Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Compositional Translation of Simulink Models into Synchronous BIP

Vassiliki Sfyrla, Georgios Tsiligiannis, Iris Safaka, Marius Bozga and Joseph Sifakis

Verimag Research Report n^{o} TR-2010-16

June 2010

Compositional Translation of Simulink Models into Synchronous BIP

Vassiliki Sfyrla, Georgios Tsiligiannis, Iris Safaka, Marius Bozga and Joseph Sifakis

June 2010

Abstract

We present a method for the translation of a discrete-time fragment of Simulink into the synchronous subset of the BIP language. The translation is fully compositional, that is, it preserves completely the original structure and reveals the minimal control coordination structure needed to perform the correct computation within Simulink models. Additionally, this translation can be seen as providing an alternative operational semantics of Simulink models using BIP. The advantages are twofold. It allows for integration of Simulink models within heterogeneous BIP designs. It enables the use of validation and automatic implementation techniques already available for BIP on Simulink models. The translation is currently implemented in the Simulink2BIP tool. We report several experiments, in particular, we show that the executable code generated from BIP models has comparable runtime performances as the code produced by the Real-Time Workshop on several MATLAB models.

Keywords: Synchronous BIP, simulink

Reviewers: Marius Bozga

How to cite this report:

@techreport { ,
title = { Compositional Translation
of Simulink Models into Synchronous BIP},
authors = { Vassiliki Sfyrla, Georgios Tsiligiannis, Iris Safaka, Marius Bozga and Joseph Sifakis},
institution = { Verimag Research Report },
number = {TR-2010-16},
year = { },
note = { }
}

1 Introduction

Simulink [1] is a very popular commercial tool for model-based design and simulation of dynamic embedded systems. Simulink systems are represented graphically using blocks and communication links between the blocks. Simulink is widely used by engineers since it provides a wide variety of block libraries for implementing and testing discrete and continuous systems occurring in many application domains.

Simulink lacks many desirable features of programming languages. In particular, the Simulink semantics is provided only informally, and moreover, it is only partially documented. Also, the meaning of models depend significantly on many simulation parameters (e.g. simulation step, solver used, etc).

BIP [2] – Behavior, Interaction, Priority – is a component-based formalism for modeling, analysis and implementation of heterogeneous real-time systems. It allows the description of systems as the composition of generic atomic components characterized by their behavior (i.e., extended Petri nets) and their interface (i.e., a set of ports). In contrast to many other existing frameworks, BIP has formal semantics and is expressive enough to model directly any coordination mechanism between components using uniformly interactions and/or priorities [3]. It has been successfully used to model complex systems including mixed hardware/software systems and complex software applications [4, 5].

Synchronous BIP is a subset of the BIP framework for modeling synchronous data-flow systems[6]. The behavior of synchronous BIP components is described by *modal flow graphs*, that are structures expressing dependency relations between events (actions) occurring in the same synchronous step. There are three different modalities characterizing dependencies between events: *strong*, *weak* and *conditional*. These dependencies allow to represent easily all the coordination constraints needed for the correct execution of synchronous models. Moreover, for a syntactic subclass of modal flow graphs, deadlock-freedom and confluence can be decided at low cost.

In this paper we provide a translation for the discrete-time fragment of Simulink into synchronous BIP. The translation is subject to some restrictions. Globally, we consider only Simulink models that have explicitly specified sample time and which can be simulated using fixed-step solver in auto mode. Although similar translations already exist from Simulink to different languages, this new translation confers several advantages, on both sides. First of all, through this translation, discrete-time Simulink become available as a programming model for developing synchronous BIP components. That is, Simulink models can be smoothly integrated in larger heterogeneous BIP systems and composed with other components, either native BIP or translated from other languages (e.g., Lustre). Furthermore, BIP is supported by an extensible toolbox which includes functional validation and code generation features. The translation from Simulink into BIP allows the validation and implementation of Simulink models. In particular, compositional and incremental generation of invariants can be applied for complex Simulink models. Finally, the BIP toolset includes a highly parametric and efficient code generation chain, targeting different implementation models (sequential, multi-threaded, distributed, real-time, etc). These compilation paths are also becoming available for Simulink models.

From a more technical point of view, the translation is structural and incremental. It associates with each Simulink block a unique synchronous BIP component. For atomic blocks (such as operators), the associated components are predefined into a specific library for Simulink. For structured blocks (such as subsystems), the associated components are (recursively) obtained by composition of their inner subcomponents. This composition is also defined structurally i.e., dataflow and activation links used within Simulink blocks are translated into connectors in BIP. Moreover, our translation reveals only the minimal control coordination structure needed for correct execution of Simulink models, in each step. These properties confirm that synchronous BIP is actually an appropriate formalism for providing a formal semantics for discrete-time Simulink. We show structural equivalence between a Simulink model and the corresponding BIP model. That is, there exist a direct correspondence between the architecture of the two models. Henceforth the generated BIP models can be easily understood and validated by Simulink users.

Finally, all the synchronous BIP models obtained by translation satisfy important structural properties. According to [6], the modal flow graphs representing behavior of the obtained BIP models are well-triggered and obey the syntactic conditions for confluence and deadlock-freedom. These results guarantee predictable behavior of the considered subclass of Simulink models and validate the intuitive simulation semantics (i.e., single-trace) of Simulink.

The translation is currently implemented in the Simulink2BIP tool. We report several experiments on demonstration models provided by MATLAB/Simulink as well as benchmarks developed by ourselves. The translation time into synchronous BIP is negligible. We show that, moreover, the executable code generated from synchronous BIP models has comparable runtime performances as the code produced by the Real-Time Workshop tool provided by MATLAB.

Related Work

The work in [7] presents a translation for a subset of MATLAB/Simulink and Stateflow into equivalent hybrid automata. The translation is specified and implemented using a metamodel-based graph transformation tool. The translation allows semantics interoperability between the Simulink's standard tools and other verification tools.

The work of [8, 9] is probably the closest to our work. These papers present a compositional translation for discrete-time Simulink and respectively discrete-time Stateflow models into Lustre programs [10]. This work leverages the use of validation and (certified) code generation techniques available for Lustre to Simulink models. The translation consists of three steps: type inference, clock inference, and hierarchical bottom-up translation. It has been implemented by the `S2L` tool [11].

