Compositional Semantics and Analysis of Hierarchical Block Diagrams*

Iulia Dragomir[1], Viorel Preoteasa[1], and Stavros Tripakis[1,2]

[1] Aalto University, Finland
{iulia.dragomir, viorel.preoteasa, stavros.tripakis}@aalto.fi
[2] University of California, Berkeley, USA

Abstract. We present a compositional semantics and analysis framework for hierarchical block diagrams (HBDs) in terms of atomic and composite predicate transformers. Our framework consists of two components: (1) a compiler that translates Simulink HBDs into an algebra of transformers composed in series, in parallel, and in feedback; (2) an implementation of the theory of transformers and static analysis techniques for them in Isabelle. We evaluate our framework on several case studies including a benchmark Simulink model by Toyota.

1 Introduction

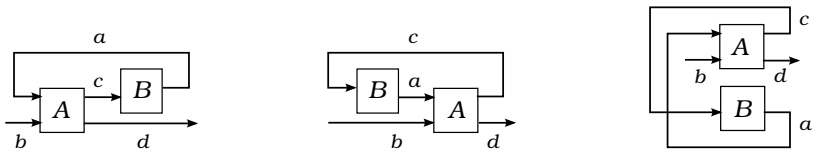
Simulink[3] is a widely used tool for modeling and simulating embedded control systems. Simulink uses a graphical language based on *hierarchical block diagrams* (HBDs). HBDs are networks of interconnected *blocks*, which can be either *basic* blocks from Simulink's libraries, or *composite* blocks (*subsystems*), which are themselves HBDs. Hierarchy is the primary *modularization* mechanism that languages like Simulink offer. It allows to structure large models and thus master their complexity, improve their readability, and so on.

In this paper we present a *compositional* semantics and analysis framework for HBDs, including but not limited to Simulink models. By "compositional" we mean exploiting the hierarchical structure of these diagrams, for instance, reasoning about individual blocks and subsystems independently, and then composing the results to reason about more complex systems. By "analysis", we mean different types of checks, including exhaustive verification (model-checking), but also static analysis such as *compatibility checking*, which aims to check whether the connections between two or more blocks in the diagram are valid, i.e., whether the blocks are compatible.

Our framework is based on the theories of *relational interfaces* and *refinement calculus of reactive systems* [23,19]. The framework can express *open*, *non-deterministic*, and *non-input-receptive* systems, and both *safety* and *liveness* properties. As syntax, we use (temporal or non) logic formulas on input,

* This work has been partially supported by the Academy of Finland, the U.S. National Science Foundation (awards #1329759 and #1139138), and by UC Berkeley's iCyPhy Research Center (supported by IBM and United Technologies).

[3] http://www.mathworks.com/products/simulink/



(a) $\mathsf{feedback}_a(P_A \circ (P_B \parallel \mathsf{Id}))$ (b) $\mathsf{feedback}_c((P_B \parallel \mathsf{Id}) \circ P_A)$ (c) $\mathsf{feedback}_{a,c}(P_A \parallel P_B)$

Fig. 1: Three ways to view and translate the same block diagram.

output, and state variables. As semantics we use *predicate* and *property transformers* [2,19]. To form complex systems from simpler ones we use composition in *series*, in *parallel*, and in *feedback*. Apart from standard verification (of a system against a property) the framework offers: (1) *compatibility* checking during composition; and (2) *refinement*, a binary relation between components, which characterizes *substitutability* (when can a component replace another one while preserving system properties). Compatibility checking is very useful, as it offers a lightweight alternative to verification, akin to type-checking [23]. Refinement has multiple usages, including compositional and incremental design, and reusability. This makes the framework compelling for application on tools like Simulink, which have a naturally compositional hierarchical language.

In order to define the semantics of HBDs in a compositional framework, one needs to do two things. First, define the semantics of every basic block in terms of an *atomic* element of the framework. We do this by defining for each Simulink basic block a corresponding (atomic) *monotonic predicate transformer* (MPT). Second, one must define the semantics of composite diagrams. We do this by mapping such diagrams to *composite* MPTs (CPTs), i.e., MPTs composed in series, in parallel, or in feedback.

As it turns out, mapping HBDs to CPTs raises interesting problems. For example, consider the block diagram in Fig. 1a. Let P_A and P_B be transformers modeling the blocks A and B in the diagram. How should we compose P_A and P_B in order to get a transformer that represents the entire diagram? As it turns out, there are several possible options. One option is to compose first P_A and P_B in series, and then compose the result in feedback, following Fig. 1a. This results in the composite transformer $\mathsf{feedback}_a(P_A \circ (P_B \parallel \mathsf{Id}))$, where $\circ$ is composition in series, $\parallel$ in parallel, and $\mathsf{feedback}_x$ is feedback applied on port x. Id is the transformer representing the identity function. A has two outputs and B only one input, therefore to connect them in series we first form the parallel composition $P_B \parallel \mathsf{Id}$, which represents a system with two inputs.

Another option is to compose the blocks in series in the opposite order, P_B followed by P_A, and then apply feedback. This results in the transformer $\mathsf{feedback}_c((P_B \parallel \mathsf{Id}) \circ P_A)$. A third option is to compose the two blocks first in parallel, and then apply feedback on the two ports a, c. This results in the transformer $\mathsf{feedback}_{a,c}(P_A \parallel P_B)$. Although semantically equivalent, these three transformers have different computational properties.

Clearly, for complex diagrams, there are many possible translation options. A main contribution of this paper is the study of these options in depth. Specifically, we present three different translation strategies: *feedback-parallel* transla-

tion which forms the parallel composition of all blocks, and then applies feedback; *incremental* translation which orders blocks topologically and composes them one by one; and *feedbackless* translation, which avoids feedback composition altogether, provided the original block diagram has no algebraic loops.

Having defined the compositional semantics of HBDs in terms of CPTs, we turn to analysis. Our main focus in this paper is checking diagram *compatibility*, which roughly speaking means that the input requirements of every block in the diagram are satisfied [23,19]. We check compatibility by (1) *expanding* the definitions of CPTs to obtain an atomic MPT; (2) *simplifying* the formulas in the atomic MPT; and (3) checking satisfiability of the resulting formulas.

We report on a toolset which implements the framework described above. The toolset consists of (1) the SIMULINK2ISABELLE compiler which translates hierarchical Simulink models into CPTs implemented in the Isabelle proof assistant[4], and (2) the implementation of the theory of CPTs, together with expansion and simplification techniques in Isabelle. We evaluate our framework on several case studies, including a Fuel Control System benchmark by Toyota [10,11].

2 Hierarchical Block Diagrams

A *hierarchical block diagram* (HBD) is a network of interconnected blocks.[5] Blocks can be either *basic* blocks (from Simulink libraries), or *composite* blocks (*subsystems*). A basic block is described by: (1) a label, (2) a list of parameters, (3) a list of in- and out-ports, (4) a vector of state variables with predefined initial values (i.e., the local memory of a block) and (5) functions to compute the outputs and next state variables. The outputs are computed from the inputs, current state and parameters. State variables are updated by a function with the same arguments. Subsystems are defined by their label, list of in- and out-ports, and the list of block instances that they contain – both atomic and composite.

