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

Abstract. This paper presents an approach to compositional contract-based verification of Simulink models. The verification approach uses Synchronous Data Flow (SDF) graphs as a formalism to obtain sequential program statements that can then be analysed using traditional refinement-based verification techniques. Automatic generation of the proof obligations needed for verification of correctness with respect to contracts, as well as automatic proofs are also discussed.

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 [17].

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. Hence, there is a continuing need to better manage the complexity of models. Since control systems also often have high reliability requirements, there is also a need to analyse the models for correctness. One approach that we have explored to address the problems above is to use contracts to aid the decomposition of models into smaller parts with well defined interfaces and to aid the analysis of those parts and their interaction for correctness.

The aim of this paper is to propose a new compositional technique for verifying functional correctness of Simulink models with respect to 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 [18,11,6]. This suggests contracts could be useful for Simulink data flow diagrams also. Furthermore, the interaction between components in

* Work done in the EFFIMA program coordinated by Fimecc.

S. Qin and Z. Qiu (Eds.): ICFEM 2011, LNCS 6991, pp. 291–306, 2011.

Simulink data flow diagrams is simpler than between objects in object-oriented systems, which means that automated formal verification can potentially be easier to do.

We have earlier developed contracts and contract-based verification methods for Simulink models [9,10,7]. Here we give more expressive contracts, similar in expressiveness to the contracts for reactive components in [16]. In addition features in [9,10,7], the contracts here can model internal state of components and relate it to the concrete state used in the Simulink diagrams. A new compositional method to verify correctness of Simulink models with respect to these contracts is also given. The formal analysis methods for Simulink models with contracts are based on translating the models to functionally equivalent sequential statements that can be analysed by traditional, refinement-based, techniques [1,3,4]. To obtain the sequential program statements, Simulink diagrams are viewed as synchronous data flow (SDF) graphs [15,14]. The benefit of using SDF graphs compared to the more ad-hoc approach in [9,10,7] is that the mapping of these data flow graphs to the sequential programs used in the analysis has been thoroughly studied. The approach is supported by a tool [8] that can automatically verify that Simulink models satisfy their contracts. Contract-based design in Simulink has been applied to larger examples [9,7]. The contracts were found to be useful both when structuring the system and for verification.

The paper starts with an overview of Simulink, 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 discussion of translation 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, as well as tool support. To illustrate the approach, contract-based verification is used on a small example.

2 Simulink

The language used to create models in Simulink is based on hierarchical data flow diagrams [17]. 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 can be parameterised with parameters that are set before model execution and remain constant during the execution. Blocks can also contain memory. Hence, their behaviour does not only depend on the current values on the in-ports and the parameter values, but also on previous in-port values.

Here only discrete Simulink models with one rate are considered. This means that a model is evaluated periodically with a given sampling rate. 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.

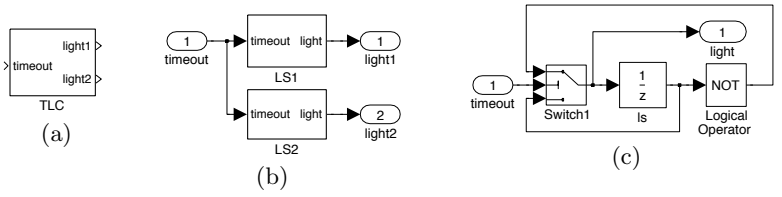

Fig. 1. (a) A subsystem that contains a simple traffic light controller, (b) its contents consisting of two individual light controllers and (c) the individual light controllers

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 [17]. The behaviour of the block is given by the difference equation 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 updates the out-ports y at sample k and g the function that updates the state x. Consider, e.g., the *Logical Operator*-block and the *Unit Delay*-block (marked by $1/z$) in Fig. 1 (c). In this case the *Logical Operator*-block negates the input, while a *Unit Delay*-block delays the input with one sampling time. The behaviour of the *Logical Operator*-block is then given by the equation $y.k = \neg u.k$. Note that this block has no internal state. A *Unit Delay*-block then has the behaviour given as $y.k = x.k \wedge x.(k+1) = u.k$. Information about other blocks can be found in the Simulink documentation [17]. The diagrams can also be hierarchically structured using the notion of subsystem blocks, which are blocks that themselves contain diagrams.

To illustrate the use of Simulink, a small example that consists of a controller for a simplified traffic light system is given. The system consists of two lights that can be either *green* (true) or *red* (false). However, both lights should not be green at the same time. When a timeout signal has the value true, the lights change. The subsystem block *TLC* in Fig. 1 (a) contains the traffic light controller. A new light configuration is computed separately for each light by the subsystems *LS1* and *LS2* (Fig. 1 (b)) at each sampling instant. Both lights are switched in case *timeout* is true otherwise they retain their values (Fig. 1 (c)).