We can also mention [12] where a restricted subset of MATLAB/Simulink, consisting of both discrete and continuous blocks, is translated into the COMDES framework (*Component-based Design of Software for Distributed Embedded Systems*). However, this work focuses on the relation between control engineering and software engineering related activities.

Finally, [13] presents a tool which automatically translates discrete-time Simulink models into the input language of the NuSMV model checker. This translation allows efficient symbolic verification techniques to Simulink models used in safety-critical systems.

The fragments translated in [7], [12] and [13] are either incomparable or handled differently. For instance, the translation reported in [7] focuses on continuous-time models, and allows for a limited discrete behavior represented using switches. The work [13] covers an important part of the discrete-time fragment, and in particular, n-dimension signals and related operators (mux, demux). Nevertheless, it does not consider blocks such as the discrete transfer functions, and moreover, it seems to be restricted to models with unique sample time. The solution chosen in [12] for handling multiple sample times is also different. Although, the precise translation is not explained thoroughly in the paper, it is claimed that it relaxes the exact timing constraints of Simulink, since they are fundamentally impossible to implement and unnecessarily restrictive.

Finally, we cover almost the same discrete-time fragment as [9]. Also, we adapt exactly the same semantics choices. However, we believe that our translation method provides a much understandable representation, which better illustrates the control and data dependencies in the Simulink model. For example, we are using (generic) explicit components for adaptation of sample times for signals going into/coming from subsystems. In the Lustre translation, this adaptation is hard-coded using sampling/interpolation operators and gets mixed with other (functional) equations of the subsystem. Also, we do not hard-wire the sample time of signals using absolute clocks. Instead, we merely track all the sample time dependencies (e.g., equalities) within the model and define them only once, at the upper layer, using a sample-time period generator.

Organization of the paper

The paper is structured as follows. An introduction to MATLAB/Simulink is presented in Section 2. Section 3 presents a short description of synchronous BIP. The translation from Simulink to the synchronous BIP is described in Section 4. The implementation and experimental results are presented in section 5. Section 6 provides conclusions and directions for future work.

2 MATLAB/Simulink

MATLAB/Simulink is a very popular commercial tool for designing and simulating hybrid dynamical systems. It is widely used for industrial applications as well as for educational purposes. In this section, we review the major Simulink concepts relevant for our translation.

2.1 Signals

Models described in the discrete-time fragment of Simulink [1] operate on discrete-time signals, that are, piecewise-constant functions defined on the time domain $\mathbb{R}_{\geq 0}$ and with values on an arbitrary data domain (usually, a fixed power set $\mathbb{R}^k$).

Simulink models define transformations on discrete-time signals by means of structured block diagrams. These diagrams are constructed hierarchically from atomic blocks, defining elementary transformations (e.g., delay, sampling, arithmetic, etc.), and dataflow links, expressing instantaneous data communication.

Every signal s in a discrete-time Simulink model is characterized by its sample time, that is, the period $k > 0$ of time at which the signal can change its value. Hence, a signal s may change its value only at integer multiples[1] of k, and remains unchanged within every left-closed right-open interval $[n{\cdot}k, (n{+}1){\cdot}k[$, for $n \in \mathbb{N}$.

In Simulink models, the sample time of signals can be either explicitly provided by the modeler e.g., as an annotation to atomic blocks, or left unspecified. In the latter situation, the sample time is *inherited*, that means, inferred from the sample times of related signals using Simulink specific inference rules.

2.2 Ports and Atomic Blocks

Data ports Simulink uses inports and outports to define dataflow connection endpoints in subsystems. They are used to transfer signals between the subsystems and their environment. The sample time of the ports defines the period for which the signal is updated (i.e., read or written). Inports and outports can be seen in figure 1(a).

Control ports Simulink uses control ports to produce triggering events (trigger port) or to provide enabling conditions (enable port) for the execution of subsystems. Figure 1(b) shows the graphical notation for the two types of control ports.

A trigger port produces an event that activates the execution of a triggered subsystem depending on some condition on an incoming signal. In Simulink, this condition can be either *rising*, *falling* or both. For example, in case of *rising*, the activation event is produced when the input signal rises from a negative or zero value to a positive value.

An enable port defines a condition for the execution of an enabled subsystem depending on an incoming signal. In Simulink, the enabling condition holds as long as the value of the incoming signal is greater than zero.

Sources and Sinks Source blocks produce signals according to some patterns and with a specified or inherited sample time. Some examples are the pulse generator and the constant blocks (see figure 1(c)).

Conversely, sink blocks read signals. An example is the scope block which is used to display graphically one or more input signals (see figure 1(d)).

Combinatorial blocks Combinatorial blocks combine one or more input signals and produce one (or more) output signal(s) as the result of an instantaneous operation. The sample times of all input and output signals are equal. Some examples of combinatorial blocks provided by Simulink are usual arithmetic operators, relational operators, boolean operators, switches, saturation blocks, lookup tables (see figure 1(e)).

Unit delay A unit-delay block delays the input signal for one period of the (input) sample time. During the first period, the unit-delay produces a user-specified constant signal value. This block may also perform a sample time change between the input and output signals as follows: the sample time of the output can be smaller than (i.e., strict integer divisor of) the sample time of the input signal (see figure 1(f)).

[1] Simulink allows as well for an offset, however for the sake of simplicity we always consider this offset equal to zero.

Zero-order hold A zero-order hold block acts as a sampler. It holds the output constant for one period of the (output) sample time with the latest value of the input. Also, this block may perform a sample time change between input and output signals, as follows: the sample time of the output can be greater than (i.e., strict integer multiple of) the sample time of the input signal (see figure 1(f)).

Transfer functions A transfer function block transforms an input signal according to a given discrete-time transfer function. The sample time of the input and output signals are equal.

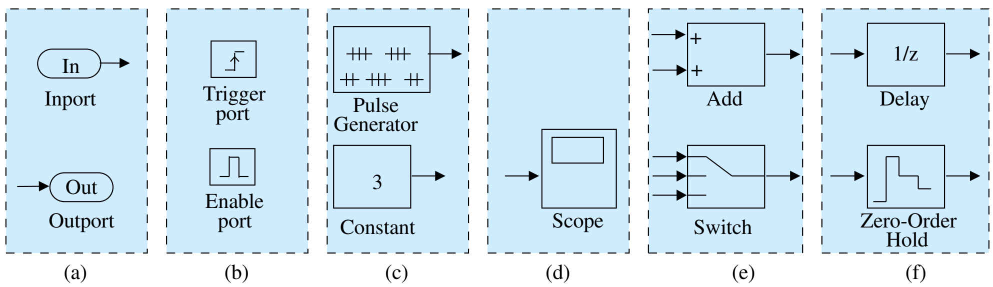


Figure 1: Ports and basic atomic blocks in Simulink

2.3 Subsystems

Subsystems are user-defined assemblies constructed recursively from atomic blocks and other subsystems. They are used to encapsulate some reusable functionality, that can be plugged (i.e., called) in a system model or other subsystems.

