Pontus Boström | Richard Grönblom | Tatu Huotari | Jonatan Wiik

An approach to contract-based verification of Simulink models

Turku Centre *for* Computer Science

TUCS Technical Report
No 985, October 2010

An approach to contract-based verification of Simulink models

Pontus Boström
Department of Information Technologies,
Åbo Akademi University,
Joukahaisenkatu 3-5, 20520 Turku, Finland
`pontus.bostrom@abo.fi`

Richard Grönblom
Department of Information Technologies,
Åbo Akademi University,
Joukahaisenkatu 3-5, 20520 Turku, Finland
`richard.gronblom@abo.fi`

Tatu Huotari
Department of Information Technologies,
Åbo Akademi University,
Joukahaisenkatu 3-5, 20520 Turku, Finland
`tatu.huotari@abo.fi`

Jonatan Wiik
Department of Information Technologies,
Åbo Akademi University,
Joukahaisenkatu 3-5, 20520 Turku, Finland
`jonatan.wiik@abo.fi`

TUCS Technical Report
No 985, October 2010

Abstract

This paper presents an approach to compositional contract-based verification of Simulink models, together with a tool that supports the approach. First, a format for contracts is presented together with a method to verify models with respect to these contracts. The verification approach uses Synchronous Data Flow (SDF) graphs as an intermediate step to obtain sequential program statements that can then be analysed using traditional refinement-based verification techniques. This gives a convenient approach to calculate the needed proof obligations using well established methods. Secondly, a tool for automatic generation of the proof obligations needed for verification is presented. This tool shows that the approach can be implemented and enables application of the method on practical problems.

Keywords: Contract-based design, Refinement, Synchronous Data Flow, Formal verification, Automatic verification, SMT solving

TUCS Laboratory
Distributed systems design laboratory

1 Introduction

Model-based design has become a widely used design method to create embedded control software. In this approach, the controller is developed together with a simulation model of the plant to be controlled. This enables simulation of the complete system and thereby some degree of evaluation and testing of the controller without using a prototype. One of the most popular tools for model-based design of control systems is Simulink [28].

Simulink has a user-friendly graphical modelling notation based on data flow diagrams, as well as good simulation tools for testing and validating controllers together with models of the controlled plant. The complexity of control systems is increasing rapidly as more functionality in many applications, such as anti-locking brakes and fuel-injection systems, is implemented in software. As the systems become more complex, the size of the Simulink models used in their design also quickly grows. This leads to the same problems with complexity as software development in any modelling or programming notation. Hence, there is a need to better manage the complexity of models. Since control systems also often have high reliability requirements, there is also a need analyse the models for correctness. One approach that we have explored to remedy the problems above is to use contracts to aid the decomposition of models into smaller parts with well defined interfaces and to analyse those parts and their interaction for correctness.

The aim of this paper is to propose a compositional verification technique for Simulink models based on contracts. Contracts here refer to pre- and postconditions for programs or program fragments. Contract-based design has become a popular method for object-oriented software development [29, 14, 9]. There the main benefit of contracts is that the interfaces and the responsibilities of the objects are clearly stated, which enables analysis of the correctness of the system. However, formal verification of object-oriented systems is difficult [8] due, e.g., to aliasing of object references and re-entrant methods. However, the philosophy behind contracts seems to be a useful concept to tackle complexity, which suggests that the same ideas could also be useful in Simulink. Furthermore, Simulink is language based on data flow diagrams, where the interaction between components is much simpler than the interaction between objects in object-oriented languages. This means that formal verification can potentially be easier to do.

We have earlier developed and analysed contracts for Simulink data flow diagrams [12, 13, 10]. However, here we give more expressive contracts and a concise method to calculate the needed proof obligations for correctness. The formal analysis methods for complete Simulink models with contracts are based on interpreting the models as action systems [1, 2, 4]. Using this approach, the well known reasoning techniques used for analysing action systems, which are based on the refinement calculus can be reused. To obtain the sequential program statements needed for analysis, Simulink diagrams are viewed as synchronous data flow (SDF) graphs [25, 24]. The benefit of using SDF graphs is that the mapping of the data flow graphs to the equivalent sequential programs used in the analysis is well investigated. The focus of this paper is on theoretical aspects and tool support. However, in [12, 10] the contracts are applied to the design of a controller for a digital hydraulics system. The contracts were shown to be useful both for structuring the system and for verifying properties of the controller.

The paper starts with an overview of Simulink with an example, as well as the proposed contract format. Then SDF graphs are presented with the translation procedure to the sequential programming notation used for analysis. This is followed by a presentation of how SDF graphs can be interpreted as action systems and how this relates to correctness. Representation of Simulink diagrams as SDF graphs is then discussed, followed by a presentation of methods for analysis of correctness with respect to contracts. To illustrate the approach, contract-based verification of an example consisting of a PID-controller is demonstrated. Finally, the tool support for the verification approach is presented.

2 Related work

Other formalisations of Simulink, as well as methods to verify Simulink models exist. In [38], a translation of discrete multi-rate Simulink models to Lustre [15] is presented. The translated models can then be analysed using the tools available for Lustre. A translation to Circus [41] can be found [16]. The focus is there on development of correct code with the Simulink diagrams as the specification. The formalisation of Simulink in *Timed Interval Calculus* (TIC) given in [17] is a very flexible approach that takes into account both continuous and discrete models. However, the proofs in that approach seem to difficult to automate, since set-theory is heavily used in the formalisation. Furthermore, Simulink also comes with its own formal verification tool, *Simulink Design Verifier* [28]. This tool can be used to prove properties about models, which are stated in special assertion blocks. The focus of the above techniques are on property verification of existing models and they do not systematically consider compositional verification based on contracts. However, correctness with respect contracts could be analysed in those frameworks also. However, our approach to verification gives a convenient approach to separately reason about both pre- and post-conditions, as well as refinement. We can also easily handle many of the imperative constructs of Matlab often used in Simulink. Furthermore, proof obligation generation is straightforward and the proof obligations are in first-order logic, which means that they possibly can be discharged by automatic SMT or constraint solvers. Contracts have also been developed for synchronous languages in [27]. Those contracts are very similar to ours. However, here we take a refinement and proof based approach to compositional verification of correctness. They only provide a correctness definition based on traces for individual components.

3 Simulink

Simulink is a graphical language based on hierarchical data flow diagrams. A Simulink diagram consists of functional blocks connected by signals (wires). The blocks represent transformations of data, while the signals give the flow of data between blocks. The blocks have in- and out-ports that act as connection points for signals. The in-ports provide data to the blocks, while the out-ports provide the results computed by the blocks. Blocks are often parameterised, to make them more flexible. However, the block parameters are set before the execution of a diagram, and remain constant during the execution. Some types of blocks also contain memory. Hence, the values computed by these blocks do not only depend on the current values on the in-ports and parameter values, but also on previous in-port values.

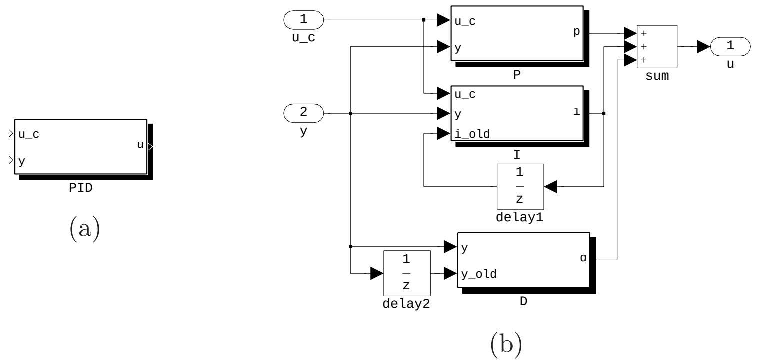


Figure 1: (a) A subsystem that implements a PID-controller and (b) its content

Here only discrete Simulink models with one sampling time are considered. This means that a model is evaluated periodically with a given sampling frequency. At each sampling instant, all blocks in the diagram are evaluated in the order given by the signals between them. The models are also assumed to be non-terminating, which is a common assumption for control systems.

In its most general form, a discrete Simulink block b contains a list of in-ports u, a list of out-ports y, parameters c and a state vector (internal memory) x [28]. The behaviour of the block is given by a difference equation of the form shown in (1).

$$\begin{aligned} &y.k = f.c.(x.k).(u.k) \\ &x.(k+1) = g.c.(x.k).(u.k) \end{aligned} \tag{1}$$

Here f denotes the function that gives the value of the out-ports y at sample k and g the function that updates the state x. Consider, e.g., the *Sum*-block (in-ports marked by +-signs) and the *Unit Delay*-blocks (marked by $1/z$) in Figure 1 (b). A *Sum*-block sums the inputs and a *Unit Delay*-block delays the input with one sampling time. Assume the Sum-block has out-port y and in-ports u_1, u_2, u_3. Its behaviour is then given by the equation $y.k = u_1.k + u_2.k + u_3.k$. Note that the *Sum*-block has no internal state. A *Unit Delay*-block on the other hand contains state to remember previous value of the input. Its behaviour is given as $y.k = x.k \wedge x.(k+1) = u.k$. Information on the behaviour of other blocks can be found in the Simulink documentation [28].

To illustrate the use of Simulink, a small example that consists of a PID-controller is presented. The PID (proportional, integral, and derivative)-controller is a standard type of controller used in many applications. The input to the controller is a lead-value and a sensor input from the system. The lead value u_c gives the value we like a measured quantity y of the controlled system to have. The output from the controller is a control signal u. In its discrete form the PID-controller can be given by the following system of difference equations [40]:

$$\begin{aligned} &u.k = p.k + i.k + d.k \\ &p.k = K(u_c.k - y.k) \\ &i.k = i.(k-1) + KT_s/T_i(u_c.k - y.k) \\ &d.k = KT_dN/(T_d + NT_s)(y.k - y.(k-1)) \end{aligned} \tag{2}$$

The controller is here sampled with a fixed sampling time T_s. Note that in the derivative term d we do not derive the lead-value u_c. This is recommended to avoid spikes in the control signal and thereby achieve better control quality.

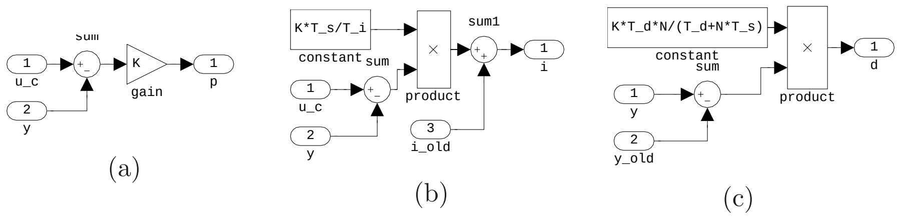


Figure 2: The content of subsystems (a) P, (b) I and (c) D from the diagram in Figure 1 (b)

The constant N is a number between 3 and 30 used to make the derivation part less active at high frequencies with the purpose to improve control quality. The PID-controller given here is a very simple variant, more features to improve control quality and flexibility of the controller can be added [40].

The subsystem block in Figure 1 (a) contains an implementation of the difference equation in (2). The Simulink diagram in Figure 1 (b) shows the content of that subsystem. The boxes with rounded corners are in- and out-blocks used to communicate with Simulink diagrams at higher levels in the subsystem hierarchy. The in- and out-blocks correspond to the in- and out-ports of the subsystem they are in. There are three subsystems inside the PID subsystem, P, I and D. All subsystems here are *atomic*, meaning that they are executed as atomic units. This is in contrast to *virtual subsystems* that are only used to syntactically group functional blocks together. A virtual subsystem is not necessarily executed as one unit, but execution of blocks from several virtual subsystems can be interleaved. These subsystems do not therefore have any influence on the behaviour of the diagram.

In Figure 1 (b), the subsystem P calculates the proportional part p of the control signal. The subsystem I calculates the integral part i of the control signal based on the value of the integral at the last sampling instant. This value is obtained from a *Unit Delay block* that delays i for one sampling instant. The subsystem D then calculates the derivative part d of the control signal. This is done using the current and previous value of the sensor value y. Finally, all three components p, i and d are summed together to give the control signal u. The content of the subsystem P is shown in Figure 2 (a), the content of I is shown in Figure 2 (b) and the content of D is shown in 2 (c).