3 Contracts in Simulink

Simulink diagrams for advanced control systems can contain thousands of blocks. For example, in the system discussed in [9], the controller contains more than 4000 blocks. To effectively manage the complexity of such large models, there is a need to better make explicit the division of responsibility between subsystems. To make verification scalable, it is also useful to reason about the interaction between subsystems at a higher level of abstraction than their detailed content, which often consist of deep hierarchies of diagrams containing hundreds of blocks.

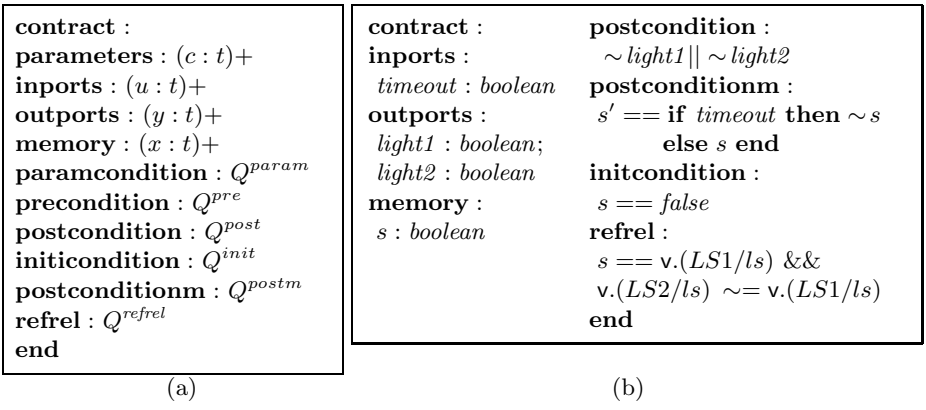

Fig. 2. (a) The abstract syntax of contracts and (b) an example contract that describes the traffic light controller subsystem

Our proposed solution to the problems above is to use contracts to describe subsystems. This enables verifying subsystem hierarchies one layer at the time, where each layer relies on the contract descriptions of the subsystems in the layer below. The contracts are mainly intended for expressing properties of control logic. System level properties such as, e.g. stability and performance, are best handled by other means.

An (atomic) subsystem can essentially be considered to be 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 behaviour. Our proposed contracts have the abstract syntax given in Fig. 2 (a). There c, u, y and x are identifiers, t is a type in the set $\{\mathsf{double}, \mathsf{int}, \mathsf{boolean}\}$, $z+$ denotes one or more occurrences of z and Q denotes a predicate. The contract first declares the parameters, in- and out-ports of the subsystem, as well as internal state (specification) variables. These are all given as lists of identifier-type pairs. The behaviour of the subsystem is described by a set of conditions. Here Q^{param} describes the block parameters used in the subsystem, Q^{init} describes the initial values of the variables x, Q^{pre} is the precondition, Q^{post} is the postcondition constricting the out-ports and Q^{postm} the postcondition constricting the new values x' of x. The specification variables in the contracts give an abstract view of the block memories inside the subsystem. The block memories in turn represent the concrete state of the Simulink model. The condition Q^{refrel} is then used to describe how the specification variables relate to block memories. In order to refer to block memories in the internal diagram, we use a naming scheme based on block naming policy in Simulink [8]. The contracts here have a similar structure and describe the same type of behaviour as the ones for reactive components in [16].

To give an idea of how contracts can be used, a contract describing the functionality of the traffic light system from Section 2 is given in Fig. 2 (b). A specification variable s is used to model the state of the first light. The

initialisation of this light is here assumed to be red (false). The postconditions then encode the desired behaviour of the controller. Both lights should not be green (true) at the same time. Note that for brevity the postcondition does not consider that the output depends on s. The refinement relation then describes how the memory in the *Unit delay*-blocks in the subsystems *LS1* and *LS2* relate to s. Here a function v is used to map block memories to variable identifiers. This mapping is discussed more in Section 6. The concrete syntax used in the contract conditions is inspired by the syntax of Matlab expressions [8].

4 Synchronous Data Flow Graphs

The goal is to verify functional correctness of Simulink models with respect to contracts. Program analysis for sequential programs have been studied extensively, e.g., [3,4]. To reuse this work, we translate the Simulink diagrams to functionally equivalent sequential programs. Furthermore, this allows us to also handle imperative constructs from Matlab, which are often used in conjunction with Simulink. To obtain such sequential programs from Simulink diagrams, we represent the diagrams as synchronous data flow (SDF) graphs [15,14] since compilation of such graphs to sequential or parallel code has been studied extensively.

