence Diagrams

source (src) and target (tgt) edges, and the result (in case of a get operation) or the argument (in case of a set operation) determine the respective creation of result and arg edges, connecting the method to the respective variable. The arguments and results of the remaining function calls are also represented by arg and result edges, respectively, connecting the functions to the corresponding variables. At last, an in edge from a function to a thread indicates

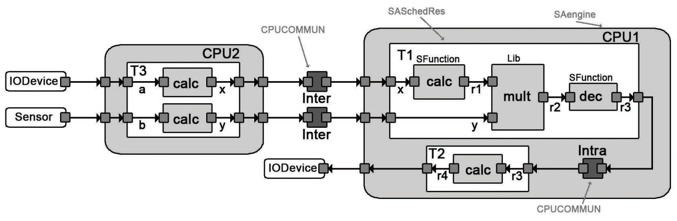


Figure 8. Generated Simulink CAAM model

which thread invokes the respective method.

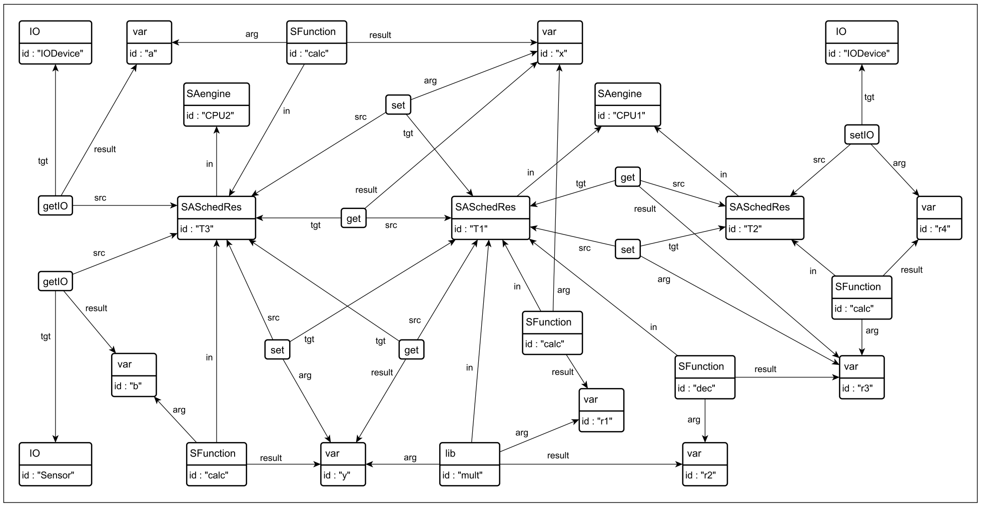


Figure 9. Initial graph.

Applying the specified rules in the initial graph (Figure 9), we obtain a graph corresponding to a Simulink diagram (Figure 16). These rules are applied to the graph until no more rules can be applied, and, thus, we get a graph that describes the Simulink diagram equivalent to the original UML diagrams.

5.2 Rules

The rules, which define the translation, can be sorted in three groups: rules IntraC, InterC, getIO and setIO with priority 1 (Figure 10); rules prefixed by Fun, Lib, or Thr with priority 2 (Figure 13); and rules prefixed by arg or res with priority 3 (Figure 15).

Rules are specified with a priority order. This feature, though not very common in graph grammar specifications, is available in the AGG system [14]. Rule priorities provide a way to schedule the application of rules: as long as a low-priority rule is enabled, no higher-priority rules can be scheduled for application. In general, this strategy simplifies rules specification, avoiding the creation of extra components (flags) necessary to enforce the order of rule applications.