4 Contracts in Simulink

Simulink diagrams for advanced control systems can contain thousands of blocks. For example, in the system discussed in [12, 26] from the area of digital hydraulics, the controller contains more than 4000 blocks and the subsystem hierarchy is more than 10 layers deep at the maximum. At this size, the complexity is starting to become a serious problem. To manage the complexity, there is a need to better make explicit the division of responsibility between subsystems and to analyse that all corner cases have been considered. It is also useful to reason about the interaction between subsystems at a higher level of abstraction than their detailed content, which often consists of a deep hierarchy of diagrams containing hundreds of blocks. Our proposed solution to the problems above is to use contracts to describe subsystems. This enables reasoning about subsystem interaction on the level of contracts, as well as reasoning about correctness of subsystems with respect to contracts. The contracts are intended for expressing properties of control logic and implementation level properties (e.g. bounds on variables). System level properties such as e.g. stability and performance are best expressed and verified by other means.

An atomic subsystem can essentially be considered to describe a block of the form in (1), where the internal diagram implements f and g and the state x is provided by the memories of the blocks inside the subsystem. A contract contains conditions to describe this type of functionality. Our proposed contracts have the following form:

$$\begin{array}{l} \text{parameters} : c : \mathit{type} \\ \text{inports} : u : \mathit{type} \\ \text{outports} : y : \mathit{type} \\ \text{memory} : x : \mathit{type} \\ \text{paramcond} : Q^{param} \\ \text{precondition} : Q^{pre} \\ \text{postcondition} : Q^{post} \\ \text{initcondition} : Q^{init} \\ \text{postconditionm} : Q^{postm} \\ \text{refrel} : Q^{refrel} \end{array} \tag{3}$$

The contract first declares the parameters, in- and out-ports of the subsystem. The internal state of the subsystem is modelled by memory variables x (also referred to as specification variables), the predicate Q^{param} describes the block parameters used in the subsystem, Q^{init} describes the initial values of x, Q^{pre} is the pre-condition, Q^{post} is the post-condition constricting the out-ports and Q^{postm} the post-condition constricting the new values of the specification variables. The use of specification variables in the contracts corresponds to the use of memory in normal Simulink blocks. The condition Q^{refrel} is then used to describe how the specification variables in the contract relate to the actual block memories inside the subsystem. The contracts here are more expressive than the ones in [12, 13, 10] as those contracts only considered input/output constraints and not internal state. However, the contracts are similar in expressiveness as the ones for synchronous components in [27].

4.1 Contract example

Consider again the PID-controller presented in Section 3. We like to describe its functionality with a contract. Since we cannot directly access old values of ports, we need specification variables to express the integral and derivative behaviour in contracts. We use i_1 to denote the previous value of the integral and y_1 to denote the previous value of the sensor input y. The subsystem also has the block parameters T_s, K, T_i, T_d and N. The parameter condition states restrictions on the parameters of the PID-controller. The main requirement is that all parameters except possibly K are positive. The requirement of N is more of a recommendation [40], but it is stated here as well. The initial values of the specification variables are not required to be zero, but here they are given as zero for simplicity. The controller does not have a precondition, since we assume that unbounded real numbers are used. The postconditions Q^{post} and

Q^{postm} encode the difference equation in (2) as a contract.

$$
\begin{array}{l}
\text{parameters :} \\
\ T_s : \mathit{double};\ K : \mathit{double};\ T_i : \mathit{double};\ T_d : \mathit{double};\ N : \mathit{double} \\
\text{inports :} \\
\ u_c : \mathit{double};\ y : \mathit{double} \\
\text{outports :} \\
\ u : \mathit{double} \\
\text{memory :} \\
\ i_1 : \mathit{double};\ y_1 : \mathit{double} \\
\text{parametercondition :} \\
\ T_s > 0 \wedge T_i > 0 \wedge T_d \geq 0 \wedge N \in [3, 30] \\
\text{precondition :} \\
\ \mathit{true} \\
\text{postcondition :} \\
\ u = K(u_c - y) + \frac{KT_s}{T_i}(u_c - y) + i_1 + \frac{KT_dN}{T_d + NT_s}(y - y_1) \\
\text{postconditionm :} \\
\ i_1' = \frac{KT_s}{T_i}(u_c - y) + i_1 \wedge y_1' = y \\
\text{initcondition :} \\
\ i_1 = 0 \wedge y_1 = 0 \\
\text{refrel :} \\
\ \mathsf{v}.\mathit{delay}2 = y_1 \wedge \mathsf{v}.\mathit{delay}1 = i_1
\end{array}
\tag{4}
$$

The refinement relation describes how the memories from the *Unit delay*-blocks *delay1* and *delay2* relate to the specification variables. The function v is explained later in Section 7.

5 Synchronous data flow graphs

We like to analyse Simulink models using the classical tools for program analysis [5, 6], which are designed for analysis of imperative programs. The reason is that they provide a mature framework for reasoning about program correctness, as well as refinement. To obtain such sequential programs from Simulink diagrams, we represent the diagrams as synchronous data flow (SDF) graphs, since compilation of such graphs to sequential or parallel code has been studied extensively. We already have necessary and sufficient conditions for when a schedule (sequential program) can be obtained from a SDF graph, as well as algorithms for scheduling [25, 24].

A data flow program is described by a directed graph where data flows between nodes along the edges. Synchronous data flow programs are a special case where the communication between nodes is synchronous, i.e., the size of the communication buffers is known in advance. The paradigm in [25, 24] is intended for heterogenous systems where the nodes can be implemented either by other data flow graphs or in some other programming notation. A node can produce a new value on its outgoing edges when data is available on all incoming edges. A node with no incoming edges can fire at any time. Nodes have to be side-effect free. The data flow graphs presented here are used for sampled signal processing systems, i.e., the nodes in the diagrams are executed periodically with a given sampling frequency. Furthermore, the SDF programs are never supposed to terminate.

We use a similar notation as in [25, 24] to describe our synchronous data flow graphs. An example is given in Figure 3. The program computes the (exponential) moving average of the input u over time, $v.k = aw.k + (1-a)D.v.k$. Here $D.v.k$ denotes the delay of v with one sampling time, $D.v.k = v.(k-1)$.

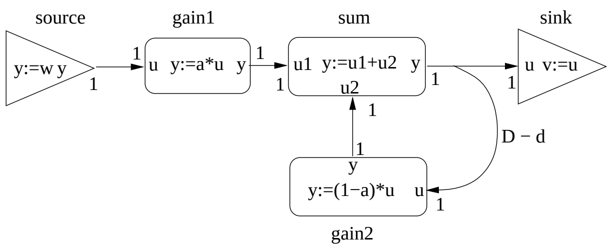


Figure 3: Example of a simple SDF program

Each node is labelled with the in- and out-port names, as well as the update statement inside the node that describes how the out-ports are modified each time the node is executed. The triangle shaped nodes are input or output nodes. They are used to model input and output of data from outside of the graph. The input blocks are assumed to always have data available [25]. The number x on an edge adjacent to the source node denotes that the node will output x pieces of data, while the number y near the destination node denotes that the block will read y pieces of data when it fires. This gives a convenient way to also handle multi-rate data flow networks. In single-rate graphs, which is our main concern here, x and y are always 1. The D on one of the edges denotes that the edge delays the data by one sampling time. Each delay also has an identifier, here d.

The nodes in the SDF graph can be statically scheduled to obtain sequential or parallel programs. In [25] necessary and sufficient conditions for this are given together with scheduling algorithms. We will only present the algorithm for obtaining a minimal *periodic admissible sequential schedule* (PASS), which represents the shortest repeating sequential program. To describe the scheduling, we first construct a *topology matrix* for the SDF graph. This matrix describes how the data availability on the edges changes during the execution of the graph. As an example, consider the graph G in Figure 3. We first number the nodes using a function n_n and edges using n_e according to:

$$\begin{array}{lcl}
n_n.source & = & 1 \\
n_n.gain1 & = & 2 \\
n_n.sum & = & 3 \\
n_n.gain2 & = & 4 \\
n_n.sink & = & 5
\end{array}
\quad \text{and} \quad
\begin{array}{lcl}
n_e.(source, gain1) & = & 1 \\
n_e.(gain1, sum) & = & 2 \\
n_e.(gain2, sum) & = & 3 \\
n_e.(sum, gain2) & = & 4 \\
n_e.(sum, sink) & = & 5
\end{array}$$

The element $(n_n.n, n_e.e)$ of the topology matrix Γ then describes how many data items node n produces on edge e when it fires. The matrix for G in Figure 3 is:

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \end{bmatrix} \qquad (5)$$

The node run at step k is specified with a vector that contains 1 in the position corresponding to the number $n_n.n$ of the node n and 0 elsewhere. For example, if the node *source* is run then $v.k$ is:

$$v.k = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$

Using the vector $v.k$ for the node executed at step k, the amount of data on the edges at step $k+1$, $b.(k+1)$, is now given as:

$$b.(k+1) = b.k + \Gamma v.k \tag{6}$$

The change to the buffers is given as the product of the topology matrix and the current $v.k$. The initial amount of data on an edge is given by the number of delays on the edge. If there is no delay the initial amount is zero, otherwise the number of data items equals the number of delays. For the graph G, the initial state is given by:

$$b.0 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix}^T \tag{7}$$

The vectors q in the null-space[1] of Γ then give the number of times the nodes can be executed in order to return the buffers to the initial state.

$$b.0 = b.0 + \Gamma q$$

The least, non-zero, integer vector in the null-space of Γ gives the number of times each node is executed in the minimal PASS. This gives an algorithm for scheduling the nodes.

1. Find the smallest integer q in the null-space of Γ
2. Construct a set S of all nodes in the graph
3. For each $\alpha \in S$, schedule α if it is runnable and then update the state $b.(k+1)$ in (6) according to v for α. A node is runnable if it has not yet been run q_α times and if execution of α does not make any buffer $b_i.(k+1)$ in (6) negative.
4. If each node α is scheduled q_α times, then stop
5. If no node in S can be scheduled, indicate a deadlock (the graph cannot be scheduled) else go to step 3.

In this paper we are only considering systems where all data-rates are one and there is at most one delay on each edge. Because of these assumptions, the algorithm can be simplified to topological sorting of the nodes in the graph according to a dependency relation. To do this, we first need to define exactly the notion of dependency between nodes:

Definition 1. *A node m depends on a node n if execution of m before n would make any buffer $b_i.(k+1)$ in (6) negative.*

The dependency in Definition 1 can be stated as the property on the graph as in Proposition 1.

Proposition 1. *A node m depends on n if there is an edge e without a delay from n to m.*

Proof. There are two cases: (1) There is an edge e from n to m and (2) there is not.

[1]The null-space of a matrix A is the set of all vectors q, such that $Aq = \mathbf{0}$

1. Because of the structure of Γ, in order for buffer $b_e.(k+1)$ in (6) to be non-negative after execution of m at step k, then $b_e.k$ has to be strictly positive. This happens if there is a delay on e and $b_e.0 = 1$ or at some point $i, i < k$, the execution of n made $b_e.i \geq 1$. Thus if e does not have a delay from n to m, we have a dependency according to Definition 1.

2. Because of the structure of Γ the buffer at $b_e.(k+1)$ will be un-affected by execution of m at step k. Thus we have no dependency according to Definition 1.

□

The nodes in the graph can now be sorted according to the dependency in Proposition 1.

Proposition 2. *A minimal PASS can be obtained for an SDF graph where all data rates are* 1 *and all edges have at most a single delay, by topologically sorting the nodes according to the dependency graph given in Proposition 1.*

Proof. When all the data rates are 1 we have that all nodes are executed once (the minimal vector q in the null-space of Γ is in this case $q = \mathbf{1}$, see Lemma 4 and Theorem 2 in [25]). The scheduling algorithm outlined above then simplifies to topological sorting according to the dependency graph in Proposition 1. □

6 Language of nodes

The computations inside nodes are described with a simple imperative programming language. This language is also the target language when translating the SDF graph to a sequential program. The focus is here on verification and the language is therefore optimised for this purpose.