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 [15,14] 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 fixed sampling rate. Furthermore, the SDF programs are never supposed to terminate.

We use a similar notation as in [15,14] to describe our synchronous data flow graphs. An example is given in Fig. 3. The program computes the (exponential) moving average v 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, $D.v.k = v.(k - 1)$.

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 [15]. The number n on an edge adjacent to the source node denotes that the node will output n pieces of data, while the number m near the destination node denotes that the block will read m pieces of data when it fires. This gives a convenient way to also handle multi-rate data flow networks. Since we only consider single-rate graphs here, n and m are always 1. The D on an edge denotes that the edge delays the data by one sampling time. Each delay also has an identifier, here d.

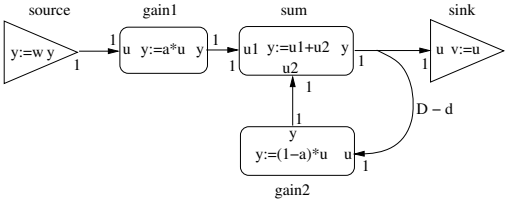

Fig. 3. Example of a simple SDF program

The nodes in the SDF graph can be statically scheduled to obtain sequential or parallel programs [15]. Here we will only present the algorithm [15] 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 change during the execution of the graph. As an example, consider the graph G in Fig. 3. We first number the nodes using a function n_n and edges using n_e according to:

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

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

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \end{bmatrix} \tag{2}$$

The node n run at step k is specified with a vector $v.k$ that contains 1 in the position given by the number $n_n.n$ 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:

$$b.(k+1) = b.k + \Gamma v.k \tag{3}$$

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. For the graph G, the initial state is given by $b.0 = \begin{bmatrix} 0\ 0\ 0\ 1\ 0 \end{bmatrix}^T$. 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.

$$b.0 = b.0 + \Gamma q \tag{4}$$

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

In case the graph is schedulable, the least [15], 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 vector 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 (3) 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 $b_i.(k+1)$ in (3) negative.
4. If each node α is scheduled q_α times, then stop
5. If no node in S can be scheduled, return error else go to step 3.

5 The Sequential Language

The computation inside nodes is 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 a language designed for this purpose is therefore used [3].

Since the analysis methods are based on the refinement calculus [3], a short introduction is needed. Each program statement is a predicate transformer from predicates on the output state space to predicates on the input state space. A predicate transformer S applied to a predicate q gives the weakest predicate describing the states from where S is guaranteed to establish q. The syntax of the statement language is given as:

$$\begin{array}{lllll} S ::= & x := E & \text{(Assignment)} & \mid & x : \mid P \quad \text{(Non} - \text{deterministic assignment)} \mid \\ & [g] & \text{(Assumption)} & \mid & S_1 ; S_2 \quad \text{(Sequential composition)} \quad \mid \\ & \{g\} & \text{(Assertion)} & \mid & S_1 \sqcap S_2 \quad \text{(Non} - \text{deterministic choice)} \end{array} \tag{5}$$

Here x is a list of variables, E a list of expressions, while g and P are predicates. For an arbitrary postcondition q we have that:

$$\begin{array}{llll} (x := E).q = q[x/E] & & x : \mid P(x, x').q = \forall x' \cdot P(x, x') \Rightarrow q[x/x'] \\ [g]\,.q \quad = g \Rightarrow q & & (S_1 ; S_2).q \quad = S_1.(S_2.q) \\ \{g\}.q \quad = g \wedge q & & (S_1 \sqcap S_2).q \quad = S_1.q \wedge S_2.q \end{array} \tag{6}$$

Each statement is thus 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 $S.true$, which is referred to as the termination guard of S, $\mathtt{t}.S \mathrel{\hat{=}} S.true$. In states where $S.true$ does not hold the statement is said to abort. A statement S is said to behave miraculously, if executed in a state where $S.false$ 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, $\mathtt{g}.S \mathrel{\hat{=}} \neg S.false$. All statements in (5) are monotonic [3]. A statement S is monotonic, if it preserves the ordering given by implication: $S.q \Rightarrow S.p,$ if $q \Rightarrow p$.

A refinement relation $\sqsubseteq$ can be defined for the predicate transformers: $S \sqsubseteq R \mathrel{\hat{=}} \forall q \cdot S.q \Rightarrow R.q$. This relation 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 [3]. 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 [1,4]. Data refinement of S by R under decoding Δ, $S \sqsubseteq_\Delta R$, is defined as: $S \sqsubseteq_\Delta R \mathrel{\hat{=}} \Delta; S \sqsubseteq R; \Delta$. The decoding Δ is normally assumed to have the form $\Delta \mathrel{\hat{=}} \{+a - c|Q\}$ [1], 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 [1]. An angelic relational assignment statement has the semantics: $\{+a - c|Q\}.q = \exists a' \cdot Q[a/a'] \wedge q[a/a']$ (see [1,4]).