The communication between subsystems and their calling environment is realized through data ports. That is, ports are simply used to convey signals produced outside (resp. inside) towards (resp. outwards) the subsystem.

In addition, there exists also some support for execution control of subsystems. Simulink offers two basic mechanisms: *trigger* conditions, that can be used to activate triggered subsystems for execution and *enabling conditions*, that are used to enable/disable the execution of a subsystem.

Triggered Subsystems Triggered subsystems execute instantaneously only when a trigger event occurs. Trigger events are defined as the rising or falling (or both) of a signal defined outside the subsystem.

Triggered subsystems do not have explicit sample time i.e., since their execution is triggered by data-change events and is not directly time dependent. Practically, Simulink requires that all blocks within triggered subsystems have inherited sample time. Consequently, triggered subsystems contain only atomic blocks and triggered subsystems but not periodic (nor continuous time) subsystems.

Example 1 *Figure 2 (left) shows a triggered subsystem. The signal y activates the execution of the triggered subsystem B. When a trigger event occurs, the subsystem instantaneously updates its input value a and writes its output b.*

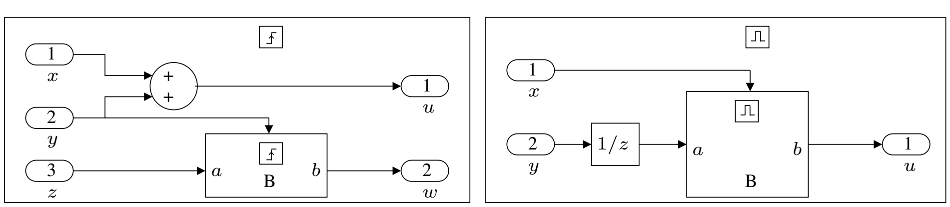


Figure 2: Example of triggered subsystem (left) and enabled subsystem (right)

Periodic and Periodic Enabled Subsystems Periodic and periodic enabled subsystems are time dependent. Their execution i.e., reading/updating of input/output signals is done according to explicit sample times defined from their inner blocks. Implicitly, all the sample times are observed on a unique global time defined for the model, that means, execution is synchronized with respect to a global time.

In the case of periodic enabled subsystems, execution is constrained by the actual value of an external signal. That is, the subsystem (i.e., its inner blocks) executes only if the enabling signal has a positive value and stays unchanged otherwise.

Finally, Simulink does not impose any syntactical restrictions on the inner blocks of periodic subsystems. However, type checking and sample time checking rules are applied to ensure consistency of computations e.g., the GCD-rule for combinatorial operators. For a detailed discussion see [9].

Example 2 *Figure 2 (right) shows an enabled subsystem. The execution of the enabled subsystem B depends on the value of the signal x. As long as the value of x is positive the subsystem updates its input a and produces b.*

3 Synchronous BIP

BIP [2] – Behavior, Interaction, Priority – is a component framework for modeling, analysis and implementation of heterogeneous real-time systems. BIP supports a component construction methodology based on the assumption that components are obtained as the superposition of three layers: (1) behavior, expressed in terms of extended automata, (2) interactions, describing the cooperation between actions of the behavior and (3) priorities, rules specifying scheduling policies for interactions. Layering implies a clear separation between behavior and architecture (connectors and priority rules).

At the lower level of BIP, atomic components contain behavior described by automata and extended with arbitrary computations (expressed as C/C++ functions/methods) on arbitrary data structures (instances of C/C++ data types). Automata transitions are triggered by ports, that are, action names used later to specify interactions. Moreover, ports may be associated with local data of atomic components. These data are available for use (i.e., reading or writing) when interactions involving that port are executed. Interactions are specified in connectors as sets of ports and have also associated an arbitrary computation involving port's data (expressed as C/C++ functions/methods). They can be executed when all atomic components involved are ready to interact i.e., every component reaches some control location enabling a transition labeled by the required port. Whenever enabled, the execution of an interaction is done in two steps: first, the interaction code is executed as an atomic step, then all involved components execute (concurrently) the local computations of the interacting transitions. When several interactions are enabled for execution, the choice is restricted according to priority rules.

Synchronous BIP [6] is a subset of BIP for modeling synchronous systems. Synchronous systems are obtained as the composition of synchronous BIP components, defined and interconnected according to specific restrictions. First, all synchronous BIP components in a system synchronize periodically on a implicit *sync* interaction. This interaction separates the *synchronous steps* within the system. Second, behavior of synchronous BIP components is described by *modal flow graphs* (MFG). These graphs express causal dependencies between ports (and their associated actions) within every synchronous step. That is, in contrast to general BIP components where control flow is represented explicitly using control states and transitions, control flow of synchronous BIP components is expressed implicitly through dependencies. This representation is appropriate for synchronous behavior, which is inherently parallel and (loosely) coordinated by clock and data dependencies.

There are three types of causal dependencies: strong, weak and conditional. For two ports p and q, we say that:

- q *strongly depends on* p if the execution of q must follow p. That means p, q cannot be executed independently, only the execution $p \cdot q$ is possible in a step.

- q *weakly depends on* p if the execution of p may be followed by q. That is either p can be executed alone or the sequence $p \cdot q$.

- q *conditionally depends on* p if both p and q can be executed, q must follow p. Conditional dependency requires that if both p and q occur, only the sequence $p \cdot q$ is possible, otherwise p and q can be executed independently.

Henceforth, we will use a simple graph-based representation for modal flow graphs. Vertices represent the ports and the edges (arrows) represent dependencies. We use solid (resp. thin, resp. dotted) arrows to denote strong (resp. weak, resp. conditional) dependencies.