Since the analysis methods are based on the refinement calculus [5], a short introduction is needed. The refinement calculus is based on Higher-Order Logic (HOL) and lattice theory. The state space of a program in the refinement calculus is assumed to be of type Σ. Predicates are functions from the state space to the type boolean, $p : \Sigma \to \mathbb{B}$. A predicate corresponds to the subset of Σ where p evaluates to *true*. Relations can be thought of as functions from elements to sets of elements, $R : \Sigma \to (\Sigma \to \mathbb{B})$. A program statement is a predicate transformer from predicates on the output state space Σ to predicates on the input state space Γ, $S : (\Sigma \to \mathbb{B}) \to (\Gamma \to \mathbb{B})$. A predicate transformer S applied to a predicate q gives the weakest predicate from where S is guaranteed to establish q. This is referred to as the weakest-precondition semantics [19, 5] of program statements. The syntax of the statement language is given as:

$$\begin{array}{lll} S ::= & x, y := E, F \mid & \text{Assignment} \\ & [g] \mid & \text{Assumption} \\ & \{g\} \mid & \text{Assertion} \\ & x, y : |P \mid & \text{Non}-\text{deterministic assignment} \\ & skip \mid & \text{Skip statement} \\ & S_1 ; S_2 \mid & \text{Sequential composition} \\ & S_1 \sqcap S_2 & \text{Non}-\text{deterministic choice} \end{array} \tag{8}$$

Here x is a variable, y a comma separated list of variables, E an expression, F a comma separated list of expressions, while g and P are predicates. For an arbitrary post-condition q we have that:

$$\begin{array}{lcl}
(x, y := E, F).q & = & (q[x, y/E, F]) \\
[g].q & = & g \Rightarrow q \\
\{g\}.q & = & g \wedge q \\
(x, y : |P(x, y, x', y').q & = & \forall x', y' \cdot P(x, y, x', y') \Rightarrow q[x, y/x', y'] \\
\mathit{skip}.q & = & q \\
(S_1; S_2).q & = & S_1.(S_2.q) \\
(S_1 \sqcap S_2).q & = & S_1.q \wedge S_2.q
\end{array} \tag{9}$$

Each statement can thus be considered to be a predicate transformer that transforms a post-condition q into the weakest precondition for the statement to establish condition q. A statement S terminates properly, if it is executed in a state where it can reach the weakest post-condition *true*. These states are described by the condition $S.\mathit{true}$, which is referred to as the termination guard of S, $\mathsf{t}.S \mathrel{\hat{=}} S.\mathit{true}$. In states where $S.\mathit{true}$ does not hold the statement is said to abort. A statement S is said to behave miraculously, if executed in a state where $S.\mathit{false}$ holds. The statement S can then establish any post-condition. The condition that describes the states where S will not behave miraculously is called the guard of S, $\mathsf{g}.S \mathrel{\hat{=}} \neg S.\mathit{false}$.

All statements in (8) are conjunctive predicate transformers [5, 7]. A predicate transformer S is conjunctive, if it satisfies the following condition for a non-empty I:

$$\bigwedge_{i \in I} S.q_i = S.(\bigwedge_{i \in I} q_i) \tag{10}$$

An important property for stepwise development is monotonicity [5]. A statement S is monotonic, if it preserves the ordering given by implication.

$$S.q \Rightarrow S.p, \quad \text{if } q \Rightarrow p \tag{11}$$

Note that conjunctivity (10) implies monotonicity (11) [5].

Refinement can be defined for the predicate transformers.

$$S \sqsubseteq R \;\hat{=}\; \forall q \cdot S.q \Rightarrow R.q \tag{12}$$

This condition states that if S can establish a postcondition q, then q can also be established by R. Since all statements are *monotonic*, refinement of an individual statement in a program leads to the refinement of the whole program [5]. We can also introduce the concept of *data refinement.* Data refinement is used when two programs do not necessarily work on the same state-space and we like to prove that one refines the other. To prove the refinement, we use a decoding statement Δ that maps the concrete state space to the abstract state space [3, 6]. Data refinement of S by R under decoding Δ, $S \sqsubseteq_\Delta R$, is defined as:

$$S \sqsubseteq_\Delta R \mathrel{\hat{=}} \Delta; S \sqsubseteq R; \Delta \tag{13}$$

The decoding Δ is normally assumed to have the form $\Delta \mathrel{\hat{=}} \{+a - c|Q\}$ [3], where $\{+a - c|Q\}$ denotes non-deterministic *angelic* assignment that removes the concrete variables c from the state space and adds the abstract variables a to the state space in manner such that Q relates a and c [3]. An angelic relational assignment statement has the semantics: $\{+a - c|Q\}.q = \exists a' \cdot Q[a/a'] \wedge q[a/a']$ (see [3, 6]).

Due to the quantification over predicates, the formulation of refinement above is not very convenient to use. We here use a condition that allows generation of proof obligations for refinement in first order logic when the abstract statement has a specific format. Using $\Delta = \{+a - c|Q\}$ and abstract statement $S = \{g\}; a, z : |P$, rule (14) can be used to prove $S \sqsubseteq_\Delta R$ [3].

$$Q \wedge g \wedge z, a = z_0, a_0 \Rightarrow R.(\exists a' \cdot Q[a/a'] \wedge P[a, a', z, z'/a_0, a', z_0, z]) \qquad (14)$$

Here a again denotes the abstract variables, c denotes the concrete variables and z common variables. The intuition is that if the precondition g holds in the abstract initial state then the concrete statement R will reach a state corresponding to an abstract state reachable by $a, z : |P$.

7 Translation of SDF graphs

An SDF graph can be translated to a sequential statement by utilising the scheduling in Proposition 2. However, first we need to introduce the communication buffers needed to handle communication between the nodes. In principle the communication between nodes is handled through FIFO-buffers [24]. However, to make the proof obligations simpler, we would like to have static buffers (shared variables). As stated earlier, we consider only a special case where all data rates are one and we only have edges with one delay. Utilising these properties, static buffering can be implemented as follows. All ports and delays are first translated as variables.

Definition 2. *Let the function* v *be a injective function from node and port or delay to variable identifier. Then* $\mathsf{v}.n.p$ *maps a node* n *and port* p *to a unique identifier, while* $\mathsf{v}.d$ *then maps a delay* d *to a unique variable identifier.*

Using the unique variable identifiers, an SDF graph can be translated to a statement in the imperative programming language in (8).

Definition 3. *Let* trans *be a function from an SDF graph to a sequential statement. The translation* $\mathsf{trans}.G$ *of SDF graph* G *is obtained as follows:*

1. *For each node* n *in* G*: Each out-port* p *in* n *is translated to a unique variable* $\mathsf{v}.n.p$*. Each unconnected in-port* p *in* n *is also translated to a unique variable* $\mathsf{v}.n.p$*.*
2. *Each delay* d *is also translated to a unique variable* $\mathsf{v}.d$*.*
3. *The sequential statements from the nodes in* G *are scheduled according to Proposition 2.*
4. *For each delay* d *on an edge* e *an update statement* $\mathsf{v}.d := \mathsf{v}.n.p$*, where* $\mathsf{v}.d$ *is the variable obtained from* d *and port* p *in* n *is the source port of* e*, is added after the statements from the source and destination nodes of* e*.*

It is easy to see that the data is handled as if FIFO-buffers were used for the special case in the paper. If there is no delay on an edge, then the required buffer size is one, since only one data element can be produced by the source node. We then have that the variable obtained from the out-port corresponds directly to a buffer with one element. In case there is one delay on an edge the required buffer size is two, since both the delayed value and the value produced by the source node have to fit into the buffer. In this case the delay variable corresponds to the head of the buffer and the variable obtained from the out-port in the source node corresponds to the tail element. Figure 4 illustrates this situation.

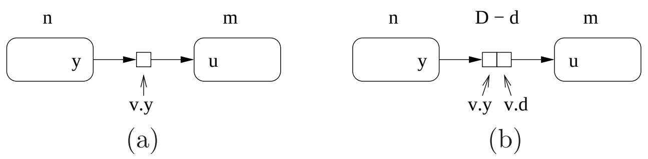


Figure 4: (a) The buffer of an edge from n to m without a delay and (b) the buffer of an edge with one delay d

Consider the SDF graph G in Figure 3. This graph is translated to the sequential program $\mathsf{trans}.G$ given below:

$$\begin{aligned}
&\mathsf{trans}.G \mathrel{\hat{=}}\\
&\quad \mathsf{v}.source.y := \mathsf{v}.G.w;\\
&\quad \mathsf{v}.gain1.y := a * \mathsf{v}.source.y;\\
&\quad \mathsf{v}.gain2.y := (1-a) * \mathsf{v}.d;\\
&\quad \mathsf{v}.sum.y := \mathsf{v}.gain1.y + \mathsf{v}.gain2.y;\\
&\quad \mathsf{v}.d := \mathsf{v}.sum.y;\\
&\quad \mathsf{v}.G.v := \mathsf{v}.sum.y
\end{aligned}$$

The statements are obtained from the nodes and then scheduled according to Definition 3. Here we assume that w and v in the in and out nodes are ports of a node G that contains the graph. Note that we have directly replaced every in-port with the out-port variable or delay variable it is connected to.

8 Action systems

To give a complete formal semantics to the SDF graphs, we view them as action systems. Action systems [1, 2, 4, 7] can be used for describing reactive and distributed systems. The formalism was invented by Back and Kurki-Suonio and inspired by Dijkstra's guarded command language [19]. The first versions used temporal logic and they were aimed at developing distributed systems [1]. Later, action systems have been analysed in the refinement calculus framework [4, 7].

An action system consists of global and local variables, an initialisation of the local variables and a *do*-loop containing actions. Action systems use refinement calculus statements for the definition of actions. An action system has the form:

$$\mathcal{A} \mathrel{\hat{=}} |[\ \mathbf{var}\ x;\ \mathbf{init}\ A_0;\ \mathbf{do}\ A\ \mathbf{od}\ \ ]| : \langle z \rangle \tag{15}$$

Here x denotes the local variables and z the global variables. The initialisation of the action system is given as a predicate A_0 that describes the initial values. The loop body consists of the demonic choice between actions constructed from conjunctive predicate transformers. All actions can therefore be written together as one single action A without loss of generality [5].

8.1 Trace semantics

In order to reason about reactive behaviour, the state of the system during execution is important. The refinement calculus cannot be used directly, since it concerns only the input-output behaviour of a program. The execution of an action system gives rise to a sequence of states, called behaviours [4, 7]. Behaviours can be finite or infinite. Here finite behaviour is always considered as something undesirable, since control systems usually are intended to run forever. Finite behaviours can be aborted or miraculous. An aborted behaviour can be considered a failure of the system and a miraculous behaviour can be considered a deadlock.

To simplify the mathematical definitions, only infinite behaviours are considered. In order to only consider infinite behaviours, terminated behaviours are extended with infinite sequences of $\bot$ or $\top$ depending on if the behaviour was aborted or miraculous [7]. These states are referred to as *improper states* [4, 7]. Then $\sigma = \langle \sigma_0, \sigma_1, \ldots \rangle$ is a sequence of states that describes a possible behaviour of $\mathcal{A}$, if the following conditions hold [4, 7]:

- the initial state satisfies the initialisation condition, $A_0.\sigma_0$
- if σ_i is improper then $\sigma_{i+1} = \sigma_i$
- if σ_i is proper then either:
  - The system aborts, $\neg\mathsf{t}.A.\sigma_i$ and $\sigma_{i+1} = \bot$, or,
  - behaves miraculously, $\mathsf{t}.A.\sigma_i \land \neg\mathsf{g}.A.\sigma_i$ and $\sigma_{i+1} = \top$, or,
  - executes normally, $\mathsf{t}.A.\sigma_i \land \mathsf{g}.A.\sigma_i \land A.(\lambda\sigma \cdot \sigma = \sigma_{i+1}).\sigma_i$

Behaviours contain local variables that cannot be observed. What can be observed is a trace $\langle z.\sigma_0, z.\sigma_1, \ldots \rangle$ of a behaviour $\langle \sigma_0, \sigma_1, \ldots \rangle$ where the global variables z have been extracted. The semantics of action system $\mathcal{A}$ is then a set of observable traces of behaviours [4, 7].

8.2 Refinement

Refinement of an action system $\mathcal{A}$ means replacing it by another system that is indistinguishable from $\mathcal{A}$ by the environment [4, 7]. On the extended state space $\Sigma \cup \{\bot, \top\}$ an ordering of traces over global variables is defined:

$$\langle z.\sigma_0, z.\sigma_1, \ldots \rangle \preceq \langle z.\tau_0, z.\tau_1, \ldots \rangle \mathrel{\hat{=}} (\forall i \cdot z.\sigma_i = z.\tau_i \lor \sigma_i = \bot) \tag{16}$$

Consider two action systems $\mathcal{A}$ and $\mathcal{A}'$. Refinement is then defined as:

$$\mathcal{A} \sqsubseteq \mathcal{A}' \mathrel{\hat{=}} (\forall t' \cdot t' \in \mathsf{tr}.\mathcal{A}' \Rightarrow (\exists t \cdot t \in \mathsf{tr}.\mathcal{A} \land t \preceq t')) \tag{17}$$

Here $\mathsf{tr}.\mathcal{A}$ denotes all traces that are generated by $\mathcal{A}$. The refinement definition states that for each trace in the refined system $\mathcal{A}'$, there exists a corresponding trace in the abstract system $\mathcal{A}$.