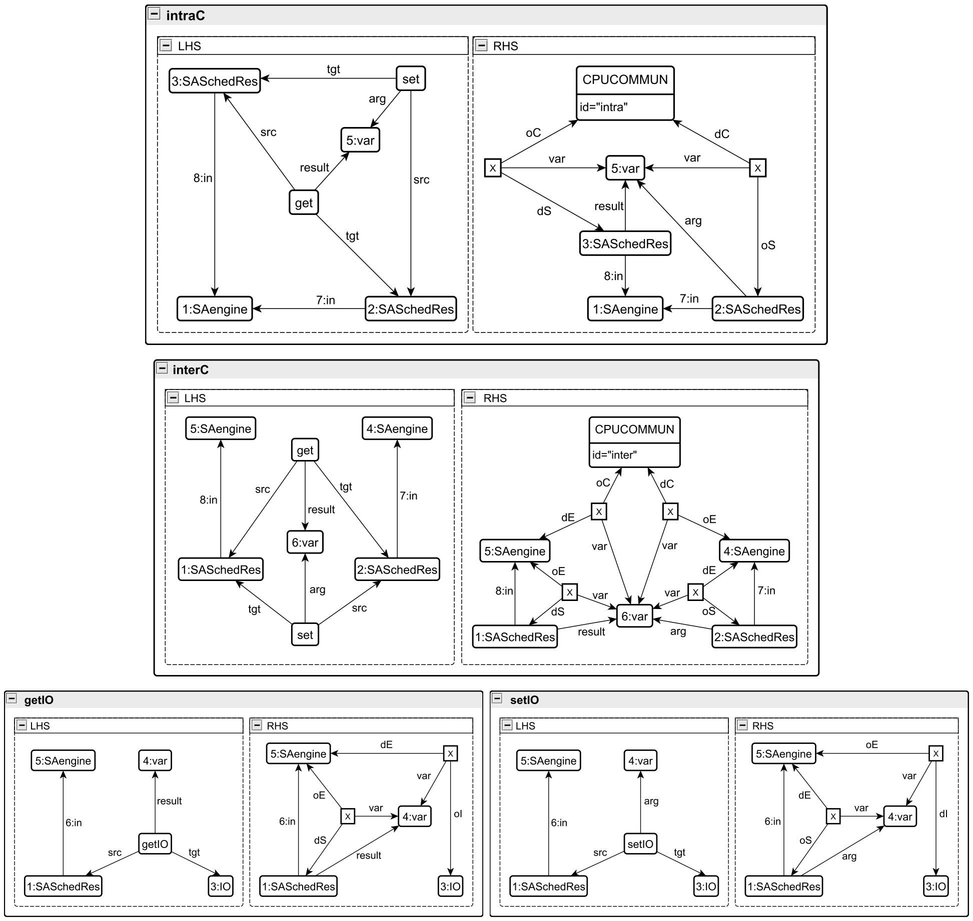


Figure 10. Rules for gets and sets with Priority 1

Rule IntraC depicted in Figure 10 identifies send and receive operations between two threads that are in the same CPU. That is, one of the threads invokes a set method from the other thread, whose argument is a variable var, while the second thread invokes a get method from the first one, whose result is stored in the same variable. In this case, an Intra CPU-COMMUN subsystem is instantiated for the corresponding variable and connections from the thread responsible for the send operation to CPUCOMMUN and from CPUCOMMUN to the receive thread are created. Rule InterC illustrated in Figure 10 is analogous, but now recognizing send and receiving operations between threads that are in different CPUs. Now, an Inter CPUCOMMUN subsystem is instantiated, besides the connections from one thread to the other. When a thread invokes a get or a set method from an IO component, rules getIO or setIO, respectively, must be applied (see Figure 10). The application of getIO creates connections from the IO component to the thread, while setIO application establishes connections from the thread to the IO subsystem. After the application of any of these rules, the get result remains available for the thread which invoked the get method (represented by creation of edge result in rules) and the set argument remains available as argument of the thread which invoked the set method (represented by creation of edge arg in rules). Figure 11 emphasizes the components of the Simulink model created from the UML diagrams in Figure 7 after the application of the IntraC rule (links connecting T1, Intra, and T2), and Figure 12 highlights the components created after application of the setIO rule (links connecting T2 output to the IODevice).