Example 3 *Figure 3 (left) shows the synchronous BIP component that samples an input value according to a slower clock* q*. The incoming and the outgoing events are triggered by different activation events* act^p *and* act^q *respectively. The strong dependencies between data events and activation events, enforce the execution of the* in^x *and* out^y *at each activation of* act^p *and* act^q *repsectively. The activation event* act^q *depends weakly on* act^p *and the output* out^y *depends conditionally on the input event* in^x*. Thus an input* v *is always read through the event* in^x *and whenever required, an output* v *is produced through the event* out^y *with the most recent value of the input.*

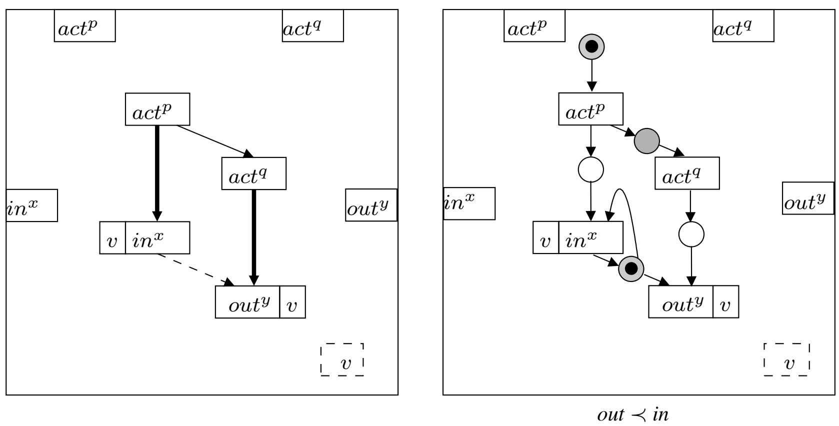


Figure 3: Example

In [6] we have proven that for the subclass of *well-triggered* modal flow graphs we can guarantee deadlock-freedom and confluence of execution using simple syntactic conditions. Consistency between the three different types of dependencies is defined by the following constraints: (i) every port must have a unique minimal strong cause and (ii) every port has exclusively either strong or weak causes. A modal flow graph is called *well-triggered* if it satisfies the above two properties.

A modal flow graph is *deadlock-free* if every synchronous step eventually terminates, that is, reaches a configuration where the component can cycle, by synchronizing with all the others (and begin the next step). For well-triggered modal flow graphs, deadlock-freedom is guaranteed if the guards of ports having strong causes are trivially true. Intuitively, this means that, once started, every computation can be carried out successfully up to a global synchronization point.

A modal flow graph is *confluent* if the result of a step is deterministic, regardless the order chosen for execution of ports. For well-triggered modal flow graphs, confluence is guaranteed by the non-interference of actions attached to independent ports, that are, ports non-causally related. More precisely, non confluent behavior can occur only if actions of independent ports are accessing the same data: different orders of execution may lead to different results.

We have defined composition of synchronous components as a partial internal operation parameterized by a set of interactions. Given a set of synchronous components, we obtain a product component by glueing together the ports (and associated actions) interconnected by interactions.

Example 4 *Figure 4 shows an example of a producer/consumer connected through a sampling component. Inputs are produced on a faster rate than outputs are consumed. The synchronous components are composed by synchronizing the activation events* act^i *and* act^o *and the data events* out *and* in*. The sampling component reads inputs each time the producer component produces outputs through the connection*

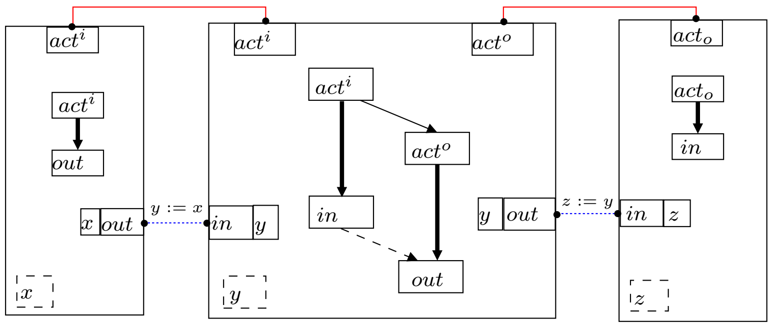


Figure 4: Example of composition of synchronous BIP components

$y := x$. When the act^o event is triggered, the sampling component provides input values to the consumer through the interaction $z := y$.

4 Translation

4.1 Overview

In [6], we provided a modular translation from the synchronous language Lustre [10] into well-triggered components of Synchronous BIP. The proposed translation exhibits maximal parallelism, that is, it enforces only the absolutely necessary dependencies between events needed for correct execution. Moreover, we have shown that the models obtained from Lustre are always deadlock-free and confluent.

The translation is from Simulink to synchronous BIP is also modular

and enjoys the same properties as the translation of Lustre. It associates with each Simulink block B a unique synchronous BIP component M_B. Moreover, basic Simulink blocks e.g., operators, are translated into elementary (explicit) synchronous BIP components. Structured Simulink blocks e.g., subsystems, are translated recursively as composition of the components associated to their contained blocks. The composition is also defined structurally i.e., dataflow and activation links used within the subsystem are translated to connectors.

Synchronous BIP components associated to Simulink blocks involve two categories of events, control events and data events:

- control events, including $act^p, \cdots$ and $trig^q, \cdots$ denote respectively *activation* events and *triggering* events. These events represent pure input and output control signals. They are used to coordinate the overall execution of modal flow graph behavior and correspond to control mechanisms provided by Simulink e.g., sample times, triggering signals, enabling conditions, etc.

- data events, including $in^x, \cdots$ and $out^y, \cdots$ denote respectively *input* events and *output* events. These events transport data values into and from the component. They are used to build the dataflow links provided by Simulink.

Modal flow graphs obtained by translation enjoy important structural properties. First, they are well-triggered [6].

Second, every data event is strongly dependent on exactly one of the activation events. Intuitively, this means that input/output of data is explicitly controlled by activation events. Third, all synchronous BIP components obey the syntactic conditions for confluence and deadlock-freedom defined in [6].

Finally, the translation of a Simulink model B needs an additional synchronous component Clk_B, which generates all activation events $act^{k_1}, act^{k_2}, \ldots$ corresponding to periodic sample times used within the model. The final result of the translation will be the composition of M_B and Clk_B with synchronization on activation events.

Within Clk_B, the activation events have are produced using a global time reference and must obey the corresponding ratio, respectively $k_1, k_2, \ldots$. A concrete example of such a synchronous BIP component is provided in figure **??**. The same construction can be easily generalized to any number of integer sample times.

