Type Inference of Simulink Hierarchical Block Diagrams in Isabelle

Viorel Preoteasa, Iulia Dragomir, Stavros Tripakis

Type Inference of Simulink Hierarchical Block Diagrams in Isabelle *

Viorel Preoteasa[1], Iulia Dragomir[2], and Stavros Tripakis[1,3]

[1] Aalto University, Finland
[2] Verimag, France
[3] University of California, Berkeley, USA

Abstract. Simulink is a de-facto industrial standard for embedded system design. In previous work, we developed a compositional analysis framework for Simulink, the Refinement Calculus of Reactive Systems (RCRS), which allows checking compatibility and substitutability of components. However, standard type checking was not considered in that work. In this paper we present a method for the type inference of Simulink models using the Isabelle theorem prover. A Simulink diagram is translated into an (RCRS) Isabelle theory. Then Isabelle's powerful type inference mechanism is used to infer the types of the diagram based on the types of the basic blocks. One of the aims is to handle formally as many diagrams as possible. In particular, we want to be able to handle even those diagrams that may have typing ambiguities, provided that they are accepted by Simulink. This method is implemented in our toolset that translates Simulink diagrams into Isabelle theories and simplifies them. We evaluate our technique on several case studies, most notably, an automotive fuel control system benchmark provided by Toyota.

1 Introduction

Simulink is a widespread tool from Mathworks for modeling and simulating embedded control systems. A plethora of formal verification tools exist for Simulink, both from academia and industry, including Mathwork's own Design Verifier. Formal verification is extremely important, particularly for safety critical systems. Formal verification techniques make steady progress and are increasingly gaining acceptance in the industry.

At the same time, we should not ignore more "lightweight" methods, which can also be very beneficial. In this paper, we are interested in particular in type checking and type inference. Type checking is regularly used in many programming languages, as part of compilation, and helps to catch many programming mistakes and sometimes also serious design errors. Type inference is a more advanced technique which usually includes type checking but in addition permits types to be inferred when those are not given by the user, thus automatically

* This work has been partially supported by the Academy of Finland and the U.S. National Science Foundation (awards #1329759 and #1139138).

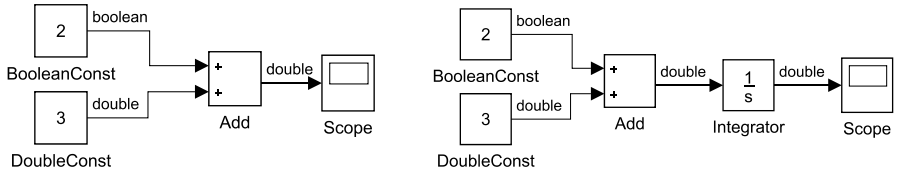


Fig. 1: Two Simulink diagrams. Both are accepted (i.e., simulated) by Simulink.

extracting valuable information about the design. Importantly, both type checking and type inference are typically much less expensive than formal verification. We therefore view both of them as complementary to formal verification for the rigorous design of safety-critical systems.

Simulink already provides some kind of type checking and inference as part of its basic functionality. In the version R2016b that we used while writing this article, the user has to open a diagram and then click on *Display* $\rightarrow$ *Signals & Ports* $\rightarrow$ *Port Data Types*, upon which Simulink (computes and) displays the typing information, for example, as shown in Fig. 1. Unfortunately, Simulink analyses are proprietary, and as such it is difficult to know what type checking and inference algorithms are used. Moreover, the way Simulink uses the typing information is often strange, as illustrated by the examples that follow.

Consider first the two diagrams shown in Fig. 1. Both these examples capture implicit type conversions performed by Simulink. In both diagrams, there are two *Constant* blocks, with values 2 and 3 respectively. In the first block, we manually set the output type to be *Boolean*. In the second block we manually set the output type to *double*. The outputs of the two constants are fed into an *Add* block which performs addition. In the rightmost diagram, the result is fed into an *Integrator*. The block *Scope* plots and displays the output over time.

Both diagrams of Fig. 1 are *accepted* by Simulink, meaning that they can be simulated. Although Simulink issues a warning that says "Parameter precision loss occurred ... A small quantization error has occurred." the results of the simulation appear as expected: a constant value 4 in the case of the leftmost diagram, and a straight slope from values 0 to 40 for the rightmost diagram, when simulated from 0 to 10 time units. Simulink performs an implicit conversion of 2 to the Boolean value *true*, and then another implicit conversion of *true* to the real value 1, in order for the addition to be performed. These implicit conversions are stipulated in the Simulink documentation (when the source block allows them). Therefore, the result is $3 + 1 = 4$.

Although these examples seem unusual, they are designed to be minimal and expose possible problems, similar to those detected in a Fuel Control System (FCS) benchmark provided by Toyota [9]. It is common practice to mix, in languages that allow it, Boolean and numeric values in a way exposed by these examples. We have tested this behavior extensively and we have observed that other languages that perform automatic conversions between Boolean and numeric values behave consistently with Simulink (e.g., C: $(\mathsf{double})3+(\mathsf{bool})2 = 4.0$, Python: $\mathsf{float}(3) + \mathsf{bool}(2) = 4.0$).