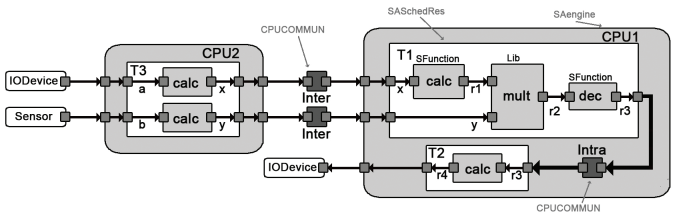


Figure 11. Simulink components created by IntraC rule (links connecting T1, Intra, and T2)

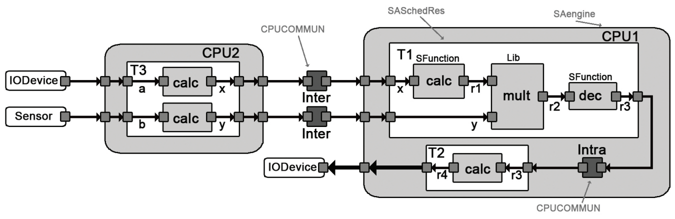


Figure 12. Simulink components created by setIO rule (links connecting T2 to IODevice)

Rules detailed in Figure 13 create connections between blocks of functions (SFunction and lib) and other functions or threads. These rules are applied when the result of a function (or thread) is used as argument of another function (or thread). Rules prefixed by Fun create connections between the output of blocks of type SFunction and the input of other blocks. Rules prefixed by Lib create connections between the output of blocks of type lib and the input of other blocks. Finally, rules prefixed by Thr create connections between the output of blocks of type SASchedRes (thread) and functions. Simulink components created by this class of rules from the UML diagrams in Figure 7 are highlighted in Figure 14 (see links inside threads).

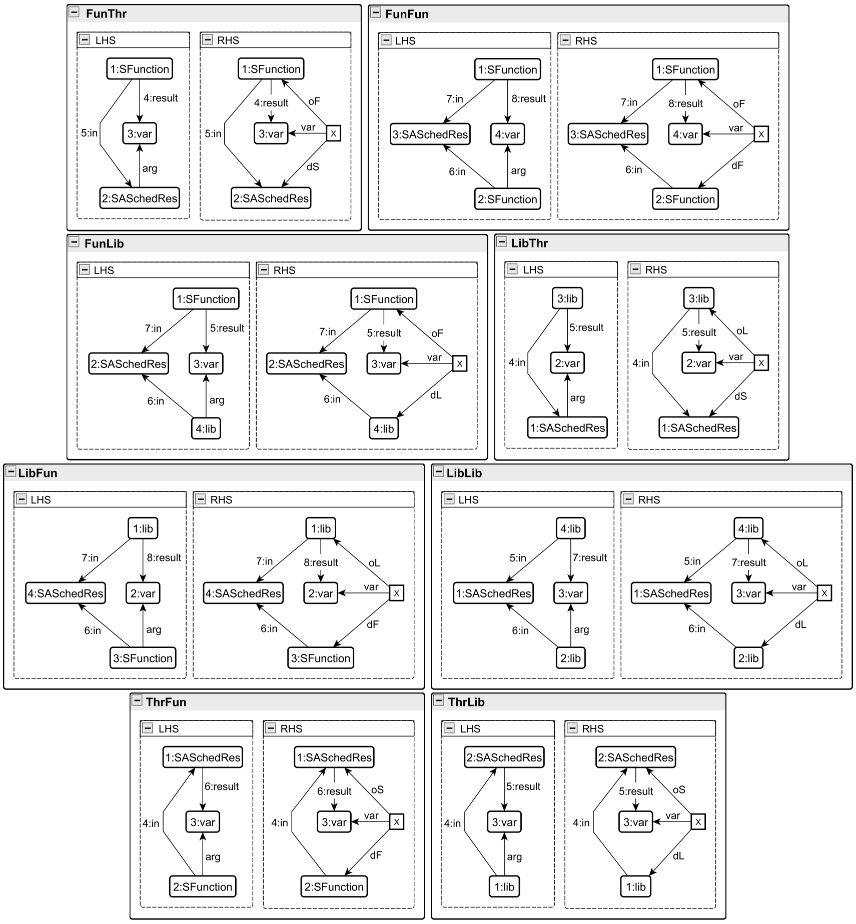


Figure 13. Rules with Priority 2

Remaining rules, illustrated in Figure 15, just execute garbage collection, deleting result and arg edges. For instance, if a result is used as an argument of more than one function,