Example 5 *Figure 5 shows the synchronous BIP component that produces different clock events for every 3, 4 and 6 units of time. The component uses a variable* c *to measure time and has five ports* $tick$, act_3, act_4, act_6 *and* $reset$. *The port* $tick$ *represents a global clock tick. This port is triggered every synchronous step and increases the value of* c *by one. A clock event* $(act_k)_{k=3,4,6}$ *is then produced each time the period* k *divides the current time* c, *denoted by* $k|c$. *The port* $reset$ *is used to reset* c *every 12 time units, that is, the least common multiple of all the periods. Let us notice that ports* act_3 *and* act_4 *are weakly dependent on the* $tick$ *port, and moreover port* act_6 *is weakly dependent on port* act_3. *The port* $reset$ *depends conditionally on the ports* act_4 *and* act_6. *The guards and the causal dependencies ensure that, in every synchronous step, exactly one of the following sequences is executed:* $tick$, $tick \cdot act_3$, $tick \cdot act_4$, $tick \cdot act_3 \cdot act_6$, $tick \cdot ((act_3 \cdot act_6)|act_4) \cdot reset$ *(where* $|$ *denote the shuffling of two sequences).*

Figure 5 (right) shows the equivalent representation of the modal flow graph using a 1-safe Petri (in every place there is at most one token at every time) net with priorities. The Petri net represents valid execution for one synchronous step. The tokens already in places define the initial marking, that is, the initial state of execution. All other places without tokens are final palces. At firing, tokens are removed from initial places and added to final places. The behavior of the Petri net is restricted by the rule which states the $reset$ *event has lower priority than all the other events.*

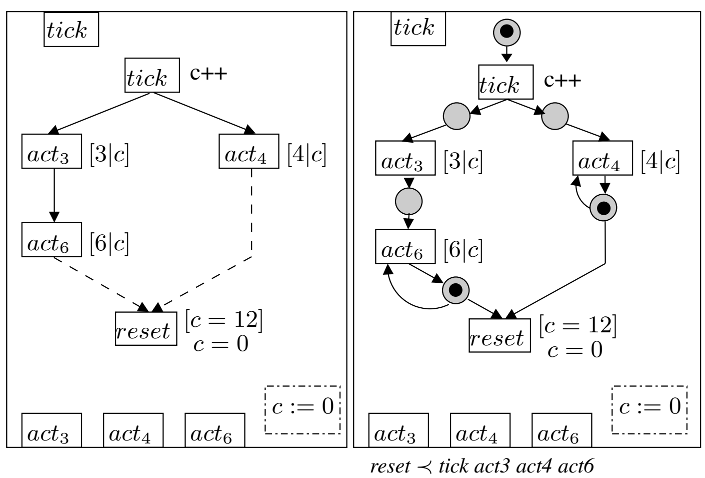
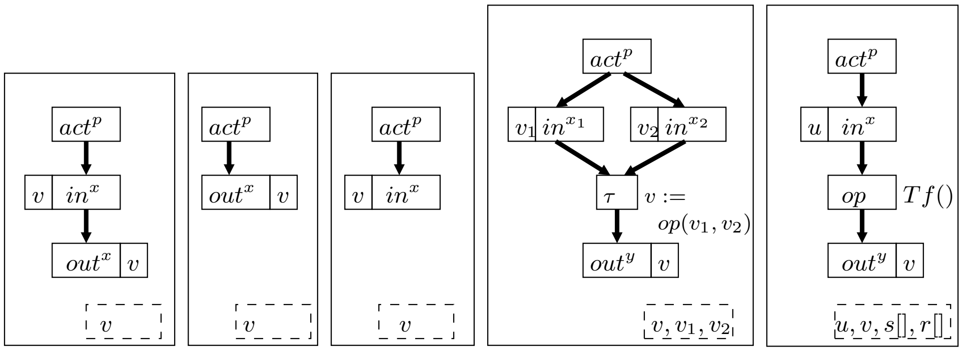


Figure 5: A multi-period clock generator described using modal flow graphs (left) and its semantics using priority Petri nets (right)

We note that the translation is subject to several restrictions. Only models simulated by Simulink can be translated to synchronous BIP. All models are simulated with the method called "solver: *fixed step* and mode: *auto*". Moreover, for the sake of simplicity, we consider only sample times with no offset and explicitly specified for all Simulink blocks.

4.2 Ports and Atomic Blocks

Simulink inports and outports are translated into elementary synchronous BIP components shown in figure 6 (a). These graphs represent a simple identity flow i.e., for a port x, at each activation event act^p one value v of data comes in and goes out through the events in^x and respectively out^x.



(a) (b) (c) (d) (d)

Figure 6: Elementary components for in/outports (a), sources (b), sinks (c) and combinatorial blocks (d)

Simulink sources and sink blocks are translated into elementary modal flow graphs as shown in figure 6 (b). At each activation event act^p, these graphs produce (respectively consume) one data value v through the output event out^x (respectively input event in^x).

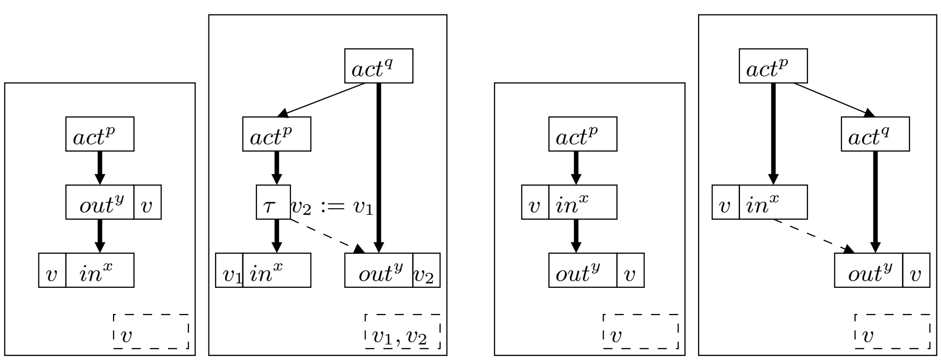
Combinatorial blocks are translated as shown in figure 6 (c). At each activation event act^p, actual data values v_1, v_2 are received on all input events in^{x_1}, in^{x_2} and then, the output value v is computed and sent on the output port out^y.