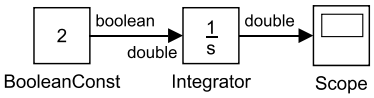


Fig. 2: A diagram rejected by Simulink.

Now, consider the diagram shown in Fig. 2, where the output of the same Boolean constant block as the one used in the previous diagrams is fed directly into the integrator. In this case, Simulink *rejects* this diagram (meaning it refuses to simulate it). It issues an error message saying: "Data type mismatch. Input of Integrator expects a signal of data type 'double'. However, it is driven by a signal of data type 'boolean'." The Integrator, as well as other block types, accepts only inputs of type double and implicit conversions (from Boolean to double or vice-versa) are not allowed and performed. We remark that Simulink does not treat diagrams in a consistent way with respect to typing. One of the goals of this paper is to present a formal type checking and inference framework for Simulink, where such examples are treated consistently (and meaningfully).

The contribution of this work is a type inference mechanism for Simulink diagrams, on top of the type inference mechanism of the Isabelle theorem prover [12]. One important feature of this approach is handling Simulink basic blocks locally, without knowledge of their environment. The challenge of this work is embedding the more relaxed type system of Simulink into the formal type system of Isabelle, while preserving the semantics, and as much typing information as possible. We apply this technique to several case studies, including the FCS benchmark.

This work is part of a larger project on translating Simulink diagrams into Isabelle theories suitable for analysis and verification [15,6,16]. Because Isabelle's language is formal and precise, we can directly obtain concise and correct code in other languages that can be used for processing Simulink models. For example, from the Isabelle model we easily obtain Python code for simulations, and Z3 SMT solver [4] model for automatically checking properties.

Our techniques apply to the entire Simulink language, provided we know how to translate basic blocks. Simulink contains many basic blocks but all of them fall into some of the categories discussed in this paper.

2 Related Work

The verification of Simulink diagrams has been extensively studied in the literature, by proposing model transformations of Simulink diagrams to a formal framework. Formal frameworks include Hybrid Automata [1], BIP [19], NuSMV [10], Boogie [17], Timed Interval Calculus [2], Function Blocks [24], I/O Extended Finite Automata [25], Hybrid CSP [26], and SpaceEx [11]. Many of the target formalisms define a typing feature, and the proposed model translations make use of it: a basic block is mapped to some "expression" on inputs and outputs, where the types of inputs and outputs are dependent of the block type. The static type checking is then delegated to the target framework, if such func-

tionality is available. However, these studies mostly aim for formal verification of Simulink diagrams and do not report about type checking.

The most relevant work with respect to type checking Simulink diagrams is described in [23] and [18]. [23] presents a translation from discrete-time Simulink to Lustre, where the type system of Simulink is formalized as a simple polymorphic type system and unification is used to infer types. It is unclear how the above type system handles the subtleties studied in this paper. [18] presents the SimCheck framework, which among other functions allows the user to annotate ports and wires with types and also units (e.g., *cm*). A translation to Yices [7] supports the automated static and behavioral type checking. In contrast to SimCheck, we automatically infer the types and dimensions of signals from the Simulink diagrams, but we do not infer or check for physical units.

In previous work, we have presented the *Refinement Calculus of Reactive Systems* (RCRS) [15,6], a compositional framework for static analysis of hierarchical block diagrams in general, and Simulink models in particular. In the RCRS framework blocks are specified syntactically by general formulas (*contracts*) on input, output, and state variables. These contracts are then composed using serial, parallel and feedback composition operators. Such contracts can be seen as richer types, and the compatibility and contract synthesis methods developed in RCRS can be seen as type checking and type inference techniques. However, the contracts considered in RCRS are much more powerful than the types considered in this paper, and the compatibility and synthesis algorithms of RCRS are much more expensive (requiring in general quantifier elimination and satisfiability checking in expressive logics). Therefore, the framework proposed in this paper is much more lightweight.

In this work we use the Isabelle theorem prover which has a standard type inference mechanism [3], briefly discussed in Section 3.1. Our goal is to give an embedding of Simulink into a language and framework suitable for further processing (simplifications, checking of properties, and even simulation). Other systems for logical reasoning (e.g., PVS [13], Z3 [4], Coq [20]) could also be used for this purpose. As we use type inference, our work cannot be directly transferred to systems that do not have it (PVS, Z3). Translations of Simulink diagrams into systems with proper subtyping (PVS, Coq) need also different treatment since in these systems typing of a term is not always decidable.

We do not use type coercions (implicite type conversions) in our approach. We encode possible coercions explicitely in the representations of basic blocks.

3 Preliminaries

3.1 Isabelle

Isabelle/HOL is an interactive theorem prover based on higher order logic. Isabelle provides an environment which consists of a powerful specification and proving language and it has a rich theory library of formally verified mathematics. Notable features of Isabelle include a type system with type inference, polymorphism and overloading, and axiomatic type classes.