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, $S = \{g\}; a, z : |P$. Using $\Delta = \{+a - c|Q\}$, rule (7) can be used to prove $S \sqsubseteq_\Delta R$, see [1].

$$Q \wedge g \wedge z, a = z_0, a_0 \Rightarrow R.(\exists a' \cdot Q[a/a'] \wedge P[a, a', z, z'/a_0, a', z_0, z]) \qquad (7)$$

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 $a, z : |P$.

Simulink is used to develop control systems, where the interaction of programs with their environment rather than their input-output behaviour is important. Hence, we are here interested in *reactive systems*. Consider two systems constructed from iteration of statements S and S', $init; \mathbf{do}\ S\ \mathbf{od}$ and $init'; \mathbf{do}\ S'\ \mathbf{do}$. The behaviour of the systems can then be defined by the traces of the observable states generated during execution [2]. Data refinement can be used to show *trace refinement* [2], $init; \mathbf{do}\ S\ \mathbf{od} \sqsubseteq_{tr} init'; \mathbf{do}\ S'\ \mathbf{do}$, between the two systems if they have the same observable state. Hence, that all traces generated by the concrete system can also be generated by the abstract system. Assume we have a decoding statement Δ that states how the unobservable state of the two systems relate. It is then sufficient to prove [2]: $\Delta; init \sqsubseteq init'; \Delta$ and $\Delta; S \sqsubseteq S'; \Delta$ if S' is strict, $\mathtt{g}.S = true$. This provides a mechanism to prove correctness of the system over all executions by only analysing the iterated statements. Note also that the decoding statement can be used to provide essentially a loop invariant on the observable and concrete state, see (7).

6 Translation of SDF Graphs

An SDF graph can be translated to a functionally equivalent sequential program by utilising the scheduling in Section 4. Here we will only consider single-rate

Simulink models. Hence, in the systems we consider all data-rates are one and there is also at most one delay on each edge. First we need to introduce the buffers needed for the communication between the nodes. In principle the communication is handled through FIFO-buffers [14]. However, to make the proof obligations simpler, we would like to have static buffers (shared variables). Due to the restrictions on delays and data rates, static buffering is straightforward to implement. All ports and delays are first translated as variables.

Definition 1. *Let the function* v *be an 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 (5).

Definition 2. *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 the algorithm in Section 4.*
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*.*

Since we only consider a special case in this paper, the data is handled as if FIFO-buffers were used. If there is no delay on an edge, then the required buffer size is one, since for each data element produced on the edge one will be consumed. The variable obtained from the out-port then 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. Fig. 4 illustrates this situation.

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

$$\begin{aligned}\mathsf{trans}.G \mathrel{\hat{=}}\ & \mathsf{v}.source.y := \mathsf{v}.G.w;\\ & \mathsf{v}.gain1.y := a * \mathsf{v}.source.y;\\ & \mathsf{v}.gain2.y := (1-a) * \mathsf{v}.d;\\ & \mathsf{v}.sum.y := \mathsf{v}.gain1.y + \mathsf{v}.gain2.y;\\ & \mathsf{v}.d := \mathsf{v}.sum.y;\\ & \mathsf{v}.G.v := \mathsf{v}.sum.y\end{aligned} \tag{8}$$

The statements are obtained from the nodes and scheduled according to Definition 2. Here we assume that w and v in the in and out nodes are ports of a node

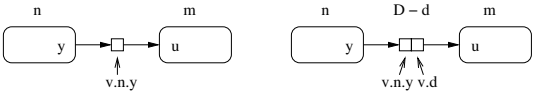

Fig. 4. (a) The buffer of an edge from n to m without a delay and (b) the buffer of an edge with one delay d

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.

We can now give a semantics to complete SDF graphs [8], i.e., graphs with no unconnected inputs. The semantics of a complete SDF graph G is here given by the traces of observable behaviour of the system obtained from the minimal PASS, $init$; **do** trans.G **od** . Hence, we can observe the state between repetitions of shortest repeating program statement. This semantics has been chosen to match the semantics of discrete single-rate Simulink, where at each sampling instant the entire model is evaluated.