Transfer functions are translated as shown in figure 6 (right). For a given transfer function $\frac{b_0 z^0 + \ldots + b_q z^{-q}}{1 + a_1 z^{-1} + \ldots + a_p z^{-p}}$ the computation is realized by the function $Tf()$ as follows:

$$\begin{array}{rcl} r[0] & := & u \\ s[0] & := & \sum_{j=0}^{q} b_j r[j] - \sum_{i=1}^{p} a_i s[i] \\ r[j] & := & r[j-1] \text{ for all } j = q \text{ down to } 1 \\ s[i] & := & s[i-1] \text{ for all } i = p \text{ down to } 1 \\ v & := & s[0] \end{array}$$

where s and r are buffers for the input/output values.

Figure 7 shows the synchronous BIP components corresponding to unit-delay blocks (a) and zero-order-hold blocks (b) of Simulink.



(a) Unit-Delay (b) Zero-Order Hold

Figure 7: Elementary components for unit-delay and zero-order-hold

Since these blocks can be used in Simulink to change the sample time of the incoming signal we provide two alternative translations. The first corresponds to identical (unchanged) sample time. In this case, the modal flow graphs are rooted by a unique activation event act^p which triggers both the input

in^x and the output out^y events. The second corresponds to different sample times for the incoming and outgoing signals. In this case, the input in^x and output out^y events are triggered by different activation events respectively, act^p and act^q. Moreover, the two activation events are also weakly dependent in some order, and this dependency enforces the Simulink restriction that unit-delay (respectively zero-order-hold) elements can be used to increase (respectively decrease) the sample time of the signal. Furthermore, input and output events are conditionally dependent on each other, in order to represent the expected behavior i.e., unit-delay is delaying any input for at least one (input) sample time period.

4.3 Subsystems

4.3.1 Triggered Subsystems

Triggered subsystems are translated into synchronous BIP components with a unique activation event $act^\perp$ and several input and output events, one for every inport respectively outport defined within the subsystem. The general interface is illustrated in figure 8 (left).

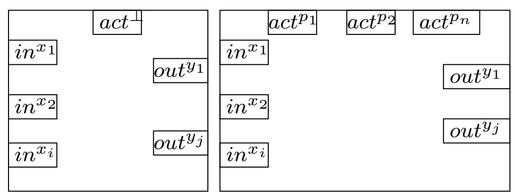


Figure 8: The general interface in components of triggered subsystem (left) and enabled subsystem (right)

We know that, according to Simulink restrictions, all the atomic blocks used within a triggered subsystem have inherited sample time. Moreover, a triggered subsystem can only contain triggered subsystems but not periodic or enabled subsystems. Hence, the only possible connections within a triggered subsystem are dataflow connections which relate outports to inports of different blocks and triggering connections which activate inner triggered subsystems.

As mentioned earlier, the translation of subsystems is structural. The synchronous BIP component corresponding to a triggered subsystem is obtained by composition of its constituent components. The composition i.e, the connectors, reflects the data-flow and activation links used within the subsystem.

More precisely, the translation proceeds as follows.

First, it collects the synchronous BIP components of all of the constituent blocks. We distinguish the following categories:

- in/outports - ports are translated as shown in the previous section. Components associated with ports play a particular role in the definition of the interface of the resulting (composed) component. Input (respectively output) events defined by the components associated to inports (respectively outports) will not be connected by composition within the subsystem and become part of the interface.

- atomic blocks are translated as shown in the previous section. Let us remark that all these blocks will lead to components with a unique activation event $act^\perp$. In particular, this is also the case for unit-delay and zero-order-hold elements since they are activated by the unique sample time of the subsystem;

- triggered subsystems - these subsystems are translated recursively, following the same procedure. We simply rely on their interface to connect them.

Second, the components are composed by synchronization according to dataflow and triggering connections in Simulink. The different types of connections and their translation are illustrated in figure 9 (left). We distinguish basically three cases:

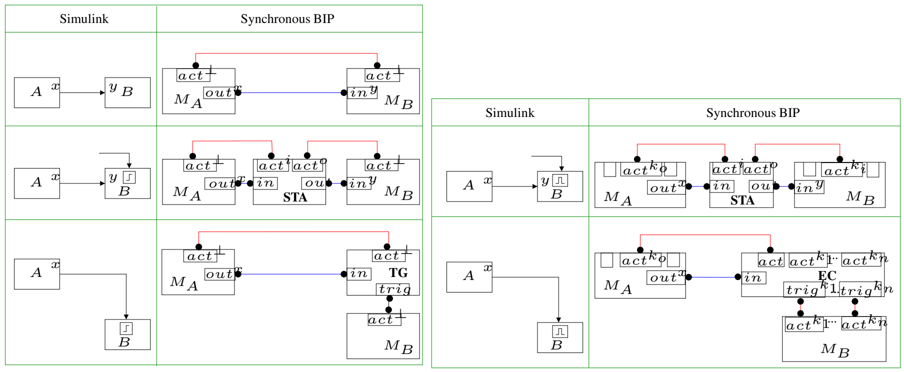


Figure 9: Translation of connections in triggered subsystems (left) and in enabled subsystem (right)

- dataflow connection between blocks operating on the same sample time e.g., outport x of block A is connected to inport y of block B as shown in figure 9 (left-top). In this case, the dataflow connection is translated into a strong synchronization between the output event out^x of M_A and input event in^y of M_B. Moreover, the activation events of M_A and M_B are also strongly synchronized.
- dataflow connection between blocks operating on different sample times e.g., outport x of block A is connected to inport y of block B which is triggered by some other event, as shown in figure 9 (left-mid). In this case, the connection is realized by passing through a *sample-time-adapter* (STA) component. This component is presented in detail in figure 10 (left) and allows the correct transfer of data between a producer and a consumer activated by different events. Let us notice that the two activation events of the adapter component are indeed synchronized with the activation events of respectively M_A and M_B.