Isabelle's type system includes the basic types $\mathsf{bool}, \mathsf{real}, \mathsf{int}, \mathsf{nat}$, type variables $'a$, $'b$, etc., and predefined type constructors $'a \to 'b$ (functions from $'a$ to $'b$) and $'a \times 'b$ (Cartesian product of $'a$ and $'b$). *Type expressions* are build from basic types and type variables using the type constructors. For term $f(x, g(y))$ we can specify that it has a type t by using $: t$ after the term $f(x, g(y)) : t$.

Definitions in Isabelle are introduced using declarations of the form

$$\mathsf{definition}\ f(x)(y)(g) = g(x)(y).$$

This definition introduces a function $f : 'a \to 'b \to ('a \to 'b \to 'c) \to 'c$, and Isabelle uses the *type inference* mechanism to deduce its type. The type of f is the *most general type* such that the expression $f(x)(y)(g) = g(x)(y)$ is *well typed*. A type t is *more general* than a type t' if t' can be obtained from t by instantiating the type variables in t with some type expressions [12].

We can also use specific types in definitions:

$$\mathsf{definition}\ h(x : \mathsf{real})(y)(g) = g(x)(y)$$

In our translation of Simulink to Isabelle we use the type inference mechanism.

Another important feature of Isabelle that we use is the type classes [8]. This is a mechanism that can be used, for example, to overload a polymorphic function $+ : 'a \to 'a \to 'a$ on different types for $'a$.

$\mathsf{class\ plus} =$
  $\mathsf{fixes} + : 'a \to 'a \to 'a$

$\mathsf{instantiation\ real : plus}$
  $\mathsf{definition}\ x + y = \ldots$

$\mathsf{instantiation\ nat : plus}$
  $\mathsf{definition}\ 0 + x = x \mid \mathsf{Suc}(x) + y = \mathsf{Suc}(x + y)$

We define the type class plus with the constant $+$ of polymorphic type $'a \to 'a \to 'a$, and two instantiations to natural and real numbers. In a term $x + y$, the type of x and y is not just a type variable $'a$, but a type variable $'a$ of class plus. This is represented syntactically as $x : 'a : \mathsf{plus}$. The terms $(x : \mathsf{nat}) + y$ and $(x : \mathsf{real}) + y$ are well typed because the types nat and real are defined as instances of plus. Moreover, in the term $(x : \mathsf{nat}) + y$, the plus operator is the one defined in the instance of $\mathsf{nat : plus}$, while $x + y$ does not in general have a definition. The term $(x : \mathsf{bool}) + y$ is not well typed because bool is not defined as an instance of plus.

3.2 Representation of Simulink Diagrams as Predicate Transformers

A (fragment of a) Simulink diagram is modeled intuitively as a discrete symbolic transition system with input, output, current and next state. The intuition behind this representation is the following. Initially, the current state has a default value. The system representation works in discrete steps, and, at each step, it updates the output and the next state based on the input and the current state.

For example, an integrator block like the one from Fig. 1 is discretized as a system parameterized by $dt > 0$, with input x and current state s, and output $y := s$ and next state $s' := s + x \cdot dt$.

Formally, we model these systems in Isabelle as *monotonic predicate transformers* [5], mapping predicates (sets) over the output and next state into predicates (sets) over the input and current state. A monotonic predicate transformer S with input x, current state s, output y and next state s', for a set q of pairs (y, s'), $S(q)$ returns the set of all pairs (x, s) such that if the execution of S starts in (x, s) then S does not fail and results in a pair $(y, s') \in q$. A detailed discussion of the choice for this semantics is outside the scope of this paper, and is extensively presented in [15,21,6].

In Isabelle, the integrator block is represented as the predicate transformer

$$\mathsf{Integrator}(dt)(q)(x, s) = q(s, s + x \cdot dt)$$

and it has the type $'a \to ('a : \mathsf{plus} \times 'a \to \mathsf{bool}) \to ('a \times 'a \to \mathsf{bool})$. In what follows we do not make a distinction between the input and current state, and output and next state, respectively. In general, a Simulink diagram is modeled as a predicate transformer with input (and current state) of a type variable $'a$, and output (and next state) of a type variable $'b$. The type of this predicate transformer is $('b \to \mathsf{bool}) \to ('a \to \mathsf{bool})$ and we use the notation $'a \overset{\circ}{\to} 'b$ for it (this may appear reversed, but is correct and in accordance with the discussion on predicate transformers above). Often $'a$ and $'b$ will be Cartesian products, including the empty product (unit). We denote by $() : \mathsf{unit}$ the *empty tuple*.

For a predicate transformer mapping, for example, inputs (x, y, z) into output expressions $(x + y, x \cdot z)$, we use the notation $[x, y, z \rightsquigarrow x + y, x \cdot z]$ where

$$[x, y, z \rightsquigarrow x + y, x \cdot z](q)(x, y, z) = q(x + y, x \cdot z)$$

Using this notation, the constant and the integrator blocks become

$$\mathsf{Const}(a) = [() \rightsquigarrow a], \qquad \mathsf{Integrator}(dt) = [x, s \rightsquigarrow s, s + x \cdot dt].$$

We denote by Id the identity predicate transformer ($[x \rightsquigarrow x]$).