6.1 Correctness of the Translation

A minimal PASS obtained with the algorithm in Section 4 is not necessarily unique. In order for the translation from SDF graph to sequential statement to be correct, all minimal PASS for the same graph should yield functionally equivalent statements. Different schedules can only be created during scheduling if several nodes are runnable at the same time, i.e., the nodes are independent. Changing the order in which the nodes are chosen then corresponds to swapping the nodes in the resulting schedule. We can thus generate all possible minimal schedules by repeated pairwise swapping of independent nodes. In order to transform a minimal PASS into any other, we then have to show that for any two statements S_1 and S_2 obtained from two independent nodes, $S_1; S_2 = S_2; S_1$. This does not hold in general even though statements from independent nodes use disjoint sets of variables. Consider for example $S_1 = \{false\}$ and $S_2 = [false]$. However, we have that $\{\mathsf{g}.(S_1; S_2)\}; S_1; S_2 = \{\mathsf{g}.(S_2; S_1)\}; S_2; S_1$. Thus for two statements T_1 and T_2 obtained from two different PASS for the same SDF graph we have: $\{\mathsf{g}.T_1\}; T_1 = \{\mathsf{g}.T_2\}; T_2$. Note that when we have a deterministic program T then it is non-miraculous [3], i.e., $\mathsf{g}.T = true$

7 SDF Graph Representation of Simulink Models

To give a semantics to Simulink models, they are mapped to SDF graphs. 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 node in the SDF notation. In this section we present how to map the most fundamental blocks to their corresponding SDF representation.

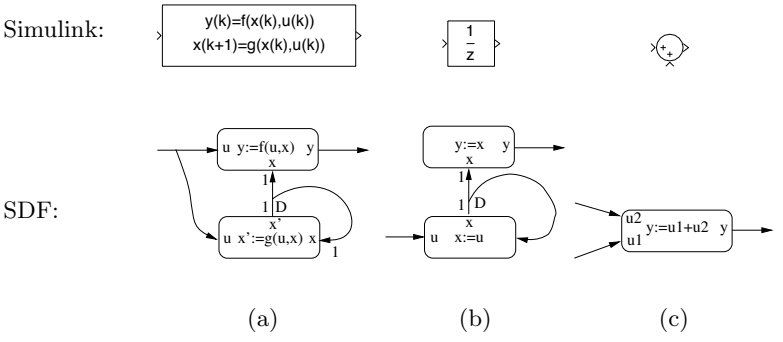

Fig. 5. The SDF representation of (a) a general Simulink block, (b) a *Unit delay*-block and (c) a *Sum*-block

7.1 Mapping Simulink Blocks to Nodes

We can differentiate between the following important Simulink blocks: *Functional blocks*, *In and out blocks* and *Subsystem blocks*.

Functional blocks. These blocks in the Simulink library directly encapsulates a difference equation. Consider again a Simulink block with the general form in (1). The implementation of the block as an SDF graph is shown in Fig. 5 (a). The behaviour of the block is described by two equations, which are not necessarily executed together. All Simulink functional blocks are then special cases of this general pattern: consider, e.g., the *Unit delay*-block and the *Sum*-block shown with their SDF representations in Fig. 5 (b) and (c), respectively. Note that we here only consider Simulink blocks that do not have side effects.

In and out blocks. These blocks are used to obtain inputs from in-ports of the containing subsystem, as well as export values to the out-ports. In and out blocks correspond to in and out nodes in the SDF graphs.

Subsystem blocks. Subsystem blocks that are used for structuring Simulink diagrams. 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. This flattening might lead to scalability problems in the verification, and atomic subsystems should be preferred instead. The atomic subsystems 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 Fig. 6. Its SDF representation (denoted $\mathsf{sdf}.S$) is obtained by recursively translating its content.

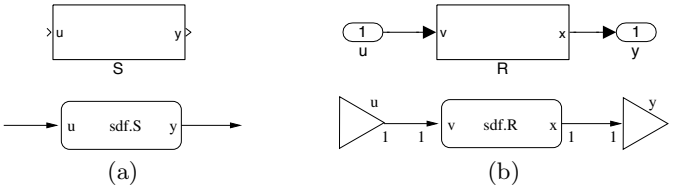

Fig. 6. (a) A Simulink atomic subsystem S and the corresponding SDF node and (b) the contents of S and its corresponding SDF representation

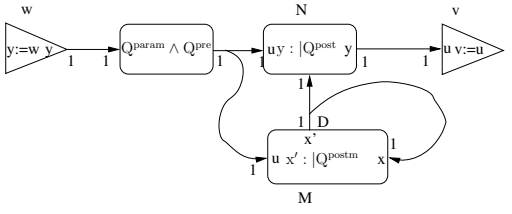

Fig. 7. SDF graph obtained from the contract specification of an atomic subsystem

7.2 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 (atomic) subsystems as abstractions of the subsystem behaviours 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 $\{Q^{pre}\}; x : |Q^{post}$, see [3].