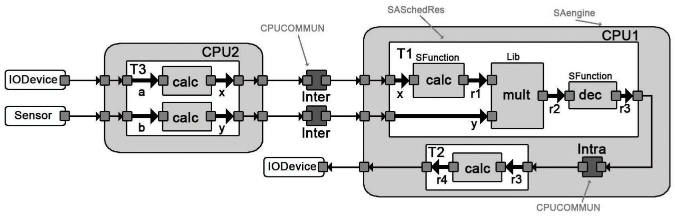


Figure 14. Simulink components created by rules with priority 2 (links inside threads)

it is not possible to delete this edge until all connections have been created. Because of that, these rules have the highest priority.

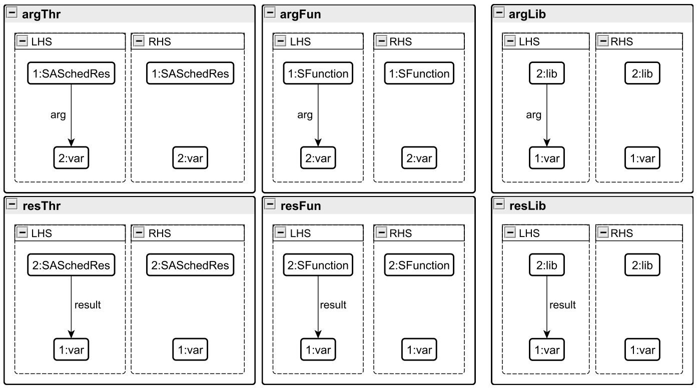


Figure 15. Rules with Priority 3

5.3 Final graph

The final graph obtained from the UML diagrams in Figure 7 (automatically through AGG) after all rule applications is depicted in Figure 16. It is important to observe that, for each UML graph, just one final Simulink graph is obtained (see discussion in Section 6).

The translation of this graph to a Simulink block diagram is straightforward. Vertices IO, SAengine, CPUCOMMUN, SASchedRes, SFunction and lib represent the corresponding Simulink blocks with the same identifier in Figure 8. The hierarchy of blocks is captured by in edges. The connections and data flow between blocks are represented by vertices X together with their connections (edges with source in X in the graph). Edges with prefix o indicate the source of the connection, and edges prefixed with d indicate the destination (target) of