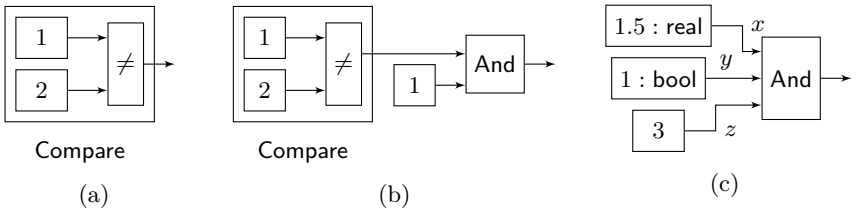
A block diagram is modeled in Isabelle as an expression of predicate transformers corresponding to the basic blocks, using three composition operators: serial ($\circ$), parallel ($\parallel$), and feedback (fb). The serial composition of predicate transformers is exactly the composition of functions. The parallel and feedback compositions are described in [6,16]. For this presentation, the typing of these operations is important. The typing of the serial composition is standard. The parallel and feedback compositions have the types:

$$\begin{aligned} \parallel &: ('a \overset{\circ}{\to} 'b) \to ('c \overset{\circ}{\to} 'd) \to ('a \times 'c \overset{\circ}{\to} 'b \times 'd), \\ \mathsf{fb} &: ('a \times 'b \overset{\circ}{\to} 'a \times 'c) \to ('b \overset{\circ}{\to} 'c) \end{aligned}$$

Using these notations, the predicate transformer for the rightmost diagram from Fig. 1 is given by

$$(((\mathsf{Const}(\mathsf{s\_bool}(2)) \parallel \mathsf{Const}(3)) \circ [x, y \rightsquigarrow x + y]) \parallel \mathsf{Id}) \circ \mathsf{Integrator}(dt)$$

We use here the Id predicate transformer to model and connect the current state of the integrator. We also use the polymorphic function $\mathsf{s\_bool}$ which for 2 returns True if the type of the result is Boolean, and 1 if the type of the result is real. The type of the result in this case is real as it is inferred from the addition block (following the constant blocks).



(a) (b) (c)

Fig. 3: (a) Comparison on constants, (b) Comparison into conjunction, (c) And on typed constants

4 Constant Blocks

Simulink diagrams may contain constant blocks, parameterized by numeric constants. These are blocks without input and with one single output which is always equal to the constant's parameter. By default, Simulink constants do not have associated types. In order to have the possibility to instantiate these types later for reals, integers, Booleans, or other types, we use uninterpreted constants. By default, numeric constants in Isabelle are polymorphic. If no type is explicitly set to a constant in a term $t = 12$, then Isabelle associates a type variable $'a : \mathsf{numeral}$ to this constant. If the term is used in a context where the type is more specific ($t = 12 \wedge \mathsf{Suc}(t) = t'$) then Isabelle uses the type class instantiation to the specific type (in this case natural because of the successor function).

Due to this polymorphic treatment of constants, in some contexts the problem arises that the types of these constants are not part of the type of the resulting predicate transformer. Consider for example the diagram from Fig. 3a. The Isabelle definition for this diagram is

$$\mathsf{Compare} = (\mathsf{Const}(1 : 'a : \mathsf{numeral}) \parallel \mathsf{Const}(2)) \circ [x, y \rightsquigarrow x \neq y] \ (= [() \rightsquigarrow 1 \neq 2])$$

In this definition $'a$ is the inferred type of constants 1 and 2. The problem with this definition is that the type $'a$ is not part of the type of $\mathsf{Compare} : \mathsf{unit} \xrightarrow{\circ} \mathsf{bool}$. If this definition was allowed, then we would have an unsound system, because for example if $'a$ is instantiated by real, then $(1 : \mathsf{real}) \neq 2$ is true and $\mathsf{Compare} = [() \rightsquigarrow \mathsf{True}]$, but if $'a$ is instantiated by unit, then $(1 : \mathsf{unit}) \neq 2$ is false (unit contains only one element) and $\mathsf{Compare} = [() \rightsquigarrow \mathsf{False}]$, and we can derive $[() \rightsquigarrow \mathsf{False}] = [() \rightsquigarrow \mathsf{True}]$ which is false. In order to instantiate unit for $'a$, the type unit must be of class $\mathsf{numeral}$. Although by default this is not the case in Isabelle, we can easily add an instantiation of unit as $\mathsf{numeral}$ and obtain this contradiction.