Assume we have subsystem S in Fig. 6 (a), which is described by the contract C in Fig. 2 (a). We then get the SDF graph representation, $\mathsf{sdf}.C$, shown in Fig. 7 for the contract. This is the most abstract description of S that can be used when analysing models where the subsystem is used. Note that this is very similar to the translation of the general Simulink block in Fig. 5 (a). The reason is that the contract gives an abstract description of the same type of behaviour.

A functionally equivalent sequential program statement $\mathsf{trans}.(\mathsf{sdf}.C)$ can now be obtained. This is again done using the translation procedure in Definition 2.

$$\begin{aligned}\mathsf{trans}.(\mathsf{sdf}.C)] \mathrel{\hat{=}}\ & \mathsf{v}.w.y := \mathsf{v}.S.w; \{Q^{param}[u/\mathsf{v}.w.y] \wedge Q^{pre}[u/\mathsf{v}.w.y]\}; \\ & \mathsf{v}.N.y : |Q^{post}[x, u, y/\mathsf{v}.d, \mathsf{v}.w.y, \mathsf{v}.N.y]; \mathsf{v}.S.v := \mathsf{v}.N.y; \\ & \mathsf{v}.M.x' : |Q^{postm}[x, u, x'/\mathsf{v}.d, \mathsf{v}.w.y, \mathsf{v}.M.x'']; \mathsf{v}.d := \mathsf{v}.M.x'\end{aligned} \tag{9}$$

As in (8), references to the inports are directly substituted by the variable obtained from the connected outport or delay. Note that again the variables w and v in the in and out nodes are considered ports in the containing subsystem S.

8 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 (Fig. 7) preserves the behaviour, i.e. refines, the complete system. Assume we have a Simulink model $\mathcal{M}$ containing an atomic subsystem M with contract C. The semantics of the Simulink model is given by the translation to sequential statements. The abstract statement obtained from the model $\mathcal{M}$ where M is used can be written as $\mathsf{trans.}(\mathsf{sdf}.\mathcal{M}) \mathrel{\hat{=}} S_1; \mathsf{trans.}(\mathsf{sdf}.C)[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2$. The concrete statement is then given as $\mathsf{trans.}(\mathsf{sdf}.\mathcal{M}') \mathrel{\hat{=}} S_1; \mathsf{trans.}(\mathsf{sdf}.M)[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2$. In the complete translation all connected in-ports of M are replaced by the port or block memory they are connected to (see Section 6). 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. According to Section 5, in order to prove trace refinement $init; \mathbf{do}\ \mathsf{trans.}(\mathsf{sdf}.\mathcal{M})\ \mathbf{od} \sqsubseteq_{tr} init'; \mathbf{do}\ \mathsf{trans.}(\mathsf{sdf}.\mathcal{M}')\ \mathbf{od}$ it is sufficient to prove data refinement of the initialisation and the statement inside the loop. The observable state is considered to be all variables except the ones internal to subsystem M and contract C. For the statement we thus need to prove:

$$\begin{array}{l} \Delta; S_1; \mathsf{trans.}(\mathsf{sdf}.C)[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2 \\ \sqsubseteq S_1; \mathsf{trans.}(\mathsf{sdf}.M)[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2; \Delta \end{array} \tag{10}$$

Since the refinement only concerns the internals of M, the decoding Δ refers only to the internal variables of $\mathsf{trans.}(\mathsf{sdf}.C)$ and $\mathsf{trans.}(\mathsf{sdf}.M)$. Here it has the form $\Delta \mathrel{\hat{=}} \{-\mathsf{v}.b_n.p_n, \mathsf{v}.d_n + \mathsf{v}.x, \mathsf{v}.b_o.p_o | Q^{refrel}\}$, where p_n denotes the new out-ports, p_o denotes old 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 Fig. 2 (b)) 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 (10) we then need to show that:

$$\Delta; \mathsf{trans.}(\mathsf{sdf}.C) \sqsubseteq \mathsf{trans.}(\mathsf{sdf}.M); \Delta \tag{11}$$

Proof.