Simulink allows to model both discrete and continuous-time blocks. For example, UnitDelay (graphically represented as the $\frac{1}{z}$ block in a Simulink diagram) is a discrete-time block which outputs at step n the input at step $n-1$. An Integrator is a continuous-time block whose output is described by a differential equation solved with numerical methods. We interpret a Simulink model as a discrete-time model (essentially an input-output state machine, possibly infinite-state) which evolves in a sequence of discrete steps. Each step has duration Δt, which is a parameter (user-defined or automatically computed by Simulink based on the blocks' time rates).

Algebraic-loop-free diagrams. In this paper we consider diagrams which are free from *algebraic loops*. By "algebraic loop" we mean a feedback loop resulting in instantaneous cyclic dependencies. More precisely, the way we define and check for algebraic loops is the following: first, we build a directed dependency

[4] https://isabelle.in.tum.de/

[5] Our exposition focuses on HBDs as implemented in Simulink, but our method and tool can also be applied to other block-diagram based languages with minor changes.

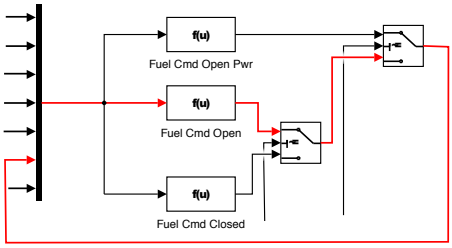
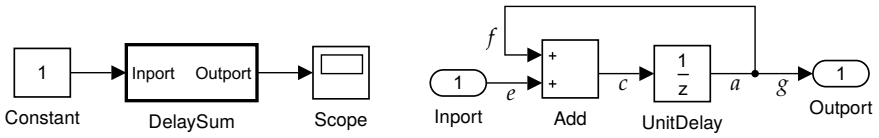


Fig. 2: An extract of Toyota's Simulink Fuel Control System model [10,11]: this diagram is algebraic-loop-free despite the fact that the feedback loop in red is not "broken" by blocks such as Integrator or UnitDelay.

graph whose nodes are the input/output ports of the diagram, and whose edges correspond to connections or to input-output dependencies within a block; second, we check whether this graph has a cycle. The class of algebraic-loop-free diagrams includes all diagrams whose feedback loops are "broken" by blocks such as Integrator or UnitDelay. The output of such blocks does not depend on their input (it only depends on their state), which prevents a cycle from forming in the dependency graph. For example, the diagram of Fig. 1 is algebraic-loop-free if the output of block B does not depend on its input.

But algebraic-loop-free diagrams can also be diagrams where feedback loops are not broken by Integrators or UnitDelays. An example is shown in Fig. 2. Despite the feedback loop in red, which creates an apparent dependency cycle, this diagram is algebraic-loop-free. The reason is that the Fuel Cmd Open block is the function $\frac{1}{14.7}(-0.366 + 0.08979u_7u_3 - 0.0337u_7u_3^2 + 0.0001u_7^2u_3)$, where $u = (u_1, u_2, ..., u_7)$ is the input vector. This function only depends on variables u_3, u_7 of the vector u, and is independent from u_1, u_2, u_4, u_5, u_6. Since the output of the block does not depend on the 6th input link (i.e., u_6), the cycle is broken. Similarly, the outputs of Fuel Cmd Open Pwr and Fuel Cmd Closed are also independent from u_6, which prevents the other two feedback loops from forming a cyclic dependency. This type of algebraic-loop-free pattern abounds in Simulink models found in the industry.

Running example. Throughout the paper we illustrate our methods using a simple example of a counter, shown in Fig. 3. This is a hierarchical (two-level) Simulink model. The top-level diagram (Fig. 3a) contains three block instances: the step of the counter as a Constant basic block, the subsystem DelaySum, and the Scope basic block which allows to view simulation results. The subsystem DelaySum (Fig. 3b) contains a UnitDelay block instance which models the state of the counter. UnitDelay can be specified by the formula $a = s \wedge s' = c$, where c is the input, a the output, s the current state and s' the next state variable. We assume that s is initially 0. The Add block instance adds the two input values and outputs the result in the same time step: $c = f + e$. The *junction* after link a (black dot in the figure) can be seen as a basic block duplicating (or *splitting*) its input to its two outputs: $f = a \wedge g = a$.



(a) Hierarchical Block Diagram (b) DelaySum Subsystem

Fig. 3: Simulink model of a counter with step 1.

3 Basic Blocks as Monotonic Predicate Transformers

Monotonic predicate transformers [6] (MPTs) are an expressive formalism, used within the context of programming languages to model non-determinism, correctness (both functional correctness and termination), and refinement [2]. In this paper we show how MPTs can also be used to give semantics to HBDs. We consider basic blocks in this section, which can be given semantics in terms of *atomic* MPTs. In the next section we consider general diagrams, which can be mapped to *composite* MPTs.

3.1 Monotonic Predicate Transformers

A *predicate* on an arbitrary set Σ is a function $q : \Sigma \to \mathsf{Bool}$. Predicate q can also be seen as a subset of Σ: for $\sigma \in \Sigma$, σ belongs to the subset iff $q(\sigma)$ is true. Predicates can be ordered by the subset relation: we write $q \leq q'$ if predicate q, viewed as a set, is a subset of q'. $\mathsf{Pred}(\Sigma)$ denotes the set of predicates $\Sigma \to \mathsf{Bool}$.

A *predicate transformer* is a function $S : (\Sigma' \to \mathsf{Bool}) \to (\Sigma \to \mathsf{Bool})$, or equivalently, $S : \mathsf{Pred}(\Sigma') \to \mathsf{Pred}(\Sigma)$. S takes a predicate on Σ' and returns a predicate on Σ. S is *monotonic* if $\forall q, q' : q \leq q' \Rightarrow S(q) \leq S(q')$.

Traditionally, MPTs have been used to model sequential programs using weakest precondition semantics. Given a MPT $S : (\Sigma' \to \mathsf{Bool}) \to (\Sigma \to \mathsf{Bool})$, and a predicate $q' : \Sigma' \to \mathsf{Bool}$ capturing a set of *final* states, $S(q')$ captures the set of all *initial* states, such that if the program is started in any state in $S(q')$, it is guaranteed to finish in some state in q'. But this is not the only possible interpretation of S. S can also model input-output systems. For instance, S can model a *stateless* system with a single inport ranging over Σ, and a single outport ranging over Σ'. Given a predicate q' characterizing a set of possible *output values*, $S(q')$ characterizes the set of all *input values* which, when fed into the system, result in the system outputting a value in q'. As an example, the identity function can be modeled by the MPT $\mathsf{Id} : \mathsf{Pred}(\Sigma) \to \mathsf{Pred}(\Sigma)$, defined by $\mathsf{Id}(q) = q$, for any q.