Isabelle allows this kind of definition, but it gives a warning message ("Additional type variable(s) in specification of $\mathsf{Compare} : 'a : \mathsf{numeral}$"), and it defines the function $\mathsf{Compare}$ to depend on an additional type variable:

$$\mathsf{Compare}('a : \mathsf{numeral}) = (\mathsf{Const}(2 : 'a) \parallel \mathsf{Const}(1)) \circ [x, y \rightsquigarrow x \neq y]$$

Now $\mathsf{Compare}(\mathsf{real})$ and $\mathsf{Compare}(\mathsf{unit})$ are different terms, so they are not equal anymore and we cannot derive $[() \rightsquigarrow \mathsf{False}] = [() \rightsquigarrow \mathsf{True}]$. Assume now that we compose the $\mathsf{Compare}$ block with a conjunction block as in Fig. 3b.

$$A = (\mathsf{Compare} \parallel \mathsf{Const}(1)) \circ \mathsf{And}$$

However, this definition is now incorrect because Compare has an additional type parameter. The correct definition would be:

$$A('a : \mathsf{numeral}) = (\mathsf{Compare}('a) \parallel \mathsf{Const}(1)) \circ \mathsf{And}$$

When we generate the definition for the diagram from Fig. 3b we do not know that Compare needs the additional type parameter. To have control over the type parameters we add them systematically for all constants occurring in the diagram. Moreover, we define the constants with a variable parameter. Due to the lack of space, the rationale for these definitions is discussed in [14].

With this method the constant blocks from Fig. 3b are defined by

$$\begin{array}{l}\mathsf{ConstA}(x : 'a) = \mathsf{Const}(1 : 'a) \;\text{ and }\; \mathsf{ConstB}(y : 'b) = \mathsf{Const}(2 : 'b) \text{ and} \\ \mathsf{ConstC}(z : 'c) = \mathsf{Const}(1 : 'c)\end{array} \tag{1}$$

and the diagram is defined by

$$A(x, y, z) = (((\mathsf{ConstA}(x) \parallel \mathsf{ConstB}(y)) \circ [x, y \rightsquigarrow x \neq y]) \parallel \mathsf{ConstC}(z)) \circ \mathsf{And} \tag{2}$$

In this approach, variables x, y, z are used only to control the types of the constants. In this definition, because outputs of ConstA and ConstB are entering the comparison block, the types of x and y are unified. If we need an instance of A for type real for constants ConstA and ConstB and type Boolean for ConstC, then we can specify it using the term $A(x : \mathsf{real}, y : \mathsf{real}, z : \mathsf{bool})$.

This definition mechanism is implemented in our Simulink to Isabelle model translator under the $-$const option. When the option is set, then the constants are defined as in (1), and the diagrams using these constants are defined as in (2). When the option is not given, then the constants are defined as in:

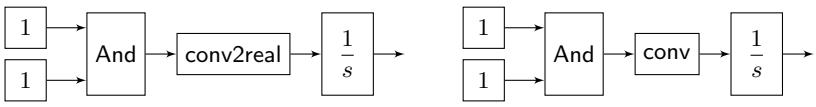
$$\mathsf{ConstA} = \mathsf{Const}(1) \;\text{ and }\; \mathsf{ConstB} = \mathsf{Const}(2) \;\text{ and }\; \mathsf{ConstC} = \mathsf{Const}(1)$$

and they are used as in: $A = (((\mathsf{ConstA} \parallel \mathsf{ConstB}) \circ [x, y \rightsquigarrow x \neq y]) \parallel \mathsf{ConstC}) \circ \mathsf{And}$. When the constant blocks in a Simulink diagram define an output type, we simply use them as in $\mathsf{Const}(1.5 : \mathsf{real})$ (Fig. 3c).

5 Conversion Blocks

Simulink diagrams may also contain conversion blocks. The type of the input of a conversion is inherited and the type of the output is usually specified (Boolean, real, ...). However we can have also situations when the output is not specified, and it is inherited from the type of the inputs of the block that follows a conversion. In Fig. 4a we illustrate an explicit conversion to real, while Fig. 4b presents an unspecified conversion.

As with the other blocks we want to define these conversions locally, without knowing the types of the inputs and outputs, when the output type is unspecified. In doing so, we use the overloading mechanism of Isabelle. Overloading is a feature that allows using the same constant name with different types. For the conversion blocks we introduce the following definitions.



(a) Conversion to real. (b) Unspecified conversion.

Fig. 4: Conversions examples.

$$\begin{aligned}&\mathsf{consts}\ \mathsf{conv} : 'a \to 'b\\ &\mathsf{overloading}\\ &\quad \mathsf{conv}(x : 'a) := x\\ &\quad \mathsf{conv}(x : \mathsf{bool}) := \mathsf{if}\ x\ \mathsf{then}\ (1 : \mathsf{real})\ \mathsf{else}\ 0\\ &\quad \mathsf{conv}(x : \mathsf{real}) := (x \neq 0)\end{aligned}$$