$$\begin{array}{ll} & \Delta; S_1; \mathsf{trans.}(\mathsf{sdf}.C)[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2 \\ \sqsubseteq & \{\text{Assumption above}\} \\ & S_1; \Delta; \mathsf{trans.}(\mathsf{sdf}.C)[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2 \\ = & \{\mathsf{v}.M.p_i, \mathsf{v}.(\mathsf{conn}.p_i) \text{ not free in } \Delta\} \\ & S_1; (\Delta; \mathsf{trans.}(\mathsf{sdf}.C))[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2 \\ \sqsubseteq & \{\text{Assumption (11) and } \mathsf{v}.(\mathsf{conn}.p_i) \text{ not free in } \mathsf{trans.}(\mathsf{sdf}.M)\} \\ & S_1; (\mathsf{trans.}(\mathsf{sdf}.M); \Delta)[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2 \\ = & \{\mathsf{v.M.p_i}, \mathsf{v}.(\mathsf{conn.p_i}) \text{ not free in } \Delta\} \\ & S_1; \mathsf{trans.}(\mathsf{sdf}.M); [\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; \Delta; S_2 \\ \sqsubseteq & \{\text{Assumption above}\} \\ & S_1; \mathsf{trans.}(\mathsf{sdf}.M)[\mathsf{v}.M.p_i/\mathsf{v}.(\mathsf{conn}.p_i)]; S_2; \Delta \end{array}$$

□

Note also that if all subsystems are implemented as deterministic diagrams, then the corresponding statements do not behave miraculously [3]. The SDF graph obtained from the Simulink model is thus non-terminating, which is the requirement for correct translation stated in Subsection 6.1.

8.1 Tool Support

Prototype tool support for this approach has been developed [8]. The tool takes a Simulink model annotated by contracts written down as text in the *Description*-field of the subsystems as argument. The tool then automatically checks that each atomic subsystem (with a contract) satisfies its contract using the approach described in this paper. Currently the tool supports virtual, atomic and enabled subsystems, a wide variety of mathematical and logical blocks, delay and memory blocks, as well as switch blocks. However, this list of handled Simulink constructs is expanding. To prove (11), the final proof obligation is after simplifcations generated using formula (7). To increase scalability when verifying that a subsystem conforms to its contract, the verification tool uses the abstractions given by the contract descriptions of the subsystems at lower levels in the subsystem hierarchy as discussed earlier. We have used the SMT solver Z3 [13] to automate the proofs. The constructs that are supported (e.g. the types of arithmetic) and the scalability of the verification is thus largely dependent on this tool.

8.2 Example of Subsystem Refinement

To give an example of the translation of Simulink models and the analysis methods, the simple traffic light controller from Section 2 is used. The subsystem, *TLC*, implementing the controller is shown in Fig. 1 (a). The contract C associated with the subsystem is given in Fig. 2 (b). The contract specification of the subsystem is translated to a sequential program statement as described in (9):

$$
\begin{array}{l}
\mathsf{trans.}(\mathsf{sdf}.C) \mathrel{\hat{=}} \\
\quad \mathsf{v}.Timeout.y := \mathsf{v}.TLC.timeout; \\
\quad \mathsf{v}.N.light1, \mathsf{v}.N.light2 : |\neg \mathsf{v}.N.light1' \vee \neg \mathsf{v}.N.light2'; \\
\quad \mathsf{v}.TLC.light1 := \mathsf{v}.N.light1; \mathsf{v}.TLC.light2 := \mathsf{v}.N.light2; \\
\quad \mathsf{v}.M.s' : |\mathsf{v}.M.s'' = \textbf{if } \mathsf{v}.Timeout.y \textbf{ then } \neg \mathsf{v}.s \textbf{ else } \mathsf{v}.s \textbf{ end}; \mathsf{v}.s := \mathsf{v}.M.s'
\end{array}
$$

The statement above should then be refined by the translation of the diagram inside the subsystem *TLC*, which is shown in Fig. 1 (b). One possible translation of the diagram is then given as:

$$
\begin{array}{rl}
\mathsf{trans.}(\mathsf{sdf}.TLC) \mathrel{\hat{=}} & \mathsf{trans.}(\mathsf{sdf}.LS1)[\mathsf{v}.LS1.timeout/\mathsf{v}.TLC.timeout]; \\
& \mathsf{trans.}(\mathsf{sdf}.LS2)[\mathsf{v}.LS2.timeout/\mathsf{v}.TLC.timeout]; \\
& \mathsf{v}.TLC.light1 := \mathsf{v}.LS1.light; \\
& \mathsf{v}.TLC.light2 := \mathsf{v}.LS2.light;
\end{array}
$$

The translation proceeds recursively through subsystems *LS1* and *LS2*. In case they would have contracts, their contract description would be used in the translation. The block memories from the unit delay blocks in *LS1* and *LS2* relate to the specification variable s as described by Q^{refrel} in Fig. 2 (b). The refinement rule (11) for subsystem refinement leads to the condition:

$$\{-\mathsf{v}.(LS1/ls), \mathsf{v}.(LS2/ls), \ldots + \mathsf{v}.s, \ldots | Q^{refrel}\}; \mathsf{trans}.(\mathsf{sdf}.C)$$
$$\sqsubseteq \mathsf{trans}.(\mathsf{sdf}.TLC); \{-\mathsf{v}.(LS1/ls), \mathsf{v}.(LS2/ls), \ldots + \mathsf{v}.s, \ldots | Q^{refrel}\}$$

The tool we have developed [8] has been used to verify this refinement. Whenever subsystem TLC is used in a model we can now use the simpler contract description when analysing the rest of the model. Since we have property (10) and we proved property (11) above, the behaviour of the complete model when the internal diagram of the subsystem is used will refine the behaviour of the model when contract description is used.