This definition of refinement is not practical for use in proofs [4, 7]. Instead refinement can be proved using the notion of data refinement discussed earlier. Consider again two action systems $\mathcal{A}$ and $\mathcal{A}'$. Assume that the concrete system $\mathcal{A}'$ has local variables c and the abstract system $\mathcal{A}$ has local variables a. To prove that $\mathcal{A} \sqsubseteq \mathcal{A}'$, we use an decoding statement $\Delta = \{+a - c|Q\}$ that maps the concrete state space to the abstract state space [3, 6]. Action system $\mathcal{A}$ is then data refined by $\mathcal{A}'$ using decoding statement Δ, if the following conditions hold [6, 4, 7]:

$$A'_0 \Rightarrow \Delta.A_0 \tag{18}$$

$$\Delta; A \sqsubseteq A'; \Delta \tag{19}$$

$$\Delta.(\mathsf{g}.A \land \mathsf{t}.A) \Rightarrow \mathsf{g}.A' \tag{20}$$

The first refinement condition (18) concerns correct refinement of the initialisation. For each possible initialisation in the concrete system, there must exist a corresponding initialisation in the abstract system. The second condition (19) concerns refinement of the action. For each action step in the concrete system there must exist a corresponding step in the abstract system. The last rule (20) states that the concrete system cannot deadlock more often than the abstract one. Note that these conditions are sound, but not complete [4].

8.3 SDF graphs as action systems

Complete SDF graphs can be viewed as action systems. Complete here means that the there are no unconnected in-ports. All nodes in a minimal PASS are executed once and the time is then advanced by one sampling time. The exact time is unimportant here, since only systems with one sampling time and no continuos behaviour are considered. It can therefore be considered to advance with one tick. We can view the SDF graph G as the action system $\mathsf{act}.G$:

$$\begin{array}{rll} \mathsf{act}.G \mathrel{\hat{=}} & |[& \textbf{var } \mathsf{v}.d_1, \ldots, \mathsf{v}.d_2; \\ & & \textbf{init } \mathit{Init} \\ & & \textbf{do } \mathsf{trans}.G \textbf{ od} \\ &]| : & \langle \mathsf{v}.b_1.p_1, \ldots, \mathsf{v}.b_m.p_k \rangle \end{array} \tag{21}$$

Here $d_1, \ldots, d_n$ are the delays in the graph, $b_1, \ldots, b_m$ the nodes in the graph and all p_i are ports. The initialisation of delays is denoted by Init. The ports of the nodes in the graph are considered to give the observable state. The action system iteratively executes the statement $\mathsf{trans}.G$ obtained by translating the SDF graph using Definition 3. This interpretation now gives a definition of a refinement relation between SDF graphs. This will later be used for compositional verification.

8.4 Correctness of the translation

In order for the translation from SDF graph to sequential program to be correct, the result from the sequential program should be the same regardless of which possible PASS that is used. For deterministic programs a proof is given in [25]. Furthermore, the program should be non-deadlocking. When considering iterated execution of the graph in our setting, miraculous execution of the translated statement corresponds to termination of the action system (see Subsection 8.1).

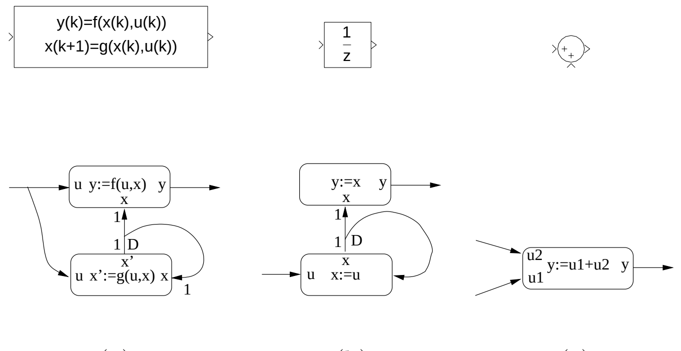
Consider two statements T_1 and T_2 obtained from two different PASS for the same SDF graph. By the same reasoning as in [25] the statements will have the same result if $\mathsf{t}.T_1 \wedge \mathsf{g}.T_1$ and $\mathsf{t}.T_2 \wedge \mathsf{g}.T_2$ hold in the before state. In case we have the situation that $\neg\mathsf{g}.T_1$ and $\neg\mathsf{t}.T_2$ hold (or vice versa) the two statements behave differently. The first program will in this case behave miraculously and the second abort. Two programs obtained from an SDF graph are thus correct in case they do not behave miraculously, $\{\mathsf{g}.T_i\}; T_i = \{\mathsf{g}.T_j\}; T_j$. Later we will show that we will only have non-miraculous programs, i.e., programs T where $\mathsf{g}.T = \mathit{true}$.

9 SDF graph representation of Simulink models

Discrete Simulink models consist of graphical data flow diagrams which are similar to SDF graphs. However, a Simulink block is not exactly the same as a

Simulink:

SDF:



(a) (b) (c)

Figure 5: (a) SDF representation of general Simulink block, (b) a *Unit delay*-block and (c) a *Sum*-block

node in the SDF notation. In this section we present how to map a number of important blocks to their corresponding SDF representation.

9.1 Mapping Simulink blocks to nodes

We can differentiate between the following important Simulink blocks:

- *Functional blocks*: Blocks in the Simulink library that directly encapsulates a difference equation
- *In and out blocks*: Blocks used to obtain inputs from in-ports of the containing subsystem, as well as export values to the out-ports
- *Virtual subsystem blocks*: Subsystem blocks that are used for structuring Simulink models, but have no impact on behaviour
- *Atomic subsystem blocks*: Subsystem blocks that are used for structuring Simulink models and that are executed as atomic units

Functional blocks Consider again a Simulink block with the general form in (1). The behaviour of the block is described by two equations, which are not necessarily executed together. The implementation of the block as an SDF graph is shown in Figure 5 (a). There are several special cases: consider, e.g., the *Unit delay*-block and the *Sum*-block. The blocks with their SDF representations are shown in Figure 5 (b) and (c), respectively.

In and out blocks In and out blocks correspond to in and out nodes in the SDF graphs.

Hierarchical diagrams Simulink diagrams are hierarchical and this has to be taken into account in the mapping to SDF graphs. The diagrams are structured using *virtual* and *atomic* subsystem blocks. Virtual subsystems are only used to syntactically group different blocks together and they do not have any affect on the behaviour of the Simulink models. Since execution of blocks from two virtual subsystems might have to be interleaved, we cannot translate virtual subsystem blocks individually and then compose the result. To handle this problem, the virtual subsystem hierarchy is flattened during the translation of the diagrams.

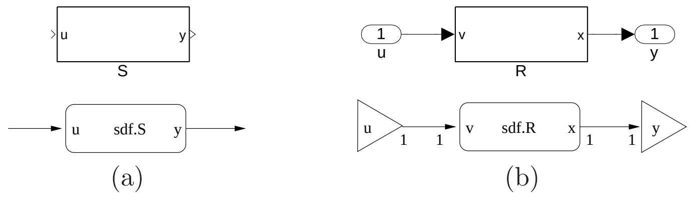


Figure 6: (a) An atomic Simulink subsystem S and the corresponding SDF node and (b) the contents of S and its corresponding SDF representation

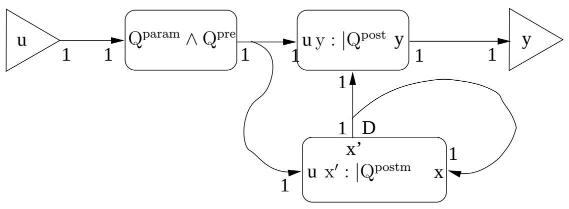


Figure 7: SDF graph obtained from the contract specification of an atomic subsystem

The atomic subsystems on the other hand are mapped to SDF nodes themselves. The content of an atomic subsystem is translated recursively to an SDF graph, which then become the content of the SDF node corresponding to the subsystem. Consider an atomic subsystem S with in-ports u and out-ports y in Figure 6. Its SDF representation (denoted $\mathsf{sdf}.S$) is obtained by recursively translating its content.

9.2 Mapping a subsystem contract description to an SDF graph

One goal of the method given in this paper is to use the contract descriptions of subsystems instead of the subsystem diagrams when analysing models. From the contract description we can directly obtain the most abstract statement that satisfies the contract. The most abstract statement that satisfies a specification concerning variables x with precondition Q^{pre} and a postcondition Q^{post}, is $\{Q^{pre}\}; x : |Q^{post}$ [5].

Assume we have subsystem S in Figure 6 (a), which is described by the contract C in equation (3). We then get the SDF graph representation, $\mathsf{sdf}.C$, shown in Figure 7 for the contract. Note that this is very similar to the translation of the general Simulink block in Figure 5. This is because the contract give an abstract description of the same type of behaviour. This is the most abstract description of S that can be used when analysing models where the subsystem is used. The sequential program statement $\mathsf{trans}.(\mathsf{sdf}.C)$ can now be obtained. This is again done according to the translation procedure in Definition 3. After simplifications we get the statement:

$$\mathsf{trans}.(\mathsf{sdf}.C) \hat{=} \{Q_v^{param} \wedge Q_v^{pre}\}; \mathsf{v}.S.y : |Q_v^{post}; \mathsf{v}.d : |Q_v^{postm} \tag{22}$$

Here we have directly removed the intermediate ports and used the subsystem ports u, y and delay d instead. In each condition subscripted by v the occurrence of port p or delay d has been replaced by its corresponding variable identifier $\mathsf{v}.S.p$ or $\mathsf{v}.d$.

10 Verification with respect to contracts

In order to do compositional verification of Simulink models, we need to show that the use of a subsystem implementation instead of its contract description preserves the behaviour (i.e. refines) the complete system. To prove preservation of behaviour of the complete system, we use the action system representation (21) and the refinement rules for action systems (18)-(20). Assume we have an atomic subsystem M with contract C. Thus the abstract program obtained when using the contract description $\mathsf{trans}.(\mathsf{sdf}.C)$ should be data refined by the program obtained when the translated diagram $\mathsf{trans}.(\mathsf{sdf}.M)$ is used instead. The abstract program obtained from a Simulink model where M is used can be written as $S_1; \mathsf{trans}.(\mathsf{sdf}.C)[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2$. The concrete program is then given as $S_1; \mathsf{trans}.(\mathsf{sdf}.M)[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2$. In the complete translation all connected in-ports of M are replaced by the port or block memory they are connected to. This is here denoted with the substitution $[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]$, where p_i are in-ports of subsystem M and $\mathsf{conn}.p_i$ denotes the out-ports or delays those ports are connected to. We thus need to prove:

$$\begin{aligned}&\Delta; S_1; \mathsf{trans}.(\mathsf{sdf}.C)[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2 \sqsubseteq\\ &S_1; \mathsf{trans}.(\mathsf{sdf}.M)[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2; \Delta\end{aligned} \tag{23}$$

Since the data refinement only concerns M, the decoding Δ is assumed to only concern the internal variables of $\mathsf{trans}.(\mathsf{sdf}.C)$ and $\mathsf{trans}.(\mathsf{sdf}.M)$. In practice the decoding Δ is of the form $\Delta \mathrel{\hat{=}} \{-\mathsf{v}.b_n.p_n, \mathsf{v}.d_n + \mathsf{v}.x | Q^{refrel}\}$, where p_n denotes the new out-ports, d_n denotes new delays obtained from Simulink block memories and x denotes specification variables in contract C. Recall that Q^{refrel} (see (3)) is a predicate that relates the specification variables in contract C with the block memories and specification variables in the diagram inside M.