MPTs can also model *stateful* systems. For instance, consider the UnitDelay described in §2. Let the input, output, and state variable of this system range over some domain Σ. Then, this system can be modeled as a MPT $S : \mathsf{Pred}(\Sigma \times \Sigma) \to \mathsf{Pred}(\Sigma \times \Sigma)$. The Cartesian product $\Sigma \times \Sigma$ captures pairs of *(input, current state)* or *(output, next state)* values. Intuitively, we can think of this system as a function which takes as input (x, s), the input x and the current state s, and

returns (y, s'), the output and the next state s', such that $y = s$ and $s' = x$. The MPT S can then be defined as follows:

$$S(q) = \{(x, s) \mid (s, x) \in q\}.$$

In the definition above we view predicates q and $S(q)$ as sets.

Syntactically, a convenient way to specify systems is using formulas on input, output, and state variables. For example, the identity system can be specified by the formula $y = x$, where y is the output variable and x is the input. The UnitDelay system can be specified by the formula $y = s \wedge s' = x$. We next introduce operators which define MPTs from predicates and relations.

For a predicate $p : \Sigma \to \mathsf{Bool}$ and a relation $r : \Sigma \to \Sigma' \to \mathsf{Bool}$, we define the *assert* MPT, $\{p\} : \mathsf{Pred}(\Sigma) \to \mathsf{Pred}(\Sigma)$, and the *non-deterministic update* MPT, $[r] : \mathsf{Pred}(\Sigma') \to \mathsf{Pred}(\Sigma)$, where:

$$\{p\}(q) = (p \wedge q) \quad \text{and} \quad [r](q) = \{\sigma \mid \forall \sigma' : \sigma' \in r(\sigma) \Rightarrow \sigma' \in q\}$$

Transformer $\{p\}$ is used to model non-input-receptive systems, that is, systems where some inputs are illegal [23]. $\{p\}$ constrains the inputs so that they must satisfy predicate p. It accepts only those inputs and behaves like the identity function. That is, $\{p\}$ models a partial identity function, restricted to the domain p. Transformer $[r]$ models an input-receptive but possibly non-deterministic system. Given input σ, the system chooses non-deterministically some output σ' such that $\sigma' \in r(\sigma)$ is true. If no such σ' exists, then the system behaves *miraculously* [2]. In our framework we ensure non-miraculous behavior as explained below, therefore, we do not detail further this term.

3.2 Semantics of Basic Blocks as Monotonic Predicate Transformers

To give semantics to basic Simulink blocks, we often combine $\{p\}$ and $[r]$ using the *serial composition* operator $\circ$, which for predicate transformers is simply function composition. Given two MPTs $S : \mathsf{Pred}(\Sigma_2) \to \mathsf{Pred}(\Sigma_1)$ and $T : \mathsf{Pred}(\Sigma_3) \to \mathsf{Pred}(\Sigma_2)$, their serial composition $(S \circ T) : \mathsf{Pred}(\Sigma_3) \to \mathsf{Pred}(\Sigma_1)$ is defined as $(S \circ T)(q) = S(T(q))$.

For example, consider a block with two inputs x, y and one output z, performing the division $z = \frac{x}{y}$. We want to state that division by zero is illegal, and therefore, the block should reject any input where $y = 0$. This block can be specified as the MPT

$$\mathsf{Div} = \{\lambda(x, y) : y \neq 0\} \circ [\lambda(x, y), z : z = \frac{x}{y}]$$

where we employ lambda-notation for functions.

In general, and in order to ensure non-miraculous behavior, we model non-input-receptive systems using a suitable assert transformer $\{p\}$ such that in $\{p\} \circ [r]$, if p is true for some input x, then there exists output y such that (x, y) satisfies r. MPTs which do not satisfy this condition are not considered in our framework. This is the case, for example, of the MPT $[\lambda(x, y), z : y \neq 0 \wedge z = \frac{x}{y}]$.

For a function $f : \Sigma \to \Sigma'$ the *functional update* $[f] : \mathsf{Pred}(\Sigma') \to \mathsf{Pred}(\Sigma)$ is defined as $[\lambda\sigma, \sigma' : \sigma' = f(\sigma)]$ and we have

$$[f](q) = \{\sigma \mid f(\sigma) \in q\} = f^{-1}(q)$$

Functional predicate transformers are of the form $\{p\} \circ [f]$, and *relational predicate transformers* are of the form $\{p\} \circ [r]$, where p is a predicate, f is a function, and r is a relation. *Atomic predicate transformers* are either functional or relational transformers. Div is a functional predicate transformer which can also be written as $\mathsf{Div} = \{\lambda x, y : y \neq 0\} \circ [\lambda x, y : \frac{x}{y}]$.

For assert and update transformers based on Boolean expressions we introduce a simplified notation that avoids lambda abstractions. If P is a Boolean expression on some variables $x_1, \ldots, x_n$, then $\{x_1, \ldots, x_n : P\}$ denotes the assert transformer $\{\lambda x_1, \ldots, x_n : P\}$. Similarly if R is a Boolean expression on variables $x_1, \ldots, x_n, y_1, \ldots, y_k$ and F is a tuple of expressions on variables $x_1, \ldots, x_n$, then $[x_1, \ldots, x_n \rightsquigarrow y_1, \ldots, y_k : R]$ and $[x_1, \ldots, x_n \rightsquigarrow F]$ are notations for $[\lambda(x_1, \ldots, x_n), (y_1, \ldots, y_k) : R]$ and $[\lambda x_1, \ldots, x_n : F]$, respectively. With these notations the Div transformer becomes:

$$\mathsf{Div} = \{x, y : y \neq 0\} \circ [x, y \rightsquigarrow \frac{x}{y}]$$

Other basic Simulink blocks include constants, delays, and integrators. Let us see how to give semantics to these blocks in terms of MPTs. A constant block parameterized by constant c has no input, and a single output equal to c. As a predicate transformer the constant block has as input the empty tuple (), and outputs the constant c:

$$\mathsf{Const}(c) = [() \rightsquigarrow c]$$

The unit delay block is modeled as the atomic predicate transformer

$$\mathsf{UnitDelay} = [x, s \rightsquigarrow s, x]$$

Simulink includes continuous-time blocks such as the *integrator*, which computes the integral $\int_0^x f$ of a function f. Simulink uses different integration methods to simulate this block. We use the Euler method with fixed time step Δt (a parameter). If x is the input, y the output, and s the state variable of the integrator, then $y = s$ and $s' = s + x \cdot \Delta t$. Therefore, the integrator can be modeled as the MPT

$$\mathsf{Integrator}(\Delta t) = [x, s \rightsquigarrow s, s + x \cdot \Delta t]$$

All other Simulink basic blocks fall within these cases discussed above. Relation (1) introduces the definitions of some blocks that we use in our examples.