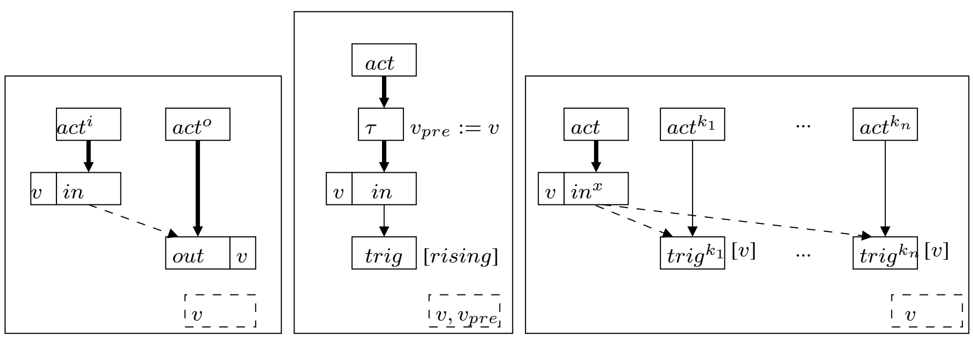


Figure 10: Additional components for the *sample-time-adapter* (left), the *trigger-generator* (middle) and the *enable-generator* (right)

- triggering connection i.e., activation of an inner triggered subsystem e.g., outport x of block A is used to trigger the block B as shown in figure 9 (left-bottom). In this case, the connection is realized by passing through a *trigger-generator* (TG) component. This component is presented in figure 10 (middle). It produces a triggering event $trig$ whenever some condition on the input signal x holds. In Simulink this condition can be either *rising* (value changed from a negative to a positive value, $g_{rising} \equiv v_{pre} \leq 0, v_{pre} < v, 0 \leq v$), *falling* (conversely, value changed from a positive to a negative value) or *either* (rising or falling).

Finally, all the $act^{\perp}$ events which are not explicitly synchronized with a $trig$ event (i.e., occurring at top level) are synchronized and exported as the $act^{\perp}$ event of the composed synchronous BIP component.

Example 6 *Figure 11 (left) illustrates the complete translation of the triggered subsystem shown in figure 2 (left). The triggered subsystem M_B is strongly synchronized with the $trig$ event produced by the trigger generator TG. STAs are connected to the inputs and outputs of the M_B and are strongly synchronized with the activation events, $trig$ and $act_\perp$. The exported activation event $act_\perp$ triggers all the activation events which are not synchronized with the $trig$ event.*

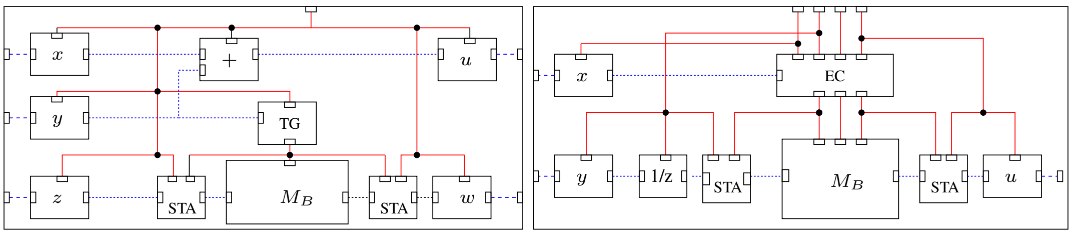


Figure 11: Complete translation of the triggered subsystem shown in figure 2 (left) and of the enabled subsystem of figure 2 (right)

4.3.2 Periodic and Enabled Subsystems

Periodic and enabled subsystems are translated to synchronous BIP components with multiple activation events $act^{k_1}, \cdots, act^{k_n}$, each such event corresponding to a fixed sample time $k_i \in \mathbb{R}$ used explicitly within the subsystem (or recursively, in some of its sub-subsystems). Also, as for triggered subsystems, the associated component has multiple input and output events, one for every inport respectively outport defined within the subsystem. The general interface is shown in figure 8 (right).

The construction of the component associated to a periodic subsystem (or enabled) subsystem is also structural and incremental. It extends the method defined previously for triggered subsystems. As before, first it collects the components for all the constituent blocks, then it composes them according to dataflow, triggering and enabling connections defined in Simulink.

The translation of the new categories of Simulink connections occurring in the context of a periodic subsystem is illustrated in figure **??**. We distinguish two new cases, as follows:

- dataflow connection between subsystems having different enabling conditions e.g., outport x of A connected to input y of B as illustrated in figure 9 (right-top). In this case, the connection is realized by passing through a sample-time-adapter component in order to accommodate for the possible different activation times for input and output events. Let us remark that only the activation events act^{k_o} and act^{k_i} triggering respectively the events out^x in M_A and in^y in M_B have to be synchronized with the adapter, whereas all other act events remain unconstrained.

- enabling condition i.e., conditional execution of the subsystem depending on some condition e.g., outport x of A defines the enabling condition for B as illustrated in figure 9 (right-bottom). Such a connection requires an additional *enabling-condition* (EC) component, presented in detail in figure 10 (right).

  Intuitively, the EC component filters out any (periodic) activation event act^{k_i} occurring when the input signal x is false (or negative). Otherwise, it propagates the activation event renamed as $trig^{k_i}$.

Any other categories of connections are handled as for triggered subsystems.

Finally, all activation events act^{k_i} which correspond to the same sample time k_i and which are not explicitly synchronized with a $trig^{k_i}$ event (i.e., occurring at top level and not filtered by some enabling condition) are strongly synchronized and exported as the act^{k_i} event on the interface of the composed synchronous BIP component.

Example 7 *Figure 11 (right) illustrates the complete translation of the enabled subsystem shown in figure 2 (right). We consider that the delay (1/z), the output y and one of the blocks inside the enabled subsystem* *are executed on different sample times. Activation events are strongly synchronized according to their sample times and exported on the interface of the global MFG. The enabling condition is producing three activation events one for each sample time associated to the enabled subsystem.*

5 Experimental work

The translation has been implemented in the Simulink2BIP tool illustrated in figure 12. The tool Simulink2BIP parses MATLAB/Simulink model files (`.mdl`), and produces synchronous BIP models (`.bip`). The generated models reuse a (hand-written) predefined component library of atomic components and connectors (`simulink.bip`). This library contains the most common atomic blocks (sources, combinatorial operators, memories, transfer functions, etc) as well as the most useful connectors (for in/out data transfer and for control activation). Synchronous BIP models can be further used either to generate standalone C++ code (using the tool `BIP2C`) or as parts of larger BIP models. In the first case, the C++ code can be compiled and executed as such i.e., no middleware is needed for execution.