Since the variables of Δ and S_1, as well as Δ and S_2 are disjoint, we have that $\Delta; S_1 \sqsubseteq S_1; \Delta$ and $\Delta; S_2 \sqsubseteq S_2; \Delta$. To prove (23) we then need to show that:

$$\Delta; \mathsf{trans}.(\mathsf{sdf}.C) \sqsubseteq \mathsf{trans}.(\mathsf{sdf}.M); \Delta \tag{24}$$

Proof.

$$\begin{aligned}
&\Delta; S_1; \mathsf{trans}.(\mathsf{sdf}.C)[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2\\
\sqsubseteq\ &\{\text{Assumption above}\}\\
&S_1; \Delta; \mathsf{trans}.(\mathsf{sdf}.C)[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2\\
=\ &\{\mathsf{v}.M.p_i, \mathsf{v}.(\mathsf{conn}.p_i) \text{ not free in } \Delta\}\\
&S_1; (\Delta; \mathsf{trans}.(\mathsf{sdf}.C))[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2\\
\sqsubseteq\ &\{\text{Assumption (24) and } \mathsf{v}.(\mathsf{conn}.p_i) \text{ not free in } \mathsf{trans}.(\mathsf{sdf}.M)\}\\
&S_1; (\mathsf{trans}.(\mathsf{sdf}.M); \Delta)[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2\\
=\ &\{\mathsf{v.M.p_i}, \mathsf{v}.(\mathsf{conn.p_i}) \text{ not free in } \Delta\}\\
&S_1; \mathsf{trans}.(\mathsf{sdf}.M); [\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; \Delta; S_2\\
\sqsubseteq\ &\{\text{Assumption above}\}\\
&S_1; \mathsf{trans}.(\mathsf{sdf}.M)[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2; \Delta
\end{aligned}$$

□

Note also that if all subsystems are implemented as deterministic diagrams, then the corresponding statements do not behave miraculously [5]. The SDF graph obtained from the Simulink model is thus non-terminating, which is the requirement for correct translation from SDF graph to sequential program stated in Subsection 8.4.

10.1 Example of subsystem refinement

To give an example of the translation of Simulink models and the verification method, the PID-controller from Section 3 is used. The subsystem, *PID*, implementing the PID-controller is shown in Figure 1 (a). The contract C associated with the subsystem is given in (4). The contract specification of the subsystem is translated to a sequential program statement as described in (22):

$$
\begin{aligned}
&\mathsf{trans}.(\mathsf{sdf}.C) \mathrel{\hat{=}} \\
&\quad \{T_s > 0 \wedge T_i > 0 \wedge T_d \geq 0 \wedge N \in [3, 30]\}; \\
&\quad \mathsf{v}.PID.u : |\mathsf{v}.PID.u' = K(\mathsf{v}.PID.u_c - \mathsf{v}.PID.y) + \tfrac{KT_s}{T_i}(\mathsf{v}.PID.u_c - \mathsf{v}.PID.y) \\
&\qquad + i_1 + \tfrac{KT_d N}{(T_d + NT_s)}(\mathsf{v}.PID.y - y_1); \\
&\quad y_1, i_1 : |i_1' = \tfrac{KT_s}{T_i}(\mathsf{v}.PID.u_c - \mathsf{v}.PID.y) + i_1 \wedge y_1' = \mathsf{v}.PID.y
\end{aligned}
$$

The statement above should then be refined by the translated diagram inside the subsystem shown in Figure 1 (b). The contents of the subsystems P, I and D in that diagram are given in Figure 2. One possible translation of the diagram inside *PID* is then given as:

$$
\begin{aligned}
&\mathsf{trans}.(\mathsf{sdf}.PID) \mathrel{\hat{=}} \\
&\quad \mathsf{trans}.(\mathsf{sdf}.P); \\
&\quad \mathsf{v}.delay1.y := \mathsf{v}.delay1; \\
&\quad \mathsf{trans}.(\mathsf{sdf}.I); \\
&\quad \mathsf{v}.delay2.y := \mathsf{v}.delay2; \\
&\quad \mathsf{trans}.(\mathsf{sdf}.D); \\
&\quad \mathsf{v}.sum.y := \mathsf{v}.P.p + \mathsf{v}.I.i + \mathsf{v}.D.d; \\
&\quad \mathsf{v}.PID.u := \mathsf{v}.sum.y; \\
&\quad \mathsf{v}.delay1 := \mathsf{v}.I.i; \mathsf{v}.delay2 := \mathsf{v}.PID.y
\end{aligned}
$$

In case P, I and D are described by contracts, that description of the subsystems can be used. Otherwise the translation process proceeds recursively through the subsystem hierarchy. The block memories from blocks *delay*1 and *delay*2 relate to the specification variables y_1 and i_1 as described by Q^{refrel} in (4). The refinement rule (24) for subsystem refinement leads to the condition:

$$
\begin{aligned}
&\{-\mathsf{v}.P.p, \mathsf{v}.I.i, \mathsf{v}.D.d, \ldots, \mathsf{v}.delay1, \mathsf{v}.delay2 + y_1, i_1 | Q^{refrel}\}; \\
&\mathsf{trans}.(\mathsf{sdf}.C) \\
&\sqsubseteq \\
&\mathsf{trans}.(\mathsf{sdf}.PID); \\
&\{-\mathsf{v}.P.p, \mathsf{v}.I.i, \mathsf{v}.D.d, \ldots, \mathsf{v}.delay1, \mathsf{v}.delay2 + y_1, i_1 | Q^{refrel}\}
\end{aligned}
$$

The tool that is described in the following sections has been used to verify this refinement. It generates the necessary proof obligation and it then uses the SMT-solver Z3 [18] to perform the proof.

11 Tool introduction

Tool support for the approach outlined in the paper has been developed [11]. The tool takes a Simulink model annotated by contracts as argument. The tool checks that each atomic subsystem (with a contract) satisfies its contract. This is done using the translation process outlined in the paper.

To have standard location for the contracts, they are written down as text in the *Description*-field of the subsystems, analogously to how JML-contracts

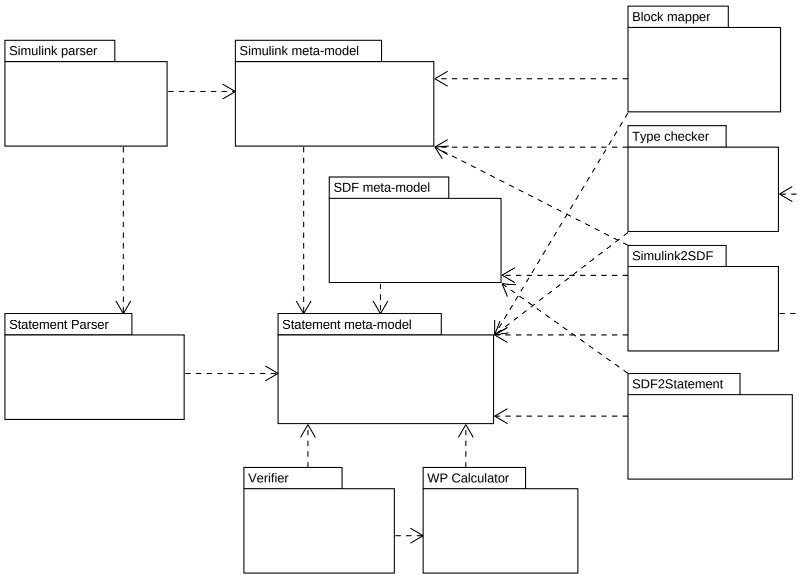


Figure 8: The architecture of the verification tool

are written as comments in the Java-code [14]. This also ensures that a contract is always associated with the correct subsystem and that the contracts do not interfere with other tools. Correctness of subsystems with respect to contracts are given by (24). To prove that condition, rule (14) is used. This rule provides formula in first-order logic (if the contracts are first order). Using this rule, the tool produces a set of proof obligations in the SMT-LIB format [34], which is a standard format read by most SMT-solvers. These proof obligations are then discharged by an SMT-solver (currently Z3 [18] is used). Finally the output from solver is handled and the results displayed in a graphical user interface.

The following sections describe the architecture and functionality of the tool. First, the the tool architecture is described. We then describe the individual components of the tool and give an overview of their functionality.

12 Tool architecture

The tool is implemented as a set of model-transformations. The models are described using meta-models created using the Eclipse Modelling Framework (EMF) [35, 22], which is a mature modelling framework from the Eclipse Foundation. One of the goals of EMF is to simplify development of domain specific modelling languages. Developers can create their own domain specific modelling languages based on the meta-model provided by EMF. This meta-model is similar to MOF [32] from OMG.

The architecture of the tool is shown in Figure 8. The tool has a blackboard architecture, where functional components manipulate instances of the meta-models. There are meta-models to describe Simulink models, SDF graphs, as well as sequential statements and contracts. The parser components convert the textual representations of contracts, statements and Simulink models into meta-model instances. The model instances are then further manipulated by different tools. The following sections give more detailed overviews of the components. We first start with a description of the meta-models used in the tool.

13 The Simulink meta-model

The Simulink meta-model describes all necessary information needed to represent the functional aspects of Simulink models relevant for verification. Instances of the meta-model can thus not store all information of the graphical layout of models and different code generation options. They could be included as extra parameters, but currently this has not been deemed necessary. The meta-model for Simulink models used by the tool is shown in Figure 9.

A Simulink model is represented as an instance of the class *Model*. A model contains a *Diagram* object. The diagram can then contain different kinds of blocks. Each block can have in-ports and out-ports. The association *dep* represents the dependencies between in- and out-ports given by the signals in the corresponding Simulink diagram. The association *fDep* represents the internal dependencies between ports in a block. Each block also has a reference to a set of parameters. These parameters are imported directly from Simulink and are used to fine-tune behaviour and looks of the block. Each block also has a reference to a *BlockParameterDefaults* object. This object stores common parameter values for each block type. This approach is also used in Simulink. The purpose is to save memory when common data is only stored once.

There are different kinds of blocks implemented as subclasses of the class *Block*. The class *FunctionalBlock* represents normal functional blocks. These blocks contain a statement describing the behaviour of the block. They can also contain internal memory. Each memory is represented by an instance of the *Memory* class. In- Out- and Enable-blocks are also represented as subclasses of class *Block*, since these types of blocks need special treatment from functional blocks. Subsystems are also special blocks that contain a diagram. This enables representation of hierarchical diagrams. Atomic subsystems are special subsystems that can contain a contract.

Many meta-models of Simulink already exist. The goal of most of them are to represent all of Simulink, not only functional aspects. These meta-models are then used for model-validation [20, 21] or/and model transformations [39, 31]. The goal of our meta-model is to give a more compact description of the functional aspects of Simulink and thereby be more convenient basis for the verification tool. Furthermore, our meta-model is based on EMF, which makes it easy to manipulate the models in Java. This is important for future development, since it is anticipated that students will work on the tool. They are most familiar with Java and can therefore start work directly without learning a new language first. Furthermore, a lot of model validation and model transformation tools have already been developed for EMF based models [22].

14 The SDF meta-model

This meta-model is used to describe SDF graphs. It is similar to the Simulink meta-model, but simpler. The meta-model is shown in Figure 10. It can represent graphs that contain different types of nodes and the connections between them. The *composite nodes* can contain graphs themselves. They can also contain a contract that give an abstract description of the node behaviour. The *conditional node* is used to represent conditionally executed subsystems. The goal is that the meta-model should be able to represent the functional aspects of SDF graphs that are needed by the tool.