$$\mathsf{Add} = [x, y \rightsquigarrow x + y] \qquad \mathsf{Split} = [x \rightsquigarrow x, x] \qquad \mathsf{Scope} = \mathsf{Id} \tag{1}$$

4 HBDs as Composite Predicate Transformers

4.1 Composite Predicate Transformers

The semantics of basic Simulink blocks is defined using monotonic predicate transformers. To give semantics to arbitrary block diagrams, we map them to *composite predicate transformers* (CPTs). CPTs are expressions over the atomic predicate transformers using *serial*, *parallel*, and *feedback* composition operators. Here we focus on how these operators instantiate on functional predicate transformers, which are sufficient for this paper. The complete formal definitions of the operators can be found in [7] and in the Isabelle theories that accompany this paper.[6]

Serial composition $\circ$ has already been introduced in §3.1. For two functional predicate transformers $S = \{p\} \circ [f]$ and $T = \{p'\} \circ [f']$, it can be shown that their serial composition satisfies:

$$S \circ T = \{p \wedge (p' \circ f)\} \circ [f' \circ f] \tag{2}$$

(2) states that input x is legal for $S \circ T$ if x is legal for S and the output of S, $f(x)$, is legal for T, i.e., $(p \wedge (p' \circ f))(x) = p(x) \wedge p'(f(x))$ is true. The output of $S \circ T$ is $(f' \circ f)(x) = f'(f(x))$.

For two MPTs $S : \mathsf{Pred}(Y) \to \mathsf{Pred}(X)$ and $T : \mathsf{Pred}(Y') \to \mathsf{Pred}(X')$, their parallel composition is the MPT $S \parallel T : \mathsf{Pred}(Y \times Y') \to \mathsf{Pred}(X \times X')$. If $S = \{p\} \circ [f]$ and $T = \{p'\} \circ [f']$ are functional predicate transformers, then it can be shown that their parallel composition satisfies:

$$S \parallel T = \{x, x' : p(x) \wedge p(x')\} \circ [x, x' \rightsquigarrow f(x), f'(x')] \tag{3}$$

(3) states that input (x, x') is legal for $S \parallel T$ if x is a legal input for S and x' is a legal input for T, and that the output of $S \parallel T$ is the pair $(f(x), f'(x'))$.

For $S : \mathsf{Pred}(U \times Y) \to \mathsf{Pred}(U \times X)$ as in Fig. 4a, the feedback of S, denoted $\mathsf{feedback}(S) : \mathsf{Pred}(Y) \to \mathsf{Pred}(X)$ is obtained by connecting output v to input u (Fig. 4b). The $\mathsf{feedback}$ operator that we use in this paper is a simplified version of the one defined in [20]. It is specifically designed for a component S having the structure shown in Fig. 4c, i.e., where the first output v depends only on the second input x. We call such components *decomposable*. The result of applying feedback to a decomposable block is depicted in Fig. 4d.

If S is a decomposable functional predicate transformer, i.e., if $S = \{p\} \circ [u, x \rightsquigarrow f'(x), f(u, x)]$, then it can be shown that $\mathsf{feedback}(S)$ is functional and it satisfies:

$$\mathsf{feedback}(S) = \{x : p(f'(x), x))\} \circ [x \rightsquigarrow f(f'(x), x)] \tag{4}$$

That is, input x is legal for the feedback if $p(f'(x), x)$ is true, and the output for x is $f(f'(x), x)$.

The fact that the diagram is algebraic-loop-free implies that whenever we attempt to compute $\mathsf{feedback}(S)$, S is guaranteed to be decomposable. However,

[6] Available at: http://users.ics.aalto.fi/iulia/sim2isa.shtml.

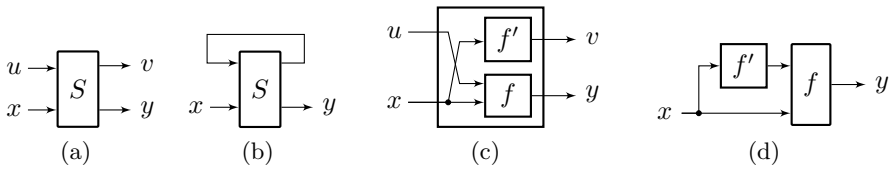


Fig. 4: (a) MPT S, (b) $\mathsf{feedback}(S)$, (c) decomposable S, (d) feedback of (c).

we only know that $S = \{p\} \circ [h]$ for some p and h, and we do not know what f and f' are. We can compute f and f' by setting $f = \mathit{snd} \circ h$, $f_0 = \mathit{fst} \circ h$, and $f'(x) = f_0(u_0, x)$ for some arbitrary fixed u_0, where *fst* and *snd* are the functions that select the first and second elements of a pair, respectively.

As an illustration of how CPTs can give semantics to HBDs, consider our running example (Fig. 3). An example mapping of the DelaySum subsystem and of the top-level Simulink model yields the following two CPTs:

$$\begin{aligned} \mathsf{DelaySum} &= \mathsf{feedback}((\mathsf{Add} \parallel \mathsf{Id}) \circ \mathsf{UnitDelay} \circ (\mathsf{Split} \parallel \mathsf{Id})) \\ \mathsf{Counter} &= (\mathsf{Const}(1) \parallel \mathsf{Id}) \circ \mathsf{DelaySum} \circ (\mathsf{Scope} \parallel \mathsf{Id}) \end{aligned} \tag{5}$$

The Id transformers in these definitions are for propagating the state introduced by the unit delay. Expanding the definitions of the basic blocks, and applying properties (2), (3), and (4), we obtain the *simplified* MPTs for the entire system:

$$\mathsf{DelaySum} = [x, s \rightsquigarrow s, s + x] \qquad \text{and} \qquad \mathsf{Counter} = [s \rightsquigarrow s, s + 1] \tag{6}$$

4.2 Translating HBDs to CPTs

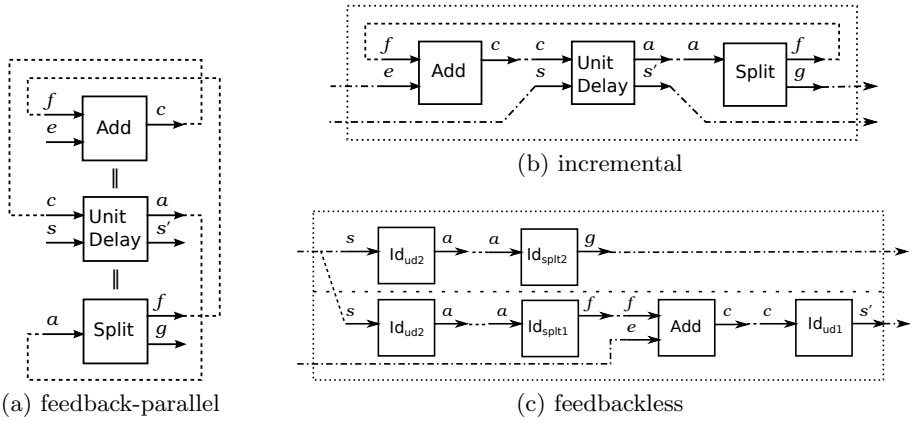
As illustrated in the introduction, the mapping from HBDs to CPTs is not unique: for a given HBD, there are many possible CPTs that we could generate. Although these CPTs are semantically equivalent, they have different *simplifiability* properties (see §4.3 and §5). Therefore, the problem of how exactly to map a HBD to a CPT is interesting both from a theoretical and from a practical point of view. In this section, we describe three different translation strategies.