This definition introduces an arbitrary function conv from a type variable $'a$ to a type variable $'b$, and it also defines three overloadings for this function. The term $\mathsf{conv}(x)$ in general is of type $'b$ and x is of type $'a$. If we restrict the type $'a$ and $'b$ to real and bool, then we have

$$(\mathsf{conv}(x : \mathsf{real}) : \mathsf{bool}) = (x \neq 0)$$

When we translate a conversion block, if we know the output type, then we use the conversion restricted to this output type, otherwise we use the unrestricted conversion. For example the conversion from Fig. 4a is translated into $[x \rightsquigarrow (\mathsf{conv}(x) : \mathsf{real})]$. The entire diagram from Fig. 4a is translated into

$$(((\mathsf{Const}(1) \parallel \mathsf{Const}(1)) \circ \mathsf{And} \circ [x \rightsquigarrow (\mathsf{conv}(x) : \mathsf{real})]) \parallel \mathsf{Id}) \circ [x, s \rightsquigarrow s, s + x \cdot dt] \quad (3)$$

The identity block (Id) is used here for the current state input of the integral block. The conversion from Fig. 4b is translated into $[x \rightsquigarrow \mathsf{conv}(x)]$. This diagram becomes

$$(((\mathsf{Const}(1) \parallel \mathsf{Const}(1)) \circ \mathsf{And} \circ [x \rightsquigarrow \mathsf{conv}(x)]) \parallel \mathsf{Id}) \circ [(x : \mathsf{real}), s \rightsquigarrow s, s + x \cdot dt]$$

and compared with (3) the only difference is that in the later case, the type of the output of the conversion is not specified. However, in both cases, the inputs of the conversions must be Boolean because of the And block, and the outputs must be real because of the integral block. In both cases the translations are equivalent to $[(s : \mathsf{real}) \rightsquigarrow s, s + dt]$.

6 Boolean Blocks

Simulink Boolean blocks are also challenging to implement due to the fact that, for example, the inputs to a conjunction block could have different types (real, Boolean, unspecified), as illustrated in Fig. 3c. In languages that allow it (e.g., C, Python), it is common practice to use numerical values in Boolean expressions, with the meaning that non-zero is true. Similarly, it is common practice to use Boolean values in numeric expressions. Simulink also allows these cases, but Isabelle does not. We show in this and next section how to solve these problems.

Consider the example from Fig. 3c. If we would simply take the conjunction of all inputs

$$(\mathsf{Const}(1.5 : \mathsf{real}) \parallel \mathsf{Const}(1 : \mathsf{bool}) \parallel \mathsf{Const}(3)) \circ [x, y, z \rightsquigarrow x \wedge y \wedge z]$$

we will obtain in Isabelle a type error, because x has type real, y has type bool and z has type $'a : \mathsf{numeral}$, and their conjunction is not well typed.

To fix this typing problem, we implement the conjunction block in the following way: $\mathsf{And} = [x, y, z \rightsquigarrow (x \neq 0) \wedge (y \neq 0) \wedge (z \neq 0)]$. In this expression the types of variables x, y, and z are independent of each other, and also of the Boolean output, and they can match the types of the blocks that are input to And. There are still some details to consider. If input x is real, then $x \neq 0$ is true if and only if x is not zero, and this coincides with the semantics of And in Simulink. However, if the input y is Boolean, then the expression $y \neq 0$ is not well typed, unless we add additional class instantiation in Isabelle:

$$\begin{array}{l}\mathsf{instantiation\ bool : zero} = \\ \quad (0 : \mathsf{bool}) := \mathsf{False}\end{array}$$

Intuitively this instantiation provides the interpretation of constant 0 as False, when 0 is used as a Boolean value. With this the expression $(y : \mathsf{bool}) \neq 0$ is equivalent to $y \neq \mathsf{False}$ and it is equivalent to y. The same holds for the expression $1 : \mathsf{bool}$ which is not well typed unless we provide an instantiation of bool as numeral, where every (non-zero) numeral constant is True. These definitions formalize the behavior described by Simulink in its documentation.

Using this approach, the translation of the diagram from Fig. 3c is:

$$(\mathsf{Const}(1.5 : \mathsf{real}) \parallel \mathsf{Const}(1 : \mathsf{bool}) \parallel \mathsf{Const}(3)) \circ [x, y, z \rightsquigarrow x \neq 0 \wedge y \neq 0 \wedge z \neq 0]$$

and it is equal to

$$(\mathsf{Const}(1.5 : \mathsf{real}) \parallel \mathsf{Const}(\mathsf{True}) \parallel \mathsf{Const}(3)) \circ [x, y, z \rightsquigarrow x \neq 0 \wedge y \wedge z \neq 0]$$

because y is of type bool and $(y \neq 0) = y$. If we expand the serial composition and simplify the term, we obtain $[() \rightsquigarrow (3 : 'a : \{\mathsf{numeral}, \mathsf{zero}\}) \neq 0]$. The equality $(3 : 'a : \{\mathsf{numeral}, \mathsf{zero}\}) \neq 0$ cannot be simplified. This is because the type $'a : \{\mathsf{numeral}, \mathsf{zero}\}$ has all numeric constants $1, 2, \ldots$ (numeral) and the constant 0 (zero), but no relationship between these constants is known. If we know that we only use the type $'a$ with instances where the numeric constants $1, 2, \ldots$ are always different from 0, then we can create a new class based on numeral and zero that has also the property that $n \neq 0$ for all $n \in \{1, 2, \ldots\}$. Formally we can introduce this class in Isabelle by

$$\begin{array}{l}\mathsf{class\ numeral\_nzero} = \mathsf{zero} + \mathsf{numeral} + \\ \quad \mathsf{assume\ numeral\_nzero[simp]} : (\forall a.\mathsf{numeral}(a) \neq 0)\end{array}$$