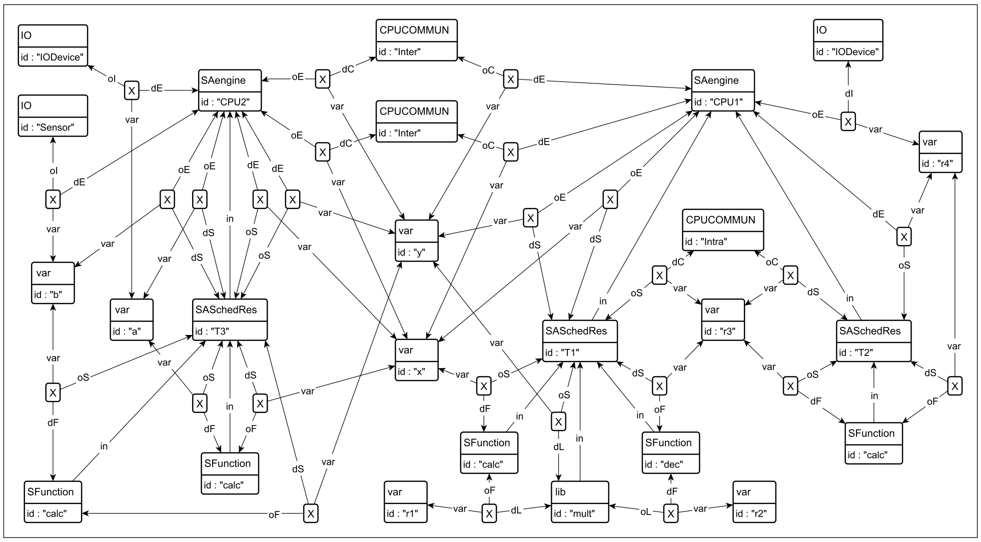


Figure 16. Graph representing a Simulink CAAM model

the connection. Edges labeled with var determine the variable that is being transmitted. For example, var with identifier a is being transmitted from the IODevice to CPU2 and from CPU2 to thread T3.

Finally, it is important to notice that the use of graph grammars to specify the translation, besides allowing the automatic translation from a UML graph to a Simulink graph, also provides a precise and formal definition for the mapping.

6 Transformation Analysis

Besides the automatic translation and precise definition of the mapping, the use of a formal language also allows the verification of some properties about the translation.

Important features that are required, when graph grammars are used to specify model transformations, are termination and confluence. Only under these conditions the existence and uniqueness of the outcoming model may be guaranteed. Now we discuss why our graph grammar specification for the mapping from UML to Simulink models satisfies such requirements. We used the AGG tool [15, 14] to proceed with the termination and confluence analysis.

6.1 Termination

The transformation process finishes when there is no rule that can be applied in the current state. For instance, a transformation does not finish when there is a rule that creates new components and is always active (that is, it can always be applied in the state graph). Another situation is when rules can be applied in a cycle, that is, components are created by rules that are deleted by others, in a situation where the application of rules that consume the items activate rules that create them. In the UML to Simulink mapping we have the following characteristics. The initial state graph is always finite. Rules with priority 1 only delete components of the initial graph. And rules with priority 2 delete arg edges (created by rules with priority 1), which will not be created again. This is because rules with priority 1 require get and set components to be applied. Such components are deleted in the first application of these rules and are not created by any other rule. Then, for each UML graph, one final Simulink graph is always obtained.

Termination was also guaranteed by the AGG tool. In AGG termination criteria are implemented for Layered Graph Transformation Systems (LGTS) with injective rules, injective matches and injective negative application conditions. A graph grammar with productions P and graph components (graph vertices and graph edges, named types) T is called *layered graph grammar* if for each rule $r \in P$ we have a rule layer $rl(r) = k$ with $0 \leq k \leq k_0$ ($k, k_0 \in \mathbb{N}$) where k_0 is the number of layers. Moreover, for each graph vertex or graph edge $t \in T$ we have a creation and a deletion layer $cl(t),\ dl(t) \in \mathbb{N}$ and each layer k is either a deletion layer or a nondeletion layer satisfying the following conditions for all $r \in P$ with $rl(r) = k$:

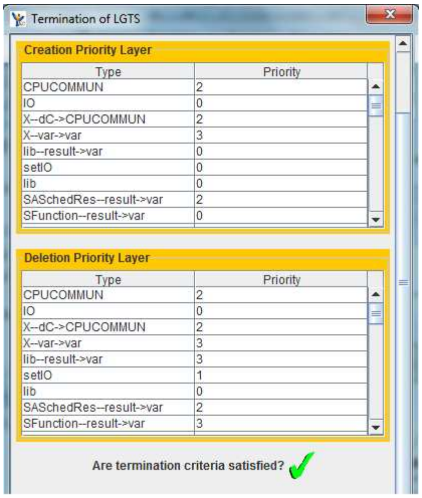
If k is a deletion layer **Deletion Layer Conditions**	If k is a nondeletion layer **Nondeletion Layer Conditions**
1. r is deleting at least one item	1. r is nondeleting, i.e. $r : L \to R$ is total and injective
2. $0 \leq cl(t) \leq dl(t) \leq k_0$ for all $t \in T$	2. r has NAC $n : L \to N$ and there is an injective $n' : N \to R$ with $n' \circ n = r$
3. r deletes $t \Rightarrow dl(t) \leq rl(t)$	3. $t \in L$ is a graph component $\Rightarrow$ $cl(t) \leq rl(r)$
4. r creates $t \Rightarrow cl(t) > rl(r)$	4. r creates $t \Rightarrow cl(t) > rl(r)$