In what follows, we describe how a *flat* (non-hierarchical), *connected* diagram is translated. If the diagram consists of many disconnected "islands", we can simply translate each island separately. Hierarchical diagrams are translated bottom-up: we first translate the subsystems, then their parent, and so on.

Feedback-parallel translation. The *feedback-parallel translation* strategy (FPT) first composes all components in parallel, and then connects outputs to inputs by applying feedback operations. FPT is illustrated in Fig. 5a, for the DelaySum component of Fig. 3b. The Split MPT models the junction after link a.

Applying FPT on the DelaySum diagram yields the following CPT:

$$\begin{aligned} \mathsf{DelaySum} = \mathsf{feedback}^3([f, c, a, e, s \rightsquigarrow f, e, c, s, a] \\ \circ (\mathsf{Add} \parallel \mathsf{UnitDelay} \parallel \mathsf{Split}) \circ [c, a, s', f, g \rightsquigarrow f, c, a, s', g]) \end{aligned}$$



(a) feedback-parallel (b) incremental (c) feedbackless

Fig. 5: Translation strategies for the DelaySum subsystem of Fig. 3b.

where $\mathsf{feedback}^3(\cdot) = \mathsf{feedback}(\mathsf{feedback}(\mathsf{feedback}(\cdot)))$ denotes application of the feedback operator 3 times, on the variables f, c, and a, respectively (recall that feedback works only on one variable at a time, the first input and first output of the argument transformer). In order to apply $\mathsf{feedback}^3$ to the parallel composition $\mathsf{Add} \parallel \mathsf{UnitDelay} \parallel \mathsf{Split}$, we first have to reorder its inputs and outputs, such that the variables on which the feedbacks are applied come first in matching order. This is achieved by the *rerouting* transformers $[f, c, a, e, s \rightsquigarrow f, e, c, s, a]$ and $[c, a, s', f, g \rightsquigarrow f, c, a, s', g]$.

Incremental translation. The *incremental translation* strategy (IT) composes components one by one, after having ordered them in topological order according to the dependencies in the diagram. When composing A with B, a decision procedure determines which composition operator(s) should be applied, based on dependencies between A and B. If A and B are not connected, parallel composition is applied. Otherwise, serial composition is used, possibly together with feedback if necessary.

The IT strategy is illustrated in Fig. 5b. First, topological sorting yields the order Add, UnitDelay, Split. So IT first composes Add and UnitDelay. Since the two are connected with c, serial composition is applied, obtaining the CPT

$$\mathsf{ICC1} = (\mathsf{Add} \parallel \mathsf{Id}) \circ \mathsf{UnitDelay}$$

As in the example in the introduction, Id is used here to match the number of outputs of Add with the number of inputs of UnitDelay.

Next, IT composes ICC1 with Split. This requires both serial composition and feedback, and yields the final CPT:

$$\mathsf{DelaySum} = \mathsf{feedback}(\mathsf{ICC1} \circ (\mathsf{Split} \parallel \mathsf{Id}))$$

It is worth noting that composing systems incrementally in this way might result in not the most natural compositions. For example, consider the diagram from Fig. 6. The "natural" CPT for this diagram is probably:

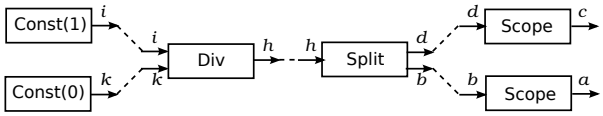


Fig. 6: Diagram ConstDiv.

$(\mathsf{Const}(1) \parallel \mathsf{Const}(0)) \circ \mathsf{Div} \circ \mathsf{Split} \circ (\mathsf{Scope} \parallel \mathsf{Scope})$. Instead, IT generates the following CPT: $(\mathsf{Const}(1) \parallel \mathsf{Const}(0)) \circ \mathsf{Div} \circ \mathsf{Split} \circ (\mathsf{Scope} \parallel \mathsf{Id}) \circ (\mathsf{Id} \parallel \mathsf{Scope})$. More sophisticated methods may be developed to extract parallelism in the diagram and avoid redundant Id compositions like in the above CPT. This study is left for future work.

Feedbackless translation. Simplifying a CPT which contains feedback operators involves performing decomposability tests, function compositions which include variable renamings, and other computations which turn out to be resource consuming (see §5). For reasons of scalability, we would therefore like to avoid feedback operators in the generated CPTs. The *feedbackless translation* strategy (NFBT) avoids feedback altogether, provided the diagram is algebraic-loop-free. The key idea is that, since the diagram has no algebraic loops, we should be able to eliminate feedback and replace it with direct operations on current- and next-state variables, just like with basic blocks. In particular, we can *decompose* UnitDelay into two Id transformers, denoted $\mathsf{Id}_{\mathsf{ud1}}$ and $\mathsf{Id}_{\mathsf{ud2}}$: $\mathsf{Id}_{\mathsf{ud1}}$ computes the next state from the input, while $\mathsf{Id}_{\mathsf{ud2}}$ computes the output from the current state.

Generally, we decompose all components having multiple outputs into several components having each a single output. For each new component we keep only the inputs they depend on, as shown in Fig. 5c. Thus, the Split component from Fig. 5b is also divided into two Id components, denoted $\mathsf{Id}_{\mathsf{splt1}}$ and $\mathsf{Id}_{\mathsf{splt2}}$.

Decomposing into components with single outputs allows to compute a separate CPT for each of the outputs. Then we take the parallel composition of these CPTs to form the CPT of the entire diagram. Doing so on our running example, we obtain:

$$\mathsf{DelaySum} = [s, e \rightsquigarrow s, s, e] \circ \Big(\big(\mathsf{Id}_{\mathsf{ud2}} \circ \mathsf{Id}_{\mathsf{splt2}}\big) \parallel \big(((\mathsf{Id}_{\mathsf{ud2}} \circ \mathsf{Id}_{\mathsf{splt1}}) \parallel \mathsf{Id}) \circ \mathsf{Add} \circ \mathsf{Id}_{\mathsf{ud1}}\big) \Big)$$

Because $\mathsf{Id}_{\mathsf{ud1}}$, $\mathsf{Id}_{\mathsf{ud2}}$, $\mathsf{Id}_{\mathsf{splt1}}$ and $\mathsf{Id}_{\mathsf{splt2}}$ are all Ids, and $\mathsf{Id} \circ A = A \circ \mathsf{Id} = A$ and $\mathsf{Id} \parallel \mathsf{Id} = \mathsf{Id}$ (thanks to polymorphism), this CPT is reduced to $\mathsf{DelaySum} = [s, e \rightsquigarrow s, s, e] \circ (\mathsf{Id} \parallel \mathsf{Add})$. Our tool directly generates this simplified CPT.