9 Conclusions

This paper presents one approach to automatically verify that Simulink models satisfy contracts stating functional properties. The method is based on representing Simulink diagrams as SDF graphs to obtain a functionally equivalent sequential program statements that can be analysed using traditional refinement-based methods. This gives an approach to compositionally verify large models. As a by-product, we also obtain a method for contract-based verification for any SDF-based notation. The approach has also been implemented in a tool [8].

Other formalisations of Simulink supported by verification tools exist in Lustre [19] and Circus [12]. However, these approaches do not consider compositional, contract-based, verification. Contracts could be analysed in those frameworks also, but our approach gives a convenient way to separately reason about both pre- and post-conditions, as well as refinement. Our method can also easily handle the imperative constructs from Matlab that are often used in conjunction with Simulink, which would problematic in Lustre. The tool with the goals closest to ours is *Simulink Design Verifier* (SLDV) [17]. This tool can verify that discrete Simulink models satisfy properties given as special blocks in the diagrams. However, it does not provide a method to systematically build correctness arguments for large models as we do with contracts. SLDV verifies that from a given initial state a state violating the given properties cannot be reached, while our approach is an inductive argument stating that if we start from a state satisfying the refinement relation the model will again end up in such a state and behave according to the contract description. Furthermore, SDLV cannot handle non-linear arithmetic, which Z3 can handle to some degree. This makes it limited for verification of complex properties involving arithmetic. Its main focus is perhaps also more on verifying control logic that involves Stateflow [17].

The work can extended in several directions. Multi-rate systems and more of the Simulink modelling language should be considered. SDF graphs already support multi-rate systems. However, the SDF multi-rate notion does not directly correspond to the one in Simulink. Boogie [5] should also be investigated as a tool for automatic verification of the sequential statements obtained by our translation process, since it is already a very mature tool for this purpose. As a conclusion, SDF graphs in conjunction with the theory of refinement seem to give a good basis for contract-based verification of Simulink models, since mature automatic verification tools and techniques can be used.

References

1. Back, R.-J.R., von Wright, J.: Refinement calculus, part I: Sequential nondeterministic programs. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1989. LNCS, vol. 430, pp. 42–66. Springer, Heidelberg (1990)
2. Back, R.-J.R., von Wright, J.: Trace refinement of action systems. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 367–384. Springer, Heidelberg (1994)
3. Back, R.-J.R., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer, Heidelberg (1998)
4. Back, R.-J.R., von Wright, J.: Encoding, decoding and data refinement. Formal Aspects of Computing 12, 313–349 (2000)
5. Barnett, M., Chang, B.Y.E., Deline, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
6. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: The Spec# experience. Communications of the ACM 54(6) (2011)
7. Boström, P.: Formal design and verification of systems using domain-specific languages. Ph.D. thesis, Åbo Akademi University (TUCS) (2008)
8. Boström, P., Grönblom, R., Huotari, T., Wiik, J.: An approach to contract-based verification of Simulink models. Tech. Rep. 985, TUCS (2010)
9. Boström, P., Linjama, M., Morel, L., Siivonen, L., Waldén, M.: Design and validation of digital controllers for hydraulics systems. In: The 10th Scandinavian International Conference on Fluid Power, pp. 227–241 (2007)
10. Boström, P., Morel, L., Waldén, M.: Stepwise Development of Simulink Models Using the Refinement Calculus Framework. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 79–93. Springer, Heidelberg (2007)
11. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer 7(3), 212–232 (2005)
12. Cavalcanti, A., Clayton, P., O'Halloran, C.: Control law diagrams in circus. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 253–268. Springer, Heidelberg (2005)
13. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
14. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. on Computers C-36(1) (1987)
15. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proceedings of the IEEE 75(9) (1987)
16. Maraninchi, F., Morel, L.: Logical-time contracts for reactive embedded components. In: EUROMICRO 2004. IEEE Computer Society, Los Alamitos (2004)
17. Mathworks Inc.: Simulink (2010), `http://www.mathworks.com`
18. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
19. Tripakis, S., Sofronis, C., Caspi, P., Curic, A.: Translating discrete-time Simulink to Lustre. ACM Trans. on Embedded Computing Systems 4(4), 779–818 (2005)