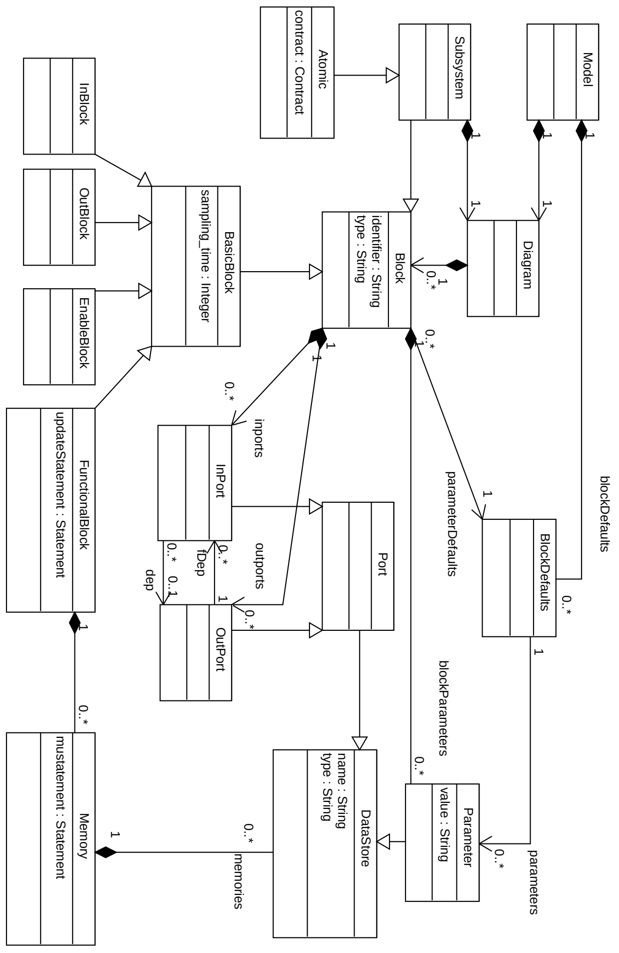


Figure 9: The meta-model that describes the structure of Simulink models

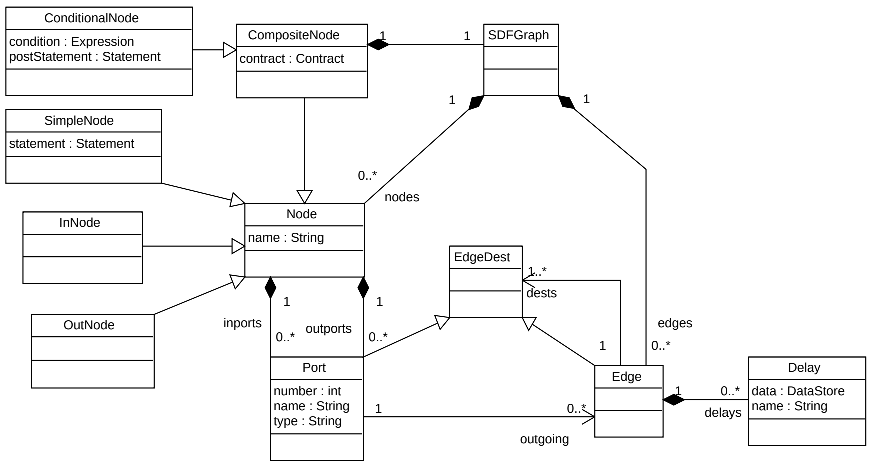


Figure 10: The meta-model that describes the structure of SDF graphs

15 The statement meta-model

The statement meta-model represents the abstract syntax of statements, expressions and contracts. The reason for keeping this a separate meta-model is that it is self contained and it is needed both by the Simulink and SDF graph meta-models. Furthermore, this meta-model also concerns constructs (statements and contracts) that are not part of the original Simulink. To have a separate meta-model seemed like the most elegant solution. The choice of EMF to represent the abstract syntax might seem redundant as the parser generator used in the parser component can already generate an abstract syntax tree. However, by using EMF based models everywhere, the same techniques can be used for all model manipulations. The abstract syntax can be described as:

$$\begin{array}{lll}
\mathit{Exp} & ::= & \mathit{BinExp} \mid \mathit{UnExp} \mid \mathit{Atom} \mid \mathit{IfThenElse} \mid \mathit{FunCall} \\
\mathit{BinExp} & ::= & \mathit{Exp}\ (+| - | * |/| \leq | \geq | = | \Rightarrow | \wedge | \vee |\mathsf{mod})\ \mathit{Exp} \\
\mathit{UnExp} & ::= & \mathit{QuantPred} \mid \mathit{NegPred} \mid \mathit{UnNumExp} \\
\mathit{Atom} & ::= & \mathit{Identifier} \mid \mathit{PredConst} \mid \mathit{Number} \\
\mathit{IfThenElse} & ::= & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Exp}\ \mathsf{else}\ \mathit{Exp}\ \mathsf{end} \\
\mathit{FunCall} & ::= & \mathit{Identifier}\ (\mathit{Exp})+ \\
\mathit{QuantPred} & ::= & (\forall|\exists)\ \mathit{Identifier} : \mathit{Type} \cdot \mathit{Exp} \\
\mathit{NegPred} & ::= & \neg \mathit{Exp} \\
\mathit{UnNumExp} & ::= & -\mathit{Exp} \\
\mathit{PredConst} & ::= & (1|0|\mathsf{true}|\mathsf{false}) \\
\mathit{Type} & ::= & (\mathsf{boolean}|\mathsf{double}|\mathsf{int32}|\mathsf{int16}|\mathsf{int8}|\mathsf{uint32}|\mathsf{uint16}|\mathsf{uint8})
\end{array} \tag{25}$$

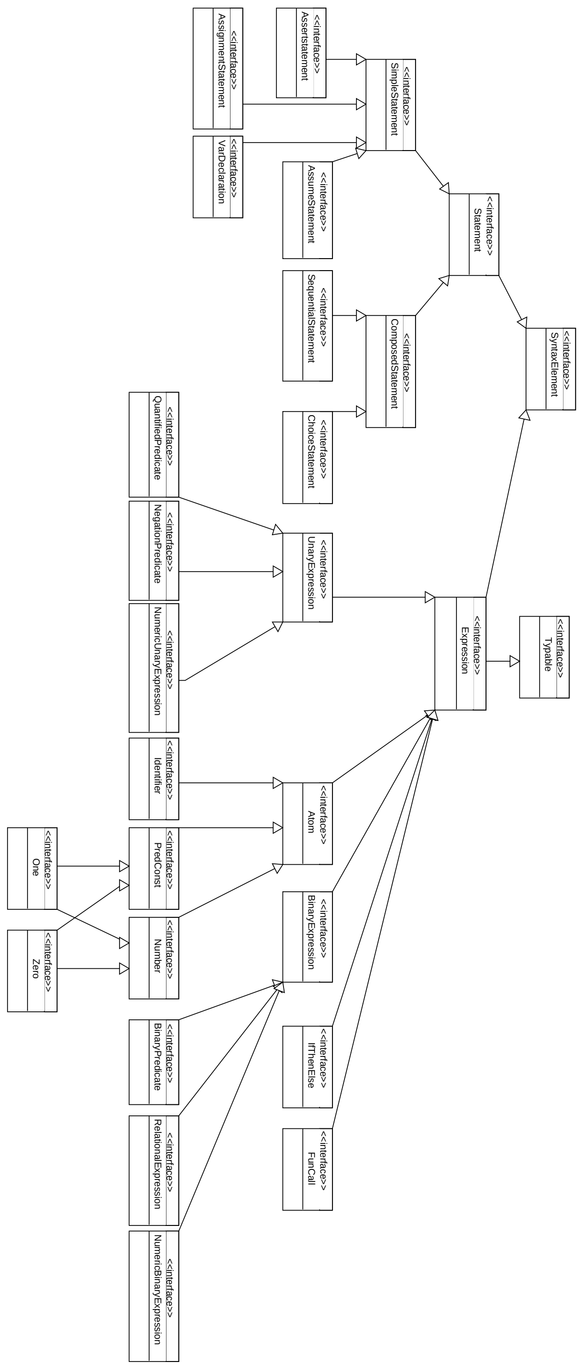
Note that we do not really separate predicates and expressions at the abstract syntax level. The reason is that Matlab/Simulink also treats predicates as expressions. Therefore, we have made this design descision in order to handle all Matlab expressions. The type checking then assigns types to the nodes in the abstract syntax tree (AST) and thereby determines if a node is a numeric expression or a predicate. The inheritance hierarchy is given in Figure 11. This gives a convenient architecture to handle syntax elements both for type-checking and verification condition generation. Note that we have subclasses for predicates. This enables easier type-checking.

The abstract syntax of statements was already described in (8) in Section 6. The EMF representation is a direct implementation of that. The semantics of these statements was also already presented in Section 6.

16 The Simulink parser

The Simulink parser reads a Simulink model file and produces a model representation that conforms to the Simulink meta-model in Figure 9. In order to achieve this goal, the files containing Simulink models have to be parsed, producing a abstract syntax tree (AST) that can be used to generate EMF models. Simulink model files are normal text files that can be easily parsed.

The selection of parser generator was made between ANTLR and Java compiler compiler (JavaCC), which are two well-established frameworks [37, 36]. JavaCC was chosen due to its established user-base and close relation to Java, which is the main development language for the overall project. Once the relatively steep learning curve of JavaCC was overcome, a parser was created that was able to analyse Simulink model-files and create an AST representing the phrase structure of said files. The following step was analysis of the AST through the use of a visitor pattern.



<<interface>> AssertStatement
<<interface>> AssignmentStatement
<<interface>> VarDeclaration
<<interface>> SimpleStatement
<<interface>> AssumeStatement
<<interface>> Statement
<<interface>> SequentialStatement
<<interface>> ComposedStatement
<<interface>> SyntaxElement
<<interface>> ChoiceStatement
<<interface>> QuantifiedPredicate
<<interface>> NegationPredicate
<<interface>> UnaryExpression
<<interface>> NumericUnaryExpression
<<interface>> Expression
<<interface>> Typable
<<interface>> Identifier
<<interface>> Atom
<<interface>> One
<<interface>> PredConst
<<interface>> Zero
<<interface>> Number
<<interface>> BinaryExpression
<<interface>> BinaryPredicate
<<interface>> IfThenElse
<<interface>> RelationalExpression
<<interface>> FunCall
<<interface>> NumericBinaryExpression

Figure 11: The inheritance hierarchy of statements and expressions in the abstract syntax representation

JavaCC comes with integrated tree-generation functionality that goes under the name JJTree [37]. The JavaCC grammar developed earlier was augmented with JJTree node declarations that define when nodes are to be added to the AST. JJTree also uses the node declarations to generate a data structure for a AST and, furthermore, creates a Java interface for the visitor. The visitor interface was implemented and a visitor capable of generating EMF model elements based on a AST was created. The visitor traverses a AST and adds elements to the EMF-model when nodes are encountered. In practice this means that once a branch has been traversed, the EMF model element it corresponds to is added to the parent element.

Simulink stores certain block attributes in a section of the model-file called *BlockParameterDefaults*, which is separate from the actual block declarations. The distributed manner in which data is stored in the Simulink model is also present in the generated ASTs. This means that some EMF model elements have to be generated based on information stored in several different branches of the AST. In order to solve this problem the attributes declared in the section *BlockParameterDefaults* of the Simulink model-file are stored in a hash-map that uses block-types as keys. Since the *BlockParameterDefaults* sections are higher in the tree hierarchy than the actual block declarations, the hash-map contains all relevant attributes when EMF model blocks are being generated. This means that a block that is being generated can be associated with corresponding attributes in the hash-map by using the type of the block as a key.

The steps described above result in an EMF model corresponding to the Simulink model that is being analysed. Through the use of well-established parsing and compilation paradigms a source file is analysed and the resulting AST is used to generate EMF models that follow the structure defined in the Simulink meta-model. Details of the Simulink model file format and the parsing can be found in [23].

17 The statement parser

This section provides an overview of the parsing of contracts and program statements. The most interesting topic is here the concrete syntax of the languages used by the verification tool. We can differentiate between the grammars for the following constructs:

- Expressions - Provides the grammar for expressions and predicates.
- Contracts - Describes the format of contracts.
- Statements - Describes the grammar for program statements

Both the contract grammar and statement grammar use the grammar for expressions.

17.1 Expressions

The grammar for predicates and expressions is given as:

```
PredExp::= UnPredExp [(&&| || | ==> | <==>) PredExp]
UnPredExp::= RelExp | ~UnExp | ExistPred | ForallPred
ExistPred::= \exists VarDeclaration (, VarDeclaration)* "." Exp
ForallPred::= \forall VarDeclaration (, VarDeclaration)* "." Exp
RelExp::= Exp [(<= | == | >= | <|>) Exp]
```

```
Exp::= Term [(+|-) Exp]
Term::= ETerm [(*|/|mod) Term]
ETerm::= Atom [^ ETerm]
Atom::= "-" Atom | Atom "'" |"(" PredExp ")" | Number |
    PredConst | Identifier | IfExp | FunCall
IfExp::= "if" Exp "then" Exp "else" Exp "end"
FunCall::= Identifier "(" ArgList ")"
ArgList::= Expression ["," ArgList]
VarDeclaration::= Identfier ":" Type
Type::= "double"  | "boolean" | "int32" | ...
```

As can be seen from the definition, the grammar is heavily based on the Matlab grammar for expressions. The goal is that an expression written using this language should be a valid Matlab expression. However, the **if** − **then** − **else** expression does not exist in Matlab. It was introduced here since it is often convenient to use in the specifications.