4.3 Simplifying CPTs and Checking Compatibility

Once a set of CPTs has been generated, they can be subjected to various static analysis and verification tasks. Currently, our toolset mainly supports static *compatibility* checks, which amount to checking whether any CPT obtained from the diagram is equivalent to the MPT $\mathsf{Fail} = \{x : \mathsf{false}\}$. Fail corresponds to an invalid component, indicating that the composition of two or more blocks in the diagram is illegal [23,19].

Compatibility checking is not a trivial task. Two steps are performed in order to check whether a certain CPT is equivalent to Fail: the CPT is (1) *expanded*, and (2) *simplified*. By *expansion* we mean replacing the serial, parallel, and feedback composition operators by their definitions (2), (3), (4). As a result of expansion, the CPT is turned into an MPT of the form $\{p\} \circ [f]$. By *simplification* we mean simplifying the formulas p and f, e.g., by eliminating internal variables.

5 Implementation and Evaluation

Our framework has been implemented and is publicly downloadable from `http://users.ics.aalto.fi/iulia/sim2isa.shtml`. The implementation consists of two components: (1) the SIMULINK2ISABELLE compiler, which takes as input Simulink diagrams and translates them into CPTs, using the strategies described in §4.2; and (2) an implementation of the theory of CPTs, together with simplification strategies and static analysis checks such as compatibility checks, in the Isabelle theorem prover. In this section we present the toolset and report evaluation and analysis results on several case studies, including an industrial-grade benchmark by Toyota [10,11].

5.1 Toolset

SIMULINK2ISABELLE, written in Python, takes as input Simulink files in XML format and produces valid Isabelle theories that can be subjected to compatibility checking and verification. The compiler currently handles a large subset of Simulink's blocks, including math and logical operators, continuous, discontinuous and discrete blocks, as well as sources, sinks, and subsystems (including enabled and switch case action subsystems). This subset is enough to express industrial-grade models such as the Toyota benchmarks.

During the parsing and preprocessing phase of the input Simulink file, the tool performs a set of checks, including algebraic loop detection, unsupported blocks and/or block parameters, malformed blocks (e.g., a function block referring to a nonexistent input), etc., and issues possible warnings/errors.

SIMULINK2ISABELLE implements all three translation strategies as options `-fp`, `-ic` and `-nfb`, and also takes two additional options: `-flat` (flatten diagram) and `-io` (intermediate outputs, applicable to `-ic`). All options apply to any Simulink model. Option `flat` flattens the hierarchy of the HBD and produces a single diagram consisting only of basic blocks (no subsystems), on which the translation is then applied. Option `io` generates and names all intermediate CPTs produced during the translation process. These names are then used in the CPT for the top-level system, to make it shorter and more readable. In addition, the intermediate CPTs can be expanded and simplified incrementally by Isabelle, and used in their simplified form when computing the CPT for the next level up. This generally results in more efficient simplification. Another benefit of producing intermediate CPTs is the detection of incompatibilities early during the simplification phase. Moreover, this indicates the group of components at fault and helps localize the error.

The second component of our toolset includes a complete implementation of the theory of MPTs and CPTs in Isabelle. In addition, we have implemented in Isabelle a set of functions (keyword **simulink**) which perform expansion and simplification automatically in the generated CPTs, and also generate automatically proved theorems of the simplified formulas. After expansion and simplification, we obtain for the top-level system a single MPT referring only to the external input, output, and state/next state variables of the system (and not to the internal links of the diagram).

For instance, when executed on our running example (Fig. 3) with the IT option, the tool produces the Isabelle code:

$$\textbf{simulink}\ \mathsf{DelaySum} = \mathsf{feedback}((\mathsf{Add} \parallel \mathsf{Id}) \circ \mathsf{UnitDelay} \circ (\mathsf{Split} \parallel \mathsf{Id}))$$
$$\textbf{simulink}\ \mathsf{Counter} = (\mathsf{Const}(1) \parallel \mathsf{Id}) \circ \mathsf{DelaySum} \circ (\mathsf{Scope} \parallel \mathsf{Id})$$

When executed in Isabelle, this code automatically generates the definitions (5) as well as the simplification theorems (6), and automatically proves these theorems. Note that the simplification theorems also contain the final MPT for the entire system. In general, when the diagram contains continuous-time blocks such as Integrator, the final simplified MPT will be parameterized by Δt.

As another example, when we run the tool on the example of Fig. 6, we obtain the theorem $\mathsf{ConstDiv} = \mathsf{Fail}$, which states that the system has no legal inputs. This reveals the incompatibility due to performing a division by zero.

5.2 Evaluation

We evaluated our toolset on several case studies, including the Foucault pendulum, house heating and anti-lock braking systems from the Simulink examples library.[7] Due to space limitations, we only present here the results obtained on the running example (Fig. 3) and the Fuel Control System (FCS) model described in [10,11]. FCS solves the problem of maintaining the ratio of air mass and injected fuel at the stoichiometric value [5], i.e., enough air is provided to completely burn the fuel in a car's engine. This control problem has important implications on lowering pollution and improving engine performance. Three designs are presented in [10,11], all modeled in Simulink, but differing in their complexity. The first model is the most complex, incorporating already available subsystems from the Simulink library. The second and third models represent abstractions of this main design, but they are still complicated for verification purposes. The second model is formalized as Hybrid I/O Automata [15], while the third is presented as Polynomial Hybrid I/O Automata [8]. We evaluate our approach on the third model designed with Simulink, available from http://cps-vo.org/group/ARCH/benchmarks. This model has a 3-level hierarchy with a total of 104 block instances (97 basic blocks and 7 subsystems), and 101 connections, of which 8 feedbacks.

First, we run all three translation strategies on each model using the SIMULINK2ISABELLE compiler. Then, we expand/simplify the CPTs within Isabelle and at the same time check for incompatibilities. The translation strategies

[7] http://se.mathworks.com/help/simulink/examples.html

		FPT		IT				NFBT
		HBD	FBD	HBD	FBD	IO-HBD	IO-FBD	
Translation	$\mathcal{T}_{\text{trans}}$	0.082	0.093	0.081	0.087	0.081	0.085	0.096
	$\mathcal{L}_{\text{cpt}}$	722	629	1131	1246	1146	1134	1159
	$\mathcal{N}_{\text{cpt}}$	10	9	10	9	14	14	15
Expans., simplif., and compatibility check	$\mathcal{T}_{\text{simp}}$	0.596	0.575	0.184	0.225	0.240	0.279	0.214
	$\mathcal{P}_{\text{simp}}$	0.006	0.005	0.005	0.006	0.006	0.007	0.006

Table 1: Experimental results for the running example (Fig. 3).

