Compositional Translation of Simulink Models into Synchronous BIP

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

Compositional Translation of Simulink Models into Synchronous BIP

Vassiliki Sfyrla, Georgios Tsiligiannis, Iris Safaka, Marius Bozga and Joseph Sifakis
Verimag - University of Grenoble - CNRS
Email: name.surname@imag.fr

***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 Simulink models.

I. Introduction

Simulink [1] is a very popular commercial tool for model-based design and simulation of dynamic embedded systems. Simulink lacks desirable features of programming languages, e.g. the Simulink semantics is provided only informally, it is only partially documented and the meaning of models depends significantly on many simulation parameters (e.g. simulation step, solver used, etc).

BIP [2] is a formalism for modeling heterogeneous real-time components. BIP is supported by an extensible toolset which includes functional validation and code generation features. The BIP toolset includes a highly parametric and efficient code generation chain, targeting different implementation models (sequential, multi-threaded, distributed, real-time, etc). Synchronous BIP is a subset of the BIP framework for modeling synchronous data-flow systems [3]. In this paper we provide a translation for the discrete-time fragment of Simulink into synchronous BIP. Through this translation, discrete-time Simulink becomes available as a programming model for developing synchronous BIP components. That is, Simulink models can be smoothly integrated in larger heterogeneous BIP systems. Moreover, the translation from Simulink into BIP allows the validation and implementation of Simulink models using the BIP tools. In particular, compositional and incremental generation of invariants can be applied for complex Simulink models by using the D-Finder tool [4].

The translation is structural and incremental, properties that confirm that synchronous BIP is an appropriate formalism for providing a formal semantics for discrete-time Simulink. The generated BIP models have the same structure as the original Simulink models, henceforth, Simulink users can easily understand and accept them. Moreover, all the synchronous BIP models obtained by translation satisfy important properties. All modal flow graphs representing the behavior are well-triggered, confluent and deadlock-free. These results guarantee predictable behavior of the generated BIP 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.

Related Work

The work in [5] presents a translation for a subset of MATLAB/Simulink and Stateflow into equivalent hybrid automata. The work of [6], [7] 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 [8]. We can also mention [9] where a restricted subset of MATLAB/Simulink, consisting of both discrete and continuous blocks, is translated into the COMDES framework. Finally, [10] presents a tool which automatically translates discrete-time Simulink models into the input language of the NuSMV model checker.

The fragments translated in [5], [9] and [10] are either incomparable or handled differently. We cover almost the same discrete-time fragment as [7]. Also, we adopt exactly the same semantic 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 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.

II. MATLAB/SIMULINK

In this section, we review the major Simulink concepts relevant for our translation.

Models described in the discrete-time fragment of Simulink [1] operate on discrete-time signals. Every signal s is a piecewise constant function characterized by its sample time, that is, the period $k > 0$ of time at which the signal can change its value.

Simulink models are constructed from ports and atomic blocks. Ports are of two types, *data ports*, defining dataflow connection endpoints in subsystems and *control ports* producing triggering and enabling events for the execution of subsystems. Amongst the most used atomic blocks are *sources*, *sinks*, *combinatorial blocks*, *unit delay*, *zero-order hold* and *transfer functions*.

Subsystems are user-defined assemblies constructed recursively from atomic blocks and subsystems. Simulink provides three types of subsystems, triggered, periodic and periodic enabled. *Triggered subsystems* execute instantaneously only when a trigger event occurs. *Periodic* and *periodic enabled subsystems* are time dependent. Their execution is done according to explicit sample times defined from their inner blocks. For periodic enabled subsystems, execution is constrained by the value of an external signal.

III. SYNCHRONOUS BIP

BIP [2] – Behavior, Interaction, Priority – is a component framework for modeling, analysis and implementation of heterogeneous real-time systems. BIP components are obtained as the superposition of three layers: (1) behavior, described by automata extended with C/C++ code, (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).

Synchronous BIP [3] 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 an implicit *sync* interaction. This interaction separates the *synchronous steps* within the system. Second, behavior of synchronous BIP components is described by *modal flow graphs* (MFGs). These graphs express causal dependencies between events (and their associated actions) within every synchronous step. This representation is appropriate for synchronous behavior, which is inherently parallel and (loosely) coordinated by clock and data dependencies.

Modal flow graphs express three types of causal dependencies: strong, weak and conditional. For two events p and q, we say that q *strongly depends on* p, if only the execution $p \cdot q$ is possible in a step, q *weakly depends on* p if either p can be executed alone or the sequence $p \cdot q$, q *conditionally depends on* p 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 eventss and the edges (arrows) represent dependencies. We use solid (resp. thin, resp. dotted) arrows to denote strong (resp. weak, resp. conditional) dependencies (see figure **??**).

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). A modal flow graph is *confluent* if the result of a step is deterministic, regardless the order chosen for execution of events.

In [3] 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. Well-triggered modal flow graphs satisfy additional conditions (i) every event must have a unique minimal strong cause and (ii) every event has exclusively either strong or weak causes.

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 events (and associated actions) interconnected by interactions.

IV. TRANSLATION

A. Overview

The translation from Simulink to synchronous BIP is modular; 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. We consider only Simulink models that have explicitly specified sample times for all signals and which can be simulated using fixed-step solver in single-tasking mode. For details on the translation see [11].