If we have an initial graph G_0, for each graph component $t \in T$ the creation and deletion layers are assigned as follows:

$$\begin{aligned} \text{cl(t)} &= \text{if } t \in G_0 \text{ then } 0 \text{ else } max\{rl(r) | r\ creates\ l\} + 1 \\ \text{dl(t)} &= \text{if } t\ is\ deleted\ by\ some\ r \text{ then } min\{rl(r) | r\ deletes\ l\} \text{ else } k_0 \end{aligned}$$

The *deletion layer conditions* ensure that the last creation of a graph component of a certain type should precede the first deletion of a component with the same type. On the other hand, *nondeletion layer conditions* ensure that if a component of a certain type occurs in the LHS of a rule then all elements of the same type were already created in the same or a previous layer. Each layered graph grammar with injective matches terminates (for details see [18]).

In AGG, the rule layer can be set or generated. In case of the graph grammar defined in Section 5, the rule layer of each rule is determined by its priority. The creation and deletion type layer is generated automatically by AGG. And, for each layer, one set of layer conditions is proved. In fact, we do not have any layer of nondeleting rules and all three layers of our specification satisfy the deletion layer conditions. Figure 17 shows the result of the analysis of AGG for our graph grammar. As illustrated in Figure 17, AGG assigns the creation and deletion layer for each graph component and checks if the termination criteria are satisfied.


Termination of LGTS

Creation Priority Layer

Type	Priority
CPUCOMMUN	2
IO	0
X--dC->CPUCOMMUN	2
X--var->var	3
lib--result->var	0
setIO	0
lib	0
SASchedRes--result->var	2
SFunction--result->var	0

Deletion Priority Layer

Type	Priority
CPUCOMMUN	2
IO	0
X--dC->CPUCOMMUN	2
X--var->var	3
lib--result->var	3
setIO	1
lib	0
SASchedRes--result->var	2
SFunction--result->var	3

Are termination criteria satisfied? ✓

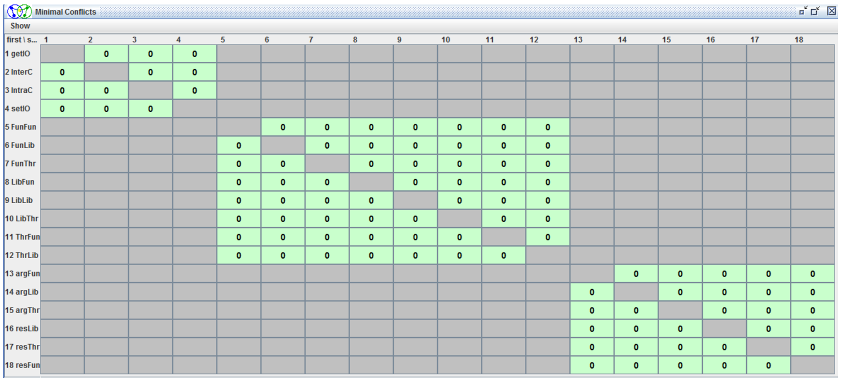
Figure 17. AGG Termination Analysis

6.2 Confluence

A model transformation is confluent if for each source model the process of transformation results in a unique target model. Critical pair analysis [14] is generally used to check if a transformation is confluent. A critical pair is a pair of transformations both starting at a common graph G such that both transformations are in conflict, and graph G is minimal according to the rules applied (that is, G only contains elements that are in the image of the matches of both rules). There exists a critical pair like above if, and only if, one rule may disable the other one. There are three reasons why rule applications can be conflicting: (i) one rule application deletes a graph component which is in the match of another rule application; (ii) one rule application generates graph components in a way that a graph structure would occur which is prohibited by a NAC of another rule application; (iii) one rule application changes attributes being in the match of another rule application. A graph grammar system is confluent if it is locally confluent and terminates. A system is locally confluent if all critical pairs are confluent, that is, all critical pairs can be derived by a sequence of transformations that leads them to a common successor graph.