The new class numeral_nzero contains the numeric constants $\{0, 1, \ldots\}$ but also it has the property that all numbers $1, 2, \ldots$ are different from 0 ($\forall a.\mathsf{numeral}(a) \neq 0$). In this property a ranges over the binary representations of the numbers $1, 2, \ldots$. This property is called numeral_nzero, and the [simp] declaration tells Isabelle to use it automatically as simplification rule. Now the equality $(3 : 'a : \mathsf{numeral\_nzero}) \neq 0$ is also automatically simplified to True.

We provide the following class instantiation:

$$\begin{array}{l}\mathsf{instantiation\ bool : numeral\_nzero} = \\ \quad (0 : \mathsf{bool}) := \mathsf{False} \ \mid \ (\mathsf{numeral}(a) : \mathsf{bool}) := \mathsf{True}\end{array}$$

Because in this class we have also the assumption $(\forall a.\mathsf{numeral}(a) \neq 0)$, we need to prove it, and it trivially holds because $\mathsf{False} \neq \mathsf{True}$. Similarly we need to introduce instantiations of numeral_nzero to real, integer, and natural numbers. In these cases, since real, integer, and natural are already instances of numeral and zero, we do not need to define 0 and $\mathsf{numeral}(a)$, but we only need to prove the property $(\forall a.\mathsf{numeral}(a) \neq 0)$.

With this new class, the translation of diagram from Fig. 3c becomes:

$$(\mathsf{Const}(1.5 : \mathsf{real}) \parallel \mathsf{Const}(1 : \mathsf{bool}) \parallel \mathsf{Const}(3 : 'a : \mathsf{numeral\_nzero})) \circ [x, y, z \rightsquigarrow x \neq 0 \wedge y \neq 0 \wedge z \neq 0]$$

Because of the properties of types real, bool, and $'a$: numeral_nzero, it is equal to

$$(\mathsf{Const}(1.5 : \mathsf{real}) \parallel \mathsf{Const}(\mathsf{True}) \parallel \mathsf{Const}(3 : 'a : \mathsf{numeral\_nzero})) \circ [x, y, z \rightsquigarrow x \neq 0 \wedge y \wedge z \neq 0]$$

and, after expanding the serial composition and symplifying the term, we obtain $[() \rightsquigarrow \mathsf{True}]$.

Although the translation of Boolean blocks is rather involved, the result obtained especially after basic Isabelle simplifications is simple and intuitive, as shown above. Moreover, for the translation of a Boolean block we do not need to consider its context, and the correctness of the translation can be assessed locally. Basically an element e in a conjunction $(e \wedge \ldots)$ is replaced by $((e \neq 0) \wedge \ldots)$. By creating the class numeral_nzero and the instantiations to bool and real, the typing of e (e : bool or e : real, ...) defines the semantics of the expression $e \neq 0$.

7 Generic Translations

The approach described so far works well for diagrams that do not mix values of different types (Boolean, real) in operations. However, there are some diagrams that are accepted by Simulink and cannot be translated with the approach described above due to type mismatch. Fig. 1 and 2 give three examples of this kind of diagrams.

Fig. 1 illustrates diagrams accepted by Simulink, while the diagram represented in Fig. 2 is not accepted by Simulink. The simulation of leftmost diagram from Fig. 1 gives 4 (2 : bool results in True, and then converted to real is 1). The rightmost diagram from Fig. 1 is equivalent to a diagram where constant 4 is input for an integral block. However none of these diagrams result in correct translations when using the method presented so far. This is due to type mismatches:

$(\mathsf{Const}(2 : \mathsf{bool}) \parallel \mathsf{Const}(3 : \mathsf{real})) \circ \mathsf{Add}$
$(((\mathsf{Const}(2 : \mathsf{bool}) \parallel \mathsf{Const}(3 : \mathsf{real})) \circ \mathsf{Add}) \parallel \mathsf{Id}) \circ [(x : \mathsf{real}), s \rightsquigarrow s, s + x \cdot dt]$
$(\mathsf{Const}(2 : \mathsf{bool}) \parallel \mathsf{Id}) \circ [(x : \mathsf{real}), s \rightsquigarrow s, s + x \cdot dt]$

In the first case, we try to add a Boolean to a real. The second example contains the first example as a sub-diagram, and it has the same type incompatibility. In

the third example the output of $\mathsf{Const}(2 : \mathsf{bool})$ of type bool is used as the input for the first component of $[(x : \mathsf{real}), s \rightsquigarrow s, s + x \cdot dt]$ which expects a real.