Synchronous BIP components associated to Simulink blocks involve 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 [3]. 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 [3].

Finally, the translation of a Simulink model B needs an additional synchronous component Clk_B, which generates all activation events $act^{k_1}, act^{k_2}, ...$ corresponding to periodic sample times $k_1, k_2, ...$ used within the model. The final result of the translation is the composition of M_B and Clk_B with synchronization on activation events. Within Clk_B, the activation events are produced using a global time reference and must the corresponding ratio, respectively $k_1, k_2, ...$

B. Ports and Atomic Blocks

Simulink ports and atomic blocks are translated into elementary synchronous BIP components. Some examples are explained below and shown in figure **??**.

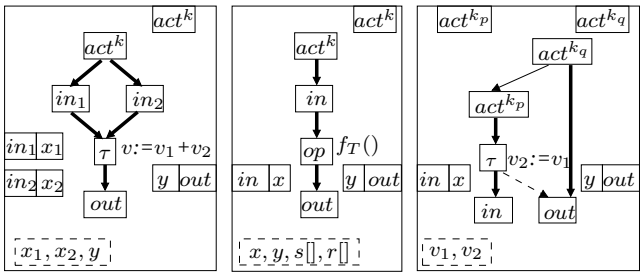

Fig. 1. Elementary MFGs for combinatorial blocks (left), transfer functions (middle) and zero-order hold (right).

For example, figure **??** (right) illustrates the synchronous BIP component for a zero-order hold block. 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 zero-order-hold elements can be used to 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. an output is produced with the most recent value of the input.

C. Subsystems

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 translation proceeds as follows. First, it collects the synchronous BIP components of all of the constituent blocks. We distinguish three categories of blocks, the *in/outports*, the *atomic blocks* and the triggered subsystems. Input (respectively output) events defined by the components associated to inports (respectively outports) become part of the interface. Atomic blocks lead to components with a unique activation event $act^{\perp}$. Triggered 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. We distinguish basically three cases. First, each *dataflow connection* between blocks operating on the same sample time is translated into a strong synchronization between an output event and an input event. Moreover, the activation events of the two components are also strongly synchronized. Second, each *dataflow connection* between blocks operating on different sample times is realized by passing through a *sample-time-adapter* component. The behavior of such a component is similar to the one of the zero-order hold. The *sample-time-adapter* component allows the correct transfer of data between a producer and a consumer activated by different events. Third, *triggering connection* is realized by passing through a *trigger-generator* component. This component produces a triggering event $trig$ whenever some condition on the input signal holds.

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.

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

All connections are handled as for triggered subsystems apart from the *enabling condition*. Such a connection requires an additional *enabling-condition* component which filters out any (periodic) activation event act^{k_i} occurring when the input signal is false (or negative). Otherwise, it propagates the activation event renamed as $trig^{k_i}$ towards the system.

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.

V. Experimental work

The translation has been implemented in the Simulink2BIP tool. 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 and ports 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. The C++ code can be compiled and executed as such i.e., no middleware is needed for execution.

Table 2 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 [7] i.e., the *Steering Wheel* application and the *Big ABC*. Finally, we have considered several artificial benchmarks, e.g. *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.

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 2 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.

Ex.	#A	#P	#T	#E	n	t_{rtw}	t_{bip}
64-bit counter	365	0	60	0	10^6	5,347s	3,115
					10^7	53,652s	31,112s
Anti-lock breaking	39	2	0	0	10^4	0,345s	1,394s
					10^6	3,200s	13,515s
Steering Wheel	120	15	1	0	10^6	0,406s	1,676s
					10^7	3,417s	16,755s
Big ABC	23	2	0	0	10^6	0,359	0,239
					10^7	3,105	2,024
Multi Period	14	0	0	1	10^6	0,465s	0,411s
					10^7	4,012s	3,658s
Enabled Subsystem	24	0	0	2	10^6	0,382s	0,380s
					10^7	3,201s	3,458s
Thermal model house	45	3	0	0	10^6	0,559s	0,853s
					10^7	5,196s	9,624s

Fig. 2. Experimental results

VI. Conclusion

We present a translation from the discrete-time fragment of Simulink into synchronous BIP. The translation is structural and incremental. The synchronous BIP components obtained by 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 have comparable performance 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 directly extended in several directions. For example, we consider only uni-dimensional signals that means we do not handle n-dimensional combinatorial operators. Also, we consider only signals with explicit sample times, i.e. we do not handle inherited sample times.

On a longer term perspective, we would like to extend our translation to the full discrete-time fragment of Simulink. This includes all of the conditionally executed subsystems, like the triggered-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/`.

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

[3] 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.

[4] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis, "D-finder: A tool for compositional deadlock detection and verification," in *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*, 2009.

[5] 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.

[6] 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.

[7] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic, "Translating discrete-time simulink to lustre," *ACM Trans. Embed. Comput. Syst.*, 2005.

[8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language Lustre," *Proceedings of IEEE*, 1991.

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

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

[11] "Compositional translation of simulink models into synchronous bip," Verimag Research Report, Tech. Rep. TR-2010-16.