17.2 Contracts

The format of contracts was already described in Section 4. The concrete syntax of the contracts is given by the following grammar:

```
Contract ::=
   "contract:"
   (
       Parameters |
       Inports |
       Outports |
       Memory |
       ParameterCondition |
       PreCondition |
       PostCondition |
       InitCondition |
       PostConditionMemory |
       RefRel
   )+
  "end"
Parameters::= "parameters:" VarDeclaration (";" VarDeclaration )*
...
ParameterCondition::= "paramcond:" Expression
...
```

The "..." means that the rest of the productions for the contract clauses in (3) follow the same pattern. Note that each contract clause can occur multiple times in a contract. Furthermore, they can be given in any order.

17.3 Statements

The statements that can be used are the ones already described in Section 6. Their concrete syntax is given by the following grammar:

```
Statement::= SeqStatement ["/\" Statement]
SeqStatement::= UnaryStatement [";" SeqStatement]
UnaryStatement::= "skip" | "{" PredExp "}" | "[" PredExp "]" |
   AssignmentStatement | "(" Statement ")"
AssignmentStatement::=
   IdentifierList ("=" ExpressionList | ":|" PredExp)
```

```
IdentifierList::= Identifier ["," IdentifierList]
ExpressionList::= PredExp ["," PredExp]
```

Note that we use the same symbol (=) for assignment as Matlab. Equality is denoted ==.

17.4 Implementation

As for the parser of Simulink models, the parser for the constructs above has been constructed using JavaCC and JJTree. The abstract syntax tree generated by JJTree is not used directly in the tool. The tree is only used as an intermediate step for creating an EMF version of the syntax tree discussed in Section 15. The reason for this is that it makes the treatment of Simulink models, contracts and statements uniform.

18 The block mapper

The functionality of blocks in Simulink is only implicitly defined. The functionality can be derived from the block type and the block parameters, but it is almost never explicitly given in the Simulink model file. To obtain the functionality for each block the blockmapper component was introduced to map functionality to block types. Function definitions are stored in a file using an XML format. Each supported block type is defined in this file. The definitions have been obtained by reading the Simulink documentation and by testing the behaviour of the blocks.

Each block without memory is associated with a statement of the form $y := f.param.u$, where y is the out-ports, f a function, $param$ some parameters and u the in-ports. In case of blocks with memory there are two statements: one for out-ports and one for the memory $y := f.param.x.u$ and $x := g.param.x.u$. This was already discussed in Section 3.

The statements are stored in a file as a string with placeholders for parameters. Block mapper then replaces the placeholder with the actual value of the parameter. Consider as an example the *Gain* block. The statement in the file is the following: *y1:=@Gain*u1*, where *@Gain* is a placeholder for a parameter with the name *Gain*. When all parameter placeholders have been replaced the block mapper calls the statement parser to create an EMF-representation of the statement. The EMF-representation of the statement is then added to the block in the current Simulink meta-model instance.

Some blocks have too complicated definitions to be represented in the XML format. In these cases the function definition refers to "handlers", which are implemented in code and produce the statements.

Block mapper also adds information on internal dependencies between in-ports and out-ports inside blocks to the Simulink meta-model. By dependency is here meant that the value of the out-port depends on the in-ports at the current sampling time. These dependencies are stored as boolean values for each in-port in the function definition file.

18.1 File format description

The functions are defined in a file using the XML format. The file contains an update statements and other needed information about each supported block. The following is an example of a complete block type definition:

```
<block type="Gain">
  <parameter name="Gain" />
  <statement expr="y1=@Gain*u1" />
  <dependency>
    <inport name="u1" dependent="true" />
  </dependency>
</block>
```

This describes the definition for the block type *Gain*. According to the definition the Gain block has a parameter *Gain* and the function is *y1=@Gain*u1*, where *y1* is the out-port, *u1* is the in-port and *Gain* is the value of the *Gain* parameter. It is also stated that the value of the out-port depends on the in-port.

There are blocks which cannot be represented in the format above. Examples of such cases are blocks with memory and blocks with a variable number of inputs. The following is an example of a block which includes a memory:

```
<block type="UnitDelay">
  <parameter name="X0" />
  <memory name="X">
    <init expr="X==@X0" />
  </memory>
  <statement expr="y1=X" />
  <mstatement expr="X=u1" />
  <dependency>
    <inport name="u1" dependent="false" />
  </dependency>
</block>
```

The above definition defines a block with a memory, *X*, together with the update and initialisation statements for the memory. Another special definition is the following one:

```
<block type="Logic">
  <parameter name="Operator">
    <replace oldValue="AND" newValue="&amp;&amp;" />
    <replace oldValue="OR" newValue="||" />
    <replace oldValue="NOT" newValue="~" />
  </parameter>
  <statement type="complex" prependIfUnary="true" >
    <start expr="y1=" />
    <separator expr="@Operator" />
    <end expr="" />
  </statement>
  <dependency>
    <inport dependent="true" />
  </dependency>
</block>
```

Here the *statement* element is defined as complex, which means that it consists of three different elements, the *start*, *separator* and *end* elements. The statement starts with the expression in the *start* element, after this every in-port (*u1*,*u2*,...) is added to the statement separated by the expression in *separator*. The last part of the statement is the expression in the *end* element. The main purpose of this construct is to make it possible to define statements that depends on the number of in-ports in a general way. The *prependIfUnary* attribute defines that the separator element should be added before the in-port if there is only one in-port in the block. These, so called, complex statements can also be nested in arbitrary many levels.

The above definition also contains *replace* elements for a parameter. This tells the application to substitute the parameter value with another value. Parameters can also contain another parameter, one example of this is the *Switch*-block.

Some blocks are too complex to be represented in an XML file in a simple way. In these cases special block handlers are implemented in code. Every handler should have a unique name which is used for reference in the XML file. This is done by specifying the statement in the following way:

```
<statement handler="NameOfHandler" />
```

where *NameOfHandler* is the name returned by the *getName* method of the handler.

18.2 Supported blocks

The goal is that the tool should support all blocks in the standard Simulink library. Currently there is a long list of supported blocks. They include many common mathematical and logical functional blocks, switches, subsystems, and delay blocks. However, the Simulink block library is fairly large and, hence, there are currently also many unsupported block types. All blocks involving creation of matrix signals and blocks for manipulation of such signals are not supported. Blocks that are also not supported include, S-function and Matlab-function blocks, Look-up tables, all continuous blocks, most discrete blocks and the model verification blocks. Other conditionally executed subsystems than the enabled subsystem are also not supported yet, which include all iterated subsystems. References to library blocks is another type of construct that is not supported. Furthermore, Stateflow is unsupported.

19 The type checker

The type-checker checks and infers types for Simulink models, as well as statements and expressions. The type-checker is based on Milner's algorithm for polymorphic type-checking and type-inference [30, 33]. The algorithm consists of building a system of *type equations* that is then solved. The *type variables* in the type equations are associated with objects in a *typing environment*

Simulink is a statically typed language. The blocks are polymorphic, meaning they can handle different types of data which satisfy certain typing rules. Simulink has a type system with structures, vectors, matrices, complex numbers and different types of numbers. To make the verification tractable in this first version of the tool, only scalar primitive types are allowed. The types known by the type checker are: *double*, *boolean*, *int32*, *int16*, *int8*, *uint32*, *uint16* and *uint8*.

For each block, we build a set of type equations based on the type of the block. Table 1 describes which type equations are added and how the typing environment is modified for a number of common block types. For example, the *Sum*-block has two in-ports i_1, i_2 and one out-port o_1. The type equations together with the mapping in the type environment then state that the types of both the in-ports are the same and that the type of the out-port is the same as the type of the in-ports. In the *InPort*-block, the in-port actually denotes

Table 1: Overview of the typing of Simulink blocks

Block type	In-ports	Out-ports	Type equations	Typing environment ϵ
Sum	i_1, i_2	o_1	$\alpha = \beta$, $\gamma = \alpha$	$\epsilon \cup \{i_1 \mapsto \alpha, i_2 \mapsto \beta, o_1 \mapsto \gamma\}$
Gain	i_1	o_1	$\alpha = \beta$	$\epsilon \cup \{i_1 \mapsto \alpha, o_1 \mapsto \beta\}$
Logical	i_1, i_2	o_1		$\epsilon \cup \{i_1 \mapsto \mathbb{B}, i_2 \mapsto \mathbb{B}, o_1 \mapsto \mathbb{B}\}$
Relational	i_1, i_2	o_1	$\alpha = \beta$	$\epsilon \cup \{i_1 \mapsto \alpha, i_2 \mapsto \beta, o_1 \mapsto \mathbb{B}\}$
Switch	i_1, i_2, i_3	o_1	$\alpha = \beta$, $\gamma = \alpha$	$\epsilon \cup \{i_1 \mapsto \alpha, i_2 \mapsto \mathbb{B}, i_3 \mapsto \beta, o_1 \mapsto \gamma\}$
InPort	i_1	o_1	$\alpha = \beta$	$\epsilon \cup \{i_1 \mapsto \alpha, o_1 \mapsto \beta\}$
InPort	i_1	o_1	$\alpha = \beta$	$\epsilon \cup \{i_1 \mapsto \alpha, o_1 \mapsto \beta\}$

the corresponding in-port of the subsystem, while the out-port gives the actual port of the block. The *OutPort*-block follows the same principle. The system of type equations is then completed by taking into account the signals between in-ports and out-ports. For a signal between in-port i and out-port o we add an equation $\epsilon.i = \epsilon.o$. This Simulink part of the type checker is in principle a simplified version of the one in [38].

In order to, at least, partially check Simulink models that contain parts the tool cannot understand, the type checking is done recursively. Each subsystem with a contract is type checked individually. In the type-checking process, subsystems with contracts inside the subsystem that is currently being processed are treated as opaque units. This means that we rely on the type specification in the contract for correctness.

Currently our type inferencer and checker is a simplification of the one in Simulink. It can only handle scalar primitive types. As seen from the type equations it will also not correctly check that e.g. booleans are not added together. A better type checking component is under development. However, type checking and inference is not our main interest with the tool and therefore this was deemed sufficient as a first version.

The statement type checker is a traditional type checker based on the same algorithm as the type-checker/inferencer for Simulink models. It checks and infers types for expressions and statements. The only special property is that 0 and 1 can be both numbers and booleans. The reason for this is that Matlab does not have a boolean type. In many cases Simulink internally uses 0 for false. This needs to be handled by the tool also. Therefore, we have made the design decision to allow 0 and 1 to be booleans. The actual type used in an expression is determined by the type-checking algorithm.

20 The Simulink to SDF translator

As described earlier, the translation from Simulink models to sequential statements used for verification is done in two steps. First the Simulink model to be verified is translated to a functionally identical SDF graph. Then that SDF graph in turn is translated to a functionally identical sequential statement. This component translates Simulink models to SDF graphs.

The translation is straightforward and it is a direct implementation of the principles outlined in Section 9. *Functional blocks* are translated to *SimpleNodes*, *InBlocks* to *InNodes*, *OutBlocks* to *OutNodes*, *Atomic subsystems* to *Compos-*

iteNodes and *conditionally executed atomic subsystems* to *ConditionalNodes*.

The translation proceeds recursively from the initial Simulink model object through the subsystem hierarchy. Subsystem blocks that are not atomic are not translated to composite nodes, which means that the virtual subsystem hierarchy is flattened. Each atomic subsystem with a contract is also individually type-checked during translation. If unknown blocks are discovered during type-checking, the subsystem diagram will not be translated. However, a node is created for the subsystem block and the rest of the graph can still rely on the node behaviour described in the contract. This means we can partially handle models that contain constructs that the tool does not understand. This is important, since Simulink contains a large and expanding library of blocks, which can also be extended by different toolboxes.

21 The SDF to Statement translator

This component takes an SDF graph and translates it into a list of verification structures. Each verification structure contains the information needed for verification of a *CompositeNode*-object (Simulink atomic subsystem). A verification structure consists of the node contract, as well as a sequential statement that conforms to the sequential statement meta-model. The statement is obtained by translating the graph of the composite node as described in Section 7. However, the tool does not yet perform the translation of general SDF graphs using the algorithms described in [25, 24].