		FPT		IT				NFBT
		HBD	FBD	HBD	FBD	IO-HBD	IO-FBD	
Translation	$\mathcal{T}_{\text{trans}}$	0.249	0.329	0.213	0.222	0.220	0.260	0.605
	$\mathcal{L}_{\text{cpt}}$	18895	17432	87006	116550	86318	108001	46863
	$\mathcal{N}_{\text{cpt}}$	127	120	127	120	236	236	269
Expansion, simplification, and compatibility check	$\mathcal{T}_{\text{simp}}$	894.472	3471.317	617.873	2439.229	267.052	417.05	57.425
	$\mathcal{P}_{\text{simp}}$	7.144	7.267	7.856	7.161	6.742	6.228	5.18
	$\mathcal{L}_{\text{simp}}$	158212	158212	157791	157797	127132	127642	122001

Table 2: Experimental results for the FCS model.

are run with the following options: FPT without/with flattening (HBD/FBD), IT without/with flattening and without/with `io` option (IO), and NFBT. NFBT by construction generates intermediate outputs and does not preserve the structure of the hierarchy in the result, thus, its result is identical with/without the options. The results from the running example are shown in Table 1 and from the FCS model in Table 2.

The notations used in the tables are as follows: (1) $\mathcal{T}_{\text{trans}}$: time to generate the Isabelle CPTs from the Simulink model, (2) $\mathcal{L}_{\text{cpt}}$: length of the produced CPTs (# characters), (3) $\mathcal{N}_{\text{cpt}}$: number of generated CPTs, (4) $\mathcal{T}_{\text{simp}}$: total time needed for expansion and simplification, (5) $\mathcal{P}_{\text{simp}}$: time to print the simplified formula, (6) $\mathcal{L}_{\text{simp}}$: length of the simplified formula (# chars). All times are in seconds. We report separately the time to print the final formulas ($\mathcal{P}_{\text{simp}}$), since printing takes significant time in the Isabelle/ML framework.

Let us now focus on Table 2 since it contains the most relevant results due to the size and complexity of the system. Observe that the translation time ($\mathcal{T}_{\text{trans}}$) is always negligible compared to the other times.[8] Also, NFBT generates the most CPTs, which are relatively short compared to the other translations. This is one of the reasons why CPTs produced by NFBT are easier to expand/simplify than those produced by the other methods. The other and main reason is that applying the feedback operator requires identifying f and f', computing several function compositions, etc. We note that a Simulink feedback connection can transfer an array of n values, which is translated by our tool as n successive applications of feedback.

[8] $\mathcal{T}_{\text{trans}}$ for NFBT is almost twice larger than for FPT, IT and IT-IO. The reason is that NFBT executes extra steps, such as splitting blocks with multiple outputs and removing CPTs that are not used in calculating the system's output.

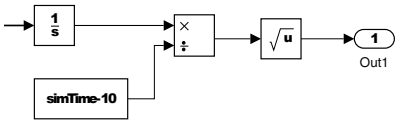


Fig. 7: Part of the FCS model: setting *simTime* to 10 results in incompatibility.

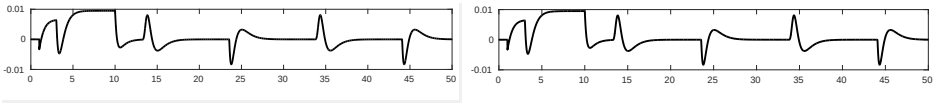
Readability. An important aspect of the produced CPTs is their *readability*. Defining quantitative readability measures such as number, length, nesting depth, etc., of the generated CPTs, is beyond the scope of this paper. Nevertheless, we can make the following (subjective) observations: (1) IT with option `-io` improves readability as the intermediate outputs allow to parse the result step by step. (2) NFBT reduces readability because this method decomposes blocks and does not preserve the hierarchy of the original model.

Equivalence of the different translations. One interesting question is whether the different translation options generate equivalent CPTs. Proving a meta-theorem stating that this is indeed the case for every diagram is beyond the scope of this paper, and part of future work. Nevertheless, we did prove that in the case of the FCS model, the final simplified MPTs resulting from each translation method are all equivalent. These proofs have been conducted in Isabelle.

Analysis. Our tool proves that the final simplified MPT of the entire FCS model is not Fail. This proves compatibility of the components in the FCS model. The obtained MPT is functional, i.e., has the form $\{p\} \circ [f]$. Its assert condition p states that the state value of an integrator which is fed into a square root is ≥ 0. We proved in Isabelle that this holds for all $\Delta t > 0$. Therefore, compatibility holds independently of the value of the time step.

We also introduced a fault in the FCS model on purpose, to demonstrate that our tool is able to detect the error. Consider the model fragment depicted in Fig. 7. If the constant *simTime* is mistakenly set to 10, the model contains a division by zero. Our tool catches this error during compatibility checking, in 51.71 seconds total (including NFBT translation, expansion, and simplification, which results in Fail).

Comparison with Simulink. In this work we give semantics of Simulink diagrams in terms of CPTs. One question that may be raised is how the CPT semantics compares to Simulink's own semantics, i.e., to "what the simulator does". Our toolset includes an option to generate simulation code (in Python) from the Isabelle CPTs. Then, we can compare the simulation results obtained from Simulink to those obtained from the CPT-generated simulation code. We performed this comparison for the FCS model: the results are shown in Fig. 8. Since the FCS model is closed (i.e., has no external inputs) and deterministic, it only has a single behavior. Therefore, we only generate one simulation plot for each method. The plot from Fig. 8a is obtained with variable step and the ode45 (Dormand-Prince) solver. The difference between the values computed by Simulink and our simulation ranges from 0 to 6.1487e-05 (in absolute value) for this solver. Better results can be obtained by reducing the step length. For instance, a step of 5e-05 gives an error difference of 2.0354e-06.



(a) Simulink simulation (b) CPT simulation

Fig. 8: Simulation plots obtained from Simulink and the simplified CPT for a 50s time interval and $\Delta t = 0.001$.

6 Related Work

A plethora of work exists on translating Simulink models to various target languages, for verification purposes or for code-generation purposes. Primarily focusing on verification and targeting discrete-time fragments of Simulink, existing works describe translations to BIP [22], NuSMV [17], or Lustre [24]. Other works study transformation of continuous-time Simulink to Timed Interval Calculus [4], Function Blocks [25], I/O Extended Finite Automata [26], or Hybrid CSP [27], and both discrete and continuous time fragments to SpaceEx Hybrid Automata [18]. The Stateflow module of Simulink, which allows to model hierarchical state machines, has been the subject of translation to hybrid automata [1,16].

Contract-based frameworks for Simulink are described in [3,21]. [3] uses pre/post-conditions as contracts for discrete-time Simulink blocks, and SDF graphs [12] to represent Simulink diagrams. Then sequential code is generated from the SDF graph, and the code is verified using traditional refinement-based techniques [2]. In [21] Simulink blocks are annotated with rich types (separate constraints on inputs and outputs, but no relations between inputs and outputs which is possible in our framework). Then the SimCheck tool extracts verification conditions from the Simulink model and the annotations, and submits them to an SMT solver for verification.