We have also used the AGG tool [15] to proceed with the critical pair analysis. After computation, the set of critical pairs represents precisely all potential conflicts in the grammar. In order to detect all potential conflicts of type (i) or (iii) described above, for each pair of rules $p1 : L1 \rightarrow R1$ and $p2 : L2 \rightarrow R2$, AGG computes graph G by overlapping $L1$ and $L2$ in all possible ways, such that the intersection of $L1$ and $L2$ contains at least one item that is deleted or changed by one of the rules and both rules are applicable to G at their respective occurrences. Potential conflicts of type (ii) are found by gluing the right-hand side of the first rule and the left-hand side together with NAC elements of the second rule.

In fact, we have no critical pairs in our graph grammar. Figure 18 shows the absence of potential conflicts between each pair of rules computed by AGG. It is possible to observe that the potential conflicts are generated just for pairs of rules with the same priority. The reason for this is that we can never apply rules with different priorities to the same graph. Rules are not in conflict because they may always be applied to disjoint portions of the state graph. Then, for each UML graph, just one final Simulink graph is obtained.



first \ s...	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1 getIO		0	0	0														
2 InterC	0		0	0														
3 IntraC	0	0		0														
4 setIO	0	0	0															
5 FunFun						0	0	0	0	0	0	0						
6 FunLib					0		0	0	0	0	0	0						
7 FunThr					0	0		0	0	0	0	0						
8 LibFun					0	0	0		0	0	0	0						
9 LibLib					0	0	0	0		0	0	0						
10 LibThr					0	0	0	0	0		0	0						
11 ThrFun					0	0	0	0	0	0		0						
12 ThrLib					0	0	0	0	0	0	0							
13 argFun														0	0	0	0	0
14 argLib													0		0	0	0	0
15 argThr													0	0		0	0	0
16 resLib													0	0	0		0	0
17 resThr													0	0	0	0		0
18 resFun													0	0	0	0	0	

Figure 18. AGG Minimal Conflicts

7 Concluding Remarks

In this paper we formally define a mapping from UML to Simulink models. Our translation considers only Simulink discrete-time blocks and generates an extension of Simulink (named CAAM), which combines algorithmic and architectural aspects into a single model. We use graph grammars as specification language, which allowed the use of the AGG tool set to automate the translation. This approach allows designers to employ UML to model the whole system and reuse this specification to generate a Simulink graph representation. The generated graph is easily converted to a Simulink CAAM model that can be used as input for a Simulink-based MPSoC design flow, which generates hardware and software for an MPSoC platform. The use of graph grammars also allowed the verification of some properties about the translation. Particularly, existence and uniqueness of the outcoming model can be assured.

In order to have a fully automatic design flow from UML to Simulink, we still have to automate the transformation from UML diagrams to their corresponding graph representation and the transformation from the graph representation of Simulink to the corresponding Simulink block diagram. We also intend to explore the verification of other types of properties, like consistency in connections of the generated Simulink graph. That is, it could be important to assure that all arguments of method calls are previously produced and all results are used.

Acknowledgement

The authors gratefully acknowledge financial support received from CNPq, FAPERGS (ARD-10/0348-8, ARD-11/0764-9) and NESS project (PRONEX-10/0043-0).

References

[1] OMG, "Omg unified modeling language infrastructure version 2.4," tech. rep., 2011.

[2] L. B. Brisolara, M. E. Kreutz, and L. Carro, "UML as front-end language for embedded systems design," in *Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation* (L. Gomes and J. M. Fernandes, eds.), ch. 1, pp. 1–23, Hershey, PA: Information Science Reference - Imprint of: IGI Publishing, 2009.