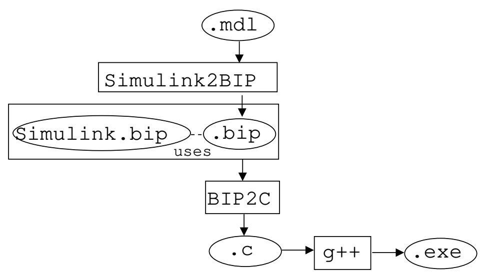


Figure 12: The tool architecture

Table 13 summarizes experimental results on several Simulink models. We have discretized and translated several demo examples available in MATLAB/Simulink including the *Anti-lock Braking* system, the *Conditionally executed subsystem*, the *Enabled subsystem demonstration* and the *Thermal model of a house*. Also, we have translated the examples provided in [9] i.e., the *Steering Wheel* application and the *Big ABC*. Finally, we have considered several artificial benchmarks, respectively the *16-bit counter*, *64-bit counter*. The table provides information about the complexity of these models. #A is the number of atomic blocks, #P the number of periodic blocks, #T the number of triggered subsystems and #E the number of enabled subsystems. As illustrated in the table, our translation tool actually covers a significant number of Simulink concepts.

For all these examples the translation time into synchronous BIP is negligible and therefore it is not reported. Moreover, in all cases, the simulation traces produced respectively by Simulink in simulation mode and by BIP are almost identical. We have observed few small differences for some examples, which are probably due to a different representation of floating-point numbers in Simulink and in BIP.

Finally, for all examples we have produced executable code using respectively the Real-Time Workshop and the BIP code generator. Table 13 reports the execution times measured using the two implementations (i.e., columns t_{rtw} for Real-Time Workshop, t_{bip} for BIP) for different numbers of iterations n. We observe that the BIP generated code slightly outperforms the Real-Time Workshop in almost all the considered examples. Nevertheless, we do not claim that BIP outperforms the Real-Time Workshop in general, because our translation and code generation does not yet cover all the models that can be actually handled by the Real-Time Workshop.

6 Conclusion

We present a translation from the discrete-time fragment of Simulink into synchronous BIP. The translation is structural and incremental. Each Simulink block is associated to a unique synchronous BIP component.

Ex.	#A	#P	#T	#E	n	t_{rtw}	t_{bip}
16-bit counter	97	0	16	0	10^6	1,190s	0,258s
					10^7	11,760s	2,586s
64-bit counter	365	0	60	0	10^6	3,330s	1,863
					10^7	59,283s	25,953s
Anti-lock breaking	39	2	0	0	10^4	0,017s	0,016s
					10^6	0,317s	1,273s
Steering Wheel	120	15	1	0	10^6	1,863s	3,330s
					10^7	7,221s	31,899s
Big ABC	23	2	0	0	10^6	0,323	0,151
					10^7	3,171	1,386
Multi Period	14	0	0	1	10^6	0,466s	0,222s
					10^7	4,313s	2,097s
Enabled Subsystem	24	0	0	2	10^6	0,382s	0,196s
					10^7	3,201s	1,756s
Thermal model house	45	3	0	2	10^6	0,562s	0,751s
					10^7	5,215s	7,565s

Figure 13: Experimental results

Dataflow and activation links are translated to BIP connectors. The synchronous BIP components obtained by the translation of Simulink models have several properties including confluence and deadlock-freedom. We provide an implementation of the translation in a tool called *Simulink2BIP*. Experiments show that the generated BIP models lead to implementations that are comparable to the generated code by Real-Time Workshop of MATLAB.

Although we cover a significant part of the discrete-time fragment of Simulink, our translation is not complete and can be rapidly extended in several directions. First of all, we have considered only uni-dimensional signals, that is, we do not handle mux/demux operators or any other n-dimensional combinatorial operators. Second, we have considered only (perfect) periodic sample times i.e., we do not handle sample times with a non-zero initial offset. Third, for periodic enabled subsystems we have translated only the *held* policy, and not yet the *reset* policy. That is, in *held* mode, the output values are kept constant as long as the block is disabled, whereas in *reset* mode, the outputs as well as the status of some internal blocks (such as integrators) have to be reset.

On a longer term perspective, we would like to extend our translation to the full discrete-time fragment. This must include all of the conditionally executed subsystems, like the triggered and enabled subsystems, the function-call subsystems as well as user defined functions blocks. Finally, we plan to define a similar translation for discrete-time Stateflow.

References

[1] `http://www.mathworks.com/products:simulink/`. 1, 2.1

[2] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time systems in BIP," in *Proceedings of SEFM'06*, pp. 3–12, invited talk. 1, 3

[3] S. Bliudze and J. Sifakis, "The algebra of connectors—structuring interaction in BIP," *IEEE Transactions on Computers*, vol. 57, no. 10, pp. 1315–1330, 2008. 1

[4] A. Basu, L. Mounier, M. Poulhiès, J. Pulou, and J. Sifakis, "Using BIP for Modeling and Verification of Networked Systems – A Case Study on TinyOS-based Networks," in *Proceedings of NCA'07*, 2007, pp. 257–260. 1

[5] S. Bensalem, M. Gallien, F. Ingrand, I. Kahloul, and T.-H. Nguyen, "Toward a more dependable software architecture for autonomous robots," *Special issue on Software Engineering for Robotics of the IEEE Robotics and Automation Magazine*, vol. 16, no. 1, pp. 67–77, March 2009. 1

[6] M. D. Bozga, V. Sfyrla, and J. Sifakis, "Modeling synchronous systems in bip," in *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*. New York, NY, USA: ACM, 2009, pp. 77–86. 1, 3, 3, 4.1

[7] A. Agrawal, G. Simon, and G. Karsai, "Semantic translation of simulink/stateflow models to hybrid automata using graph transformations," in *International Workshop on Graph Transformation and Visual Modeling Techniques*, 2004, p. 2004. 1

[8] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi, "Defining and translating a "safe" subset of simulink/stateflow into lustre," in *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*. New York, NY, USA: ACM, 2004, pp. 259–268. 1

[9] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic, "Translating discrete-time simulink to lustre," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 4, pp. 779–818, 2005. 1, 2.3, 5

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language Lustre," *Proceedings of IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991. 1, 4.1

[11] `http://www-verimag.imag.fr/ss2lus.html/`. 1

[12] N. Marian and S. Top, "Integration of simulink models with component-based software models," *Advances in Electrical and Computer Engineering*, 2008. 1

[13] B. Meenakshi, A. Bhatnagar, and S. Roy, "Tool for translating simulink models into input language of a model checker," in *ICFEM*, 2006, pp. 606–620. 1