One problem that was only briefly mentioned in Section 7 is the implementation of the function v described in Definition 2. This function should provide a mapping of ports and delays to unique identifiers. The naming policy is given in (26).

$$\begin{array}{l} \mathsf{v}.n.p = \mathsf{uniqueName}.n\hat{}\,\text{"\_"}\hat{}\,\mathsf{name}.p \\ \mathsf{v}.d = \mathsf{uniqueName}.(\mathsf{container}.d)\hat{}\,\text{"\_"}\hat{}\,\mathsf{name}.d \end{array} \tag{26}$$

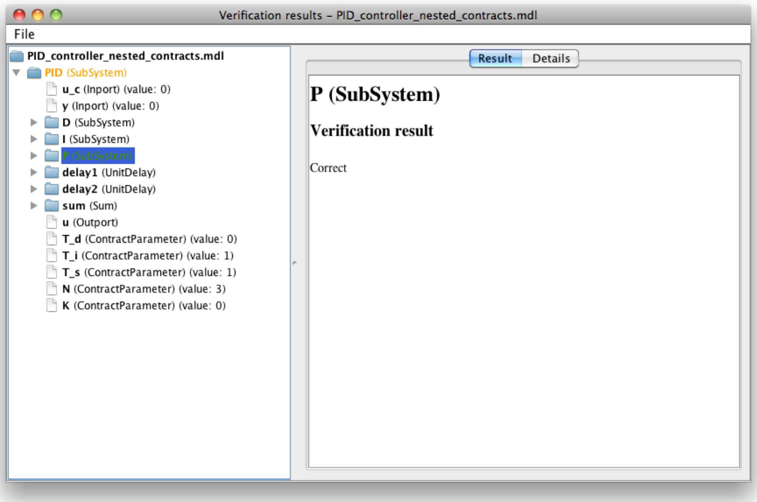
Where uniqueName is defined as:

$$\begin{array}{rl} \mathsf{uniqueName}.n = & \textbf{if } \mathsf{container}.n \neq \mathit{nil} \\ & \textbf{then } \mathsf{uniqueName}.(\mathsf{container}.n)\hat{}\,\text{"\_"}\hat{}\,\mathsf{name}.n \\ & \textbf{else } \mathsf{name}.n \textbf{ end} \end{array}$$

Here the function container.o gives the node that contains object o and name gives the name of an object. The symbol ^ denotes string concatenation. For example a port with name p in a node named n, which in turn resides in node m, would get the name m_n_p. Spaces and new line characters in node names are replaced by "__".

22 The weakest precondition calculator

The weakest precondition calculator directly implements the rules in (9) in Section 6. It takes a *Statement* object and an *Expression* object as argument. It then computes a new *Expression* object that represents the weakest precondition for the statement represented by the statement object to establish the post-condition given by the expression object given as argument.



```
Verification results – PID_controller_nested_contracts.mdl
File
PID_controller_nested_contracts.mdl
▼ PID (SubSystem)
      u_c (Inport) (value: 0)
      y (Inport) (value: 0)
    ▶ D (SubSystem)
    ▶ I (SubSystem)
    ▶ P (SubSystem)
    ▶ delay1 (UnitDelay)
    ▶ delay2 (UnitDelay)
    ▶ sum (Sum)
      u (Outport)
      T_d (ContractParameter) (value: 0)
      T_i (ContractParameter) (value: 1)
      T_s (ContractParameter) (value: 1)
      N (ContractParameter) (value: 3)
      K (ContractParameter) (value: 0)
```

P (SubSystem)

Verification result

Correct

Figure 12: A screen shot of the tool showing the verification results of the PID-controller example from Subsection 10.1

23 The verifier

The verifier takes a list of verification structures produced by the SDF to statement translator component and verifies their correctness. For each verification structure we show that the translated node graph (subsystem diagram) conforms to its contract description as described in Section 10. The proof obligation for refinement (24) is first calculated using (14) by utilising the weakest pre-condition calculator. Each resulting proof obligation is then translated to the SMT-LIB format [34]. The SMT-solver Z3 is then used to discharge the proof obligations. Any SMT-solver that can read the SMT-LIB format can in principle be used, but we have used Z3 because of its excellent support for both linear and non-linear real number arithmetic.

To visualise the result of the verification process, a graphical user interface is included. A screen shot from the tool is shown in Figure 12. The interface consists of a tree view that shows an overview of the structure of the Simulink model that is being verified, as well as a text area to display more detailed information. The result of the verification is shown by giving the subsystems in the tree view different colors. Green means that the subsystem was found correct, red means that the subsystem is incorrect and orange that the correctness is unknown. Counter examples are also shown in the tree by labeling the model elements with the values obtained from the counter example generated by verification tool. The text area then displays more detailed information if available.

24 Future work

There are many improvements and extensions that can be made both to the method and to the tool in order to make them even more useful for verification of Simulink models.

One of the key features of Simulink is the possibility to transparently and efficiently use matrices and vectors. This needs to be supported by our tool also. This would require a type checker that can also infer sizes of signals and expressions. Furthermore, an efficient translation of matrices to the SMT-LIB format is also needed.

Another limitation is that multi-rate models cannot be handled by the tool yet. However, the SDF graphs are also useful for modeling multi-rate systems and, hence, this type of systems can already be handled by that approach. To handle multi-rate models in the tool, inference of the clock of the blocks would be needed as the sampling time is only implicitly defined in Simulink. This has already been done in [38]. Furthermore, some work on defining appropriate contracts in this setting is also needed in order to get a useful method.

25 Conclusions

Contract-based development of Simulink models has earlier been determined to be useful [10, 12]. This paper presents one approach to do contract-based, compositional, automatic verification of Simulink models. The verification is based on representing Simulink diagrams as SDF graphs to obtain sequential program statements that can be analysed using traditional refinement-based methods. This gives a straightforward approach to calculate the proof obligations to determine if a given Simulink subsystem satisfies its contract. The method is compositional in the sense that the subsystems in a model can be verified individually. As a by-product, we also obtain a method for contract-based verification for any SDF-based notation. Furthermore, the approach has been implemented in a prototype tool.

There are several approaches to formal verification of Simulink models that are based on translating the diagrams to a formal notation [38, 16, 17], as well as *Simulink Design Verifier* [28]. The goal of those tools and methods is to verify different properties about Simulink models, not to provide a comprehensive framework for compositional verification. However, they can be used as verification back-ends for the contract-based approach described in this paper. We have decided to use our own tool as it allows us to generate the proof obligations in a format suitable for efficient automatic verification tools (the SMT-solver Z3). It also allows us to experiment with different approaches to verification. However, it would be interesting to investigate the use of e.g. Simulink Design Verifier as the verification back-end.

SDF graphs in conjunction with the theory of refinement seem to give a good basis for analysis of data flow diagrams. The classical theory of program analysis and refinement can be used and the generation of proof obligations, suitable for automatic proofs, needed for verification is straightforward. The developed tool support shows that the approach can be implemented, as well as enables the use of the approach on practical problems.

Acknowledgment

Work done within the EFFIMA program coordinated by Fimecc.

References

[1] R.-J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Proceedings of the 2nd ACM SIGACT-SIGOPS Sym-*

posium of Principles of Distributed Computing, pages 131–142, 1983.

[2] R.-J. R. Back and K. Sere. Stepwise refinement of action systems. *Structured Programming*, 12:17–30, 1991.

[3] R.-J. R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *LNCS*, pages 42–66. Springer-Verlag, 1989.

[4] R.-J. R. Back and J. von Wright. Trace refinement of action systems. In B. Jonsson and J. Parrow, editors, *Proc. of the 5th International Conference on Concurrency Theory, CONCUR'94*, pages 367–384, Uppsala, Sweden, 1994. Springer-Verlag.

[5] R.-J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer-Verlag, 1998.

[6] R.-J. R. Back and J. von Wright. Encoding, decoding and data refinement. *Formal Aspects of Computing*, 12:313–349, 2000.

[7] R.-J. R. Back and J. von Wright. Compositional action system refinement. *Formal Aspects of Computing*, 15:103–117, 2003.

[8] M. Barnett, R. DeLine, M. Fändrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[9] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceedings of Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2004. `http://research.microsoft.com/specsharp/`.

[10] P. Boström. *Formal design and verification of systems using domain-specific languages.* PhD thesis, Åbo Akademi University (TUCS), 2008.

[11] P. Boström, T. Huotari, R. Grönblom, and J. Wiik. A tool for contract-based verification of Simulink models. `http://users.abo.fi/pbostrom/slverificationtool/`, 2010.

[12] P. Boström, M. Linjama, L. Morel, L. Siivonen, and M. Waldén. Design and validation of digital controllers for hydraulics systems. In *The 10th Scandinavian International Conference on Fluid Power*, volume 1, pages 227–241, Tampere, Finland, 2007.

[13] P. Boström, L. Morel, and M. Waldén. Stepwise development of Simulink models using the refinement calculus framework. In *Theoretical Aspects of Computing (ICTAC2007)*, volume 4711 of *LNCS*, pages 79–93, Macao, China, 2007.

[14] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005. `http://www.cs.iastate.edu/~leavens/JML/`.

[15] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. Lustre, a declarative language for programming synchronous systems. In *14th ACM Conf. on Principles of Programming Languages*, Munich, Germany, 1987.

[16] A. L. C. Cavalcanti and P. Clayton. Verification of control systems using Circus. In *Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems*. IEEE Computer Society, 2006.

[17] C. Chen and J. S. Dong. Applying timed interval calculus to Simulink diagrams. In *Eight International Conference on Formal Engineering Methods, ICFEM 2006*, volume 4260 of *LNCS*, pages 74–93, Macao, China, 2006. Springer-Verlag.

[18] L. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS2008)*, volume 4963 of *LNCS*, pages 337–340, Budapest, Hungary, 2008. Springer.

[19] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[20] T. Farkas and D. Grund. Rule checking within the model-based development of safety-critical systems and embedded automotive software. In *Proceedings of the Eighth International Symposium on Autonomous Decentralized Systems (ISADS'07)*. IEEE Computer Society, 2007.

[21] T. Farkas, C. Hein, and T. Ritter. Automatic evaluation of modelling rules and design guidelines.

[22] Eclipse Foundation. Eclipse modeling framework (EMF). `http://www.eclipse.org/modeling/emf/`, 2010.

[23] T. Huotari. Parsing Simulink model-files and generating corresponding EMF models. Master's thesis, Åbo Akademi University, 2010. (To appear).

[24] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers*, C-36(1), 1987.

[25] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9), 1987.

[26] M. Linjama, M. Huova, P. Boström, A. Laamanen, L. Siivonen, L. Morel, M. Waldén, and M. Vilenius. Design and implementation of energy saving digital hydraulic control system. In *The 10th Scandinavian International Conference on Fluid Power*, volume 2, pages 341–359, Tampere, Finland, 2007.

[27] F. Maraninchi and L. Morel. Logical-time contracts for reactive embedded components. In *30th EUROMICRO Conference on Component-Based Software Engineering Track, ECBSE'04*, Rennes, France, August 2004.

[28] Mathworks Inc. Simulink. `http://www.mathworks.com/products/simulink`, 2010.

[29] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2 edition, 1997.

[30] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and system sciences*, 17:384–375, 1978.

[31] S. Neema, Z. Kalmar, F. Shi, A. Vizhanyo, and G. Karsai. A visually-specified code generator for Simulink/Stateflow. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE Computer Society, 2005.

[32] Object Management Group. Meta-object facility (MOF). `http://www.omg.org/mof/`, 2009.

[33] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[34] S. Ranise and C. Tinelli. The SMT-LIB standard: Version 1.2. `http://goedel.cs.uiowa.edu/smtlib/`, 2009.

[35] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addision-Wesley Professional, 2 edition, 2010.

[36] Terence Parr, et. al. ANTLR: ANother Tool for Language Recognition. `https://javacc.dev.java.net/`, 2010.

[37] The JavaCC project. JavaCC and JJTree. `https://javacc.dev.java.net/`, 2010.

[38] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):779–818, 2005.

[39] I. Stürmer, I. Kreuz, W Schäfer, and Andy Schürr. The MATE approach: Enhanced Simulink and Stateflow model transformation. In *Proc. of MathWorks Automotive Conference (MAC 2007)*, Dearborn (MI), USA, 2007.

[40] B. Wittenmark, K. J. Åström, and K.-E. Årzén. *Computer Control: an overview*. International Federation of Automatic Control (IFAC), 2002. IFAC professional brief: `http://www.ifac-control.org`.

[41] J. C. P. Woodcock and A. L. C. Cavalcanti. The semantics of Circus. In *Proceedings of ZB 2002*, volume 2272 of *LNCS*, pages 184–203. Springer-Verlag, 2002.

ISBN 978-952-12-2473-7
ISSN 1239-1891