[3] J. Vidal, F. de Lamotte, G. Gogniat, P. Soulard, and J.-P. Diguet, "A co-design approach for embedded system modeling and code generation with UML and MARTE,"

in *DATE'09*, (Belgium), pp. 226–231, European Design and Automation Association, 2009.

[4] G. Martin, "UML for embedded systems specification and design: Motivation and overview," in *Proceedings of the conference on Design, automation and test in Europe*, DATE'02, (Washington, DC, USA), pp. 773–, IEEE Computer Society, 2002.

[5] Mathworks, "Simulink." http://www.mathworks.com/. Last access: December, 2011.

[6] K. Huang, S.-i. Han, K. Popovici, L. Brisolara, X. Guerin, L. Li, X. Yan, S.-l. Chae, L. Carro, and A. A. Jerraya, "Simulink-based MPSoC design flow: case study of Motion-JPEG and h.264," in *DAC'07*, (NY, USA), pp. 39–42, ACM, 2007.

[7] Y. Vanderperren and W. Dehaene, "From UML/SysML to Matlab/Simulink: current state and future perspectives," in *Proc. of the conference on Design, automation and test in Europe*, DATE'06, (Belgium), pp. 93–93, European Design and Automation Association, 2006.

[8] L. B. Brisolara, M. F. S. Oliveira, R. Redin, L. C. Lamb, L. Carro, and F. Wagner, "Using UML as front-end for heterogeneous software code generation strategies," in *Proc. of the conference on Design, automation and test in Europe*, DATE'08, (NY, USA), pp. 504–509, ACM, 2008.

[9] C.-J. Sjöstedt, J. Shi, M. Törngren, D. Servat, D. Chen, V. Ahlsten, and H. Lönn, "Mapping Simulink to UML in the Design of Embedded Systems: Investigating Scenarios and Structural and Behavioral Mapping," in *OMER4 Post-proceedings*, 2008.

[10] T. Farkas, C. Neumann, and A. Hinnerichs, "An integrative approach for embedded software design with UML and Simulink," in *Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference - Volume 02*, COMPSAC'09, (Washington, DC, USA), pp. 516–521, IEEE Computer Society, 2009.

[11] G. Rozenberg, ed., *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. NJ, USA: World Sci. Pub. Co., 1997.

[12] N. N. Bisi, V. Pazzini, L. Foss, S. A. C. Cavalheiro, and L. B. d. Brisolara, "Utilizando gramática de grafos para o desenvolvimento de sistemas embarcados baseado em modelos UML," in *WEIT 2011 - I Workshop-Escola de Informática Teórica - Anais*, pp. 242–253, 2011.

[13] L. Foss, S. Costa, N. Bisi, L. Brisolara, and F. Wagner, "From UML to Simulink: a Graph Grammar Specification," in *14th Brazilian Symposium on Formal Methods: Short Papers*, pp. 37–42, 2011.

[14] C. Ermel, M. Rudolf, and G. Taentzer, “The agg approach: language and environment,” *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools*, pp. 551–603, 1999.

[15] “Agg: The homebase.” http://user.cs.tu-berlin.de/∼gragra/agg/. Last access: November, 2011.

[16] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[17] S.-I. Han, S.-I. Chae, L. Brisolara, L. Carro, K. Popovici, X. Guerin, A. A. Jerraya, K. Huang, L. Li, and X. Yan, “Simulink®-based heterogeneous multiprocessor soc design flow for mixed hardware/software refinement and simulation,” *Integr. VLSI J.*, vol. 42, pp. 227–245, February 2009.

[18] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay, “Termination criteria for model transformation,” in *Proc. of Internation Conference on Fundamental Approaches to Software Engineering, FASE’05* (M. Cerioli, ed.), vol. 3442 of *LNCS*, (Edinburgh, UK), pp. 49–63, Springer, April 2005.