Our work offers a compositional framework which allows compatibility checks and refinement, which is not supported in the above works. We also study different translation strategies from HBDs to an algebra with serial, parallel, and feedback composition operators, which, to the best of our knowledge, have not been previously studied.

In [9], the authors propose an n-ary parallel composition operator for the Lotos process algebra. Their motivation, namely, that there may be several different process algebra terms representing a given process network, is similar to ours. But their solution (the n-ary parallel composition operator) is different from ours. Their setting is also different from ours, and results in some significantly different properties. For instance, they identify certain process networks which cannot be expressed in Lotos. In our case, *every* HBD can be expressed as a CPT (this includes HBDs with algebraic loops, even though we do not consider these in this paper).

Modular code generation methods for Simulink models are described in [14,13]. The main technical problem solved there is how to cluster subsystems in as few clusters as possible without introducing false input-output dependencies.

7 Conclusion

In this paper we present a compositional semantics and analysis framework for hierarchical block diagrams such as those found in Simulink and similar tools. Our contributions are the following: (1) semantics of basic Simulink blocks (both stateless and stateful) as atomic monotonic predicate transformers; (2) compositional semantics of HBDs as composite MPTs; (3) three translation strategies from HBDs to CPTs, implemented in the SIMULINK2ISABELLE compiler; (4) the theory of CPTs, along with expansion and simplification methods, implemented in Isabelle; (5) automatic static analysis (compatibility checks) implemented in Isabelle; and (6) proof of concept and evaluation of the framework on a real-life Simulink model from Toyota. Our approach enables compositional and correct-by-construction system design. The top-level MPT, which can be viewed as a formal interface or contract for the overall system, is automatically generated. Moreover, it is formally defined and checked in Isabelle (the theorems are also automatically generated and proved).

As future work, the current code generation process, used to compare the Isabelle code to the Simulink code via simulation, could be extended to also generate proof-carrying, easier-to-certify embedded code, from the Isabelle theories. Other future work directions include: (1) studying other translation strategies; (2) improving the automated simplification methods within Isabelle or other solvers; (3) extending the toolset with automatic verification methods (proving requirements against the top-level MPT); and (4) extending the toolset with fault localization methods whenever the compatibility or verification checks fail.

References

1. A. Agrawal, G. Simon, and G. Karsai. Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. *Electronic Notes in Theoretical Computer Science*, 109:43 – 56, 2004.
2. R.-J. Back and J. von Wright. *Refinement Calculus. A systematic Introduction.* Springer, 1998.
3. P. Boström. Contract-Based Verification of Simulink Models. In S. Qin and Z. Qiu, editors, *ICFEM*, volume 6991 of *LNCS*, pages 291–306. Springer, 2011.
4. C. Chen, J. S. Dong, and J. Sun. A formal framework for modeling and validating Simulink diagrams. *Formal Aspects of Computing*, 21(5):451–483, 2009.
5. J. A. Cook, J. Sun, J. H. Buckland, I. V. Kolmanovsky, H. Peng, and J. W. Grizzle. Automotive Powertrain Control – A Survey. *Asian Journal of Control*, 8(3):237–260, 2006.
6. E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457, 1975.
7. I. Dragomir, V. Preoteasa, and S. Tripakis. Translating hierarchical block diagrams into composite predicate transformers. *CoRR*, abs/1510.04873, 2015.
8. G. Frehse, Z. Han, and B. Krogh. Assume-guarantee reasoning for hybrid I/O-automata by over-approximation of continuous interaction. In *CDC*, pages 479–484, 2004.
9. H. Garavel and M. Sighireanu. A graphical parallel composition operator for process algebras. In *FORTE XII*, volume 156 of *IFIP Conference Proceedings*, pages 185–202. Kluwer, 1999.

10. X. Jin, J. Deshmukh, J. Kapinski, K. Ueda, and K. Butts. Benchmarks for model transformations and conformance checking. In *ARCH*, 2014.
11. X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts. Powertrain Control Verification Benchmark. In *HSCC*, pages 253–262. ACM, 2014.
12. E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
13. R. Lublinerman, C. Szegedy, and S. Tripakis. Modular code generation from synchronous block diagrams – modularity vs. code size. In *POPL*, pages 78–89. ACM, Jan. 2009.
14. R. Lublinerman and S. Tripakis. Modularity vs. reusability: Code generation from synchronous block diagrams. In *DATE*, pages 1504–1509. ACM, Mar. 2008.
15. N. Lynch, R. Segala, and F. Vaandrager. Hybrid I/O Automata. *Inf. Comput.*, 185(1):105–157, Aug. 2003.
16. K. Manamcheri, S. Mitra, S. Bak, and M. Caccamo. A Step Towards Verification and Synthesis from Simulink/Stateflow Models. In *HSCC*, pages 317–318. ACM, 2011.
17. B. Meenakshi, A. Bhatnagar, and S. Roy. Tool for Translating Simulink Models into Input Language of a Model Checker. In *ICFEM*, volume 4260 of *LNCS*, pages 606–620. Springer, 2006.
18. S. Minopoli and G. Frehse. SL2SX Translator: From Simulink to SpaceEx Verification Tool. In *HSCC*, 2016. To appear.
19. V. Preoteasa and S. Tripakis. Refinement calculus of reactive systems. In *EMSOFT*, pages 1–10, Oct 2014.
20. V. Preoteasa and S. Tripakis. Towards compositional feedback in non-deterministic and non-input-receptive systems. *CoRR*, abs/1510.06379, 2015.
21. P. Roy and N. Shankar. SimCheck: a contract type system for Simulink. *Innovations in Systems and Software Engineering*, 7(2):73–83, 2011.
22. V. Sfyrla, G. Tsiligiannis, I. Safaka, M. Bozga, and J. Sifakis. Compositional translation of simulink models into synchronous BIP. In *SIES*, pages 217–220, July 2010.
23. S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee. A Theory of Synchronous Relational Interfaces. *ACM Trans. Program. Lang. Syst.*, 33(4):14:1–14:41, July 2011.
24. S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating Discrete-time Simulink to Lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4):779–818, Nov. 2005.
25. C. Yang and V. Vyatkin. Transformation of Simulink models to IEC 61499 Function Blocks for verification of distributed control systems. *Control Engineering Practice*, 20(12):1259–1269, 2012.
26. C. Zhou and R. Kumar. Semantic Translation of Simulink Diagrams to Input/Output Extended Finite Automata. *Discrete Event Dynamic Systems*, 22(2):223–247, 2012.
27. L. Zou, N. Zhany, S. Wang, M. Franzle, and S. Qin. Verifying Simulink diagrams via a Hybrid Hoare Logic Prover. In *EMSOFT*, pages 9:1–9:10, Sept 2013.