To be able to translate these diagrams, we use type variables instead of the concrete types bool, real, ... Because we work with expressions containing arithmetic and Boolean operations, we need to use type variables of appropriate classes. For example, to translate the leftmost diagram from Fig. 1, we cannot just use an arbitrary type $'a$ because $'a$ must be of class $\mathsf{numeral}$ for the constants 2 and 3, and of class plus. In fact only class $\mathsf{numeral}$ is required here because plus is a subclass of $\mathsf{numeral}$. The generic translation of this diagram is:

$$\mathsf{ConstA}(x : 'a : \mathsf{numeral}) = \mathsf{Const}(2 : 'a), \quad \mathsf{ConstB}(y : 'a : \mathsf{numeral}) = \mathsf{Const}(3 : 'a)$$
$$A(x, y) = (\mathsf{ConstA}(x) \parallel \mathsf{ConstB}(y)) \circ [a, b \rightsquigarrow a + b]$$

In this translation, we only need to specify the types for the constants as discussed in Section 4. However, when we use the type variable $'a$ for numeric constants $1, 2, \ldots$, then we must specify it using the class $\mathsf{numeral}$. If the expression involving the elements of type $'a$ contains some other operators, then we must include also the classes defining these operators. For example we need to have: $\mathsf{ConstA}(x : 'a : \{\mathsf{numeral}, \mathsf{mult}\}) = \mathsf{Const}((2 : 'a) \cdot 3)$. To simplify this we introduce a new class $\mathsf{simulink}$ that contains all mathematical and Boolean operators as well as all real functions that can occur in Simulink diagrams.

$$\begin{aligned}&\mathsf{class\ simulink} = \mathsf{zero} + \mathsf{numeral} + \mathsf{minus} + \mathsf{uminus} + \mathsf{power} + \mathsf{ord} +\\ &\quad \mathsf{fixes\ s\_exp,\ s\_sin} : 'a \to 'a \ \mid\ \mathsf{fixes\ s\_and} : 'a \to 'a \to 'a\\ &\quad \ldots\\ &\quad \mathsf{assume\ numeral\_nzero[simp]} : (\forall a.\mathsf{numeral}(a) \neq 0)\end{aligned}$$

Class zero contains the symbol 0, class $\mathsf{numeral}$ contains the numbers $1, 2, \ldots$, classes minus and uminus contains the binary and unary minus operators, class power contains the power and multiplication operators, and class ord contains the order operators. Because the real functions exp, sin, ... and the Boolean functions are defined just for reals and Boolean types respectively, and they do not have generic type classes, we introduce the generic versions of these functions and operators in the class $\mathsf{simulink}$ ($\mathsf{s\_exp}, \mathsf{s\_sin}, \ldots, \mathsf{s\_and}, \ldots$). Additionally we assume that constant 0 is different from all numeric constants $1, 2, \ldots$.

Using this new class the translation of the rightmost diagram from Fig. 1 is given by

$$\begin{aligned}&\mathsf{ConstA}(x : 'a : \mathsf{simulink}) = \mathsf{Const}(\mathsf{s\_bool}(2 : 'a))\\ &\mathsf{ConstB}(y : 'a : \mathsf{simulink}) = \mathsf{Const}(3 : 'a)\\ &\mathsf{Integral}(dt : 'a : \mathsf{simulink}) = [s, x \rightsquigarrow s, s + x \cdot dt]\\ &\mathsf{Add} = [(x : 'a : \mathsf{simulink}), y \rightsquigarrow x + y]\\ &A(x, y, dt) = (((\mathsf{ConstA}(x) \parallel \mathsf{ConstB}(y)) \circ \mathsf{Add}) \parallel \mathsf{Id}) \circ \mathsf{Integral}(dt)\end{aligned}$$

The inferred type of A is $A(x : 'a : \mathsf{simulink}, y : 'a, dt : 'a) : 'a \xrightarrow{\circ} 'a \times 'a$

In this generic translation there are some details to consider when translating a constant block of type Boolean like the ones from Fig. 1 ($2 : \mathsf{bool}$). In order to use $A(x, y, dt)$ in the end, we still need to instantiate the type variable $'a$. In this case, it would be appropriate to instantiate $'a$ with type real. If we simply use $\mathsf{Const}(2 : 'a)$ in definition of ConstA, then when instantiating $'a$, we will obtain

the constant 2 and we will add it to 3 resulting in 5, and this is not the result obtained when simulating the diagram in Simulink. To preserve the Simulink semantics in the generic case, we translate Boolean constants using a function s_bool which for a parameter x returns 1 if x is different from 0 and 0 otherwise:

$$\text{definition s\_bool}(x) := \text{if } x \neq 0 \text{ then } 1 \text{ else } 0$$

The typing of $x : 'a$ and of $\text{s\_bool}(x) : 'b$ defines again a more precise semantics for $\text{s\_bool}(x)$. For example if both $'a$ and $'b$ are bool, then $\text{s\_bool}(x) = x$. Similarly, we define instantiations for bool and real for all the generic functions defined in the simulink class. These instantiations are detailed in [14].

We implemented this strategy in our Simulink to Isabelle model translator under the −generic option. When this option is missing, then all blocks are defined using their specific types. If this option is given, then only type variables of class simulink are used.

Additionally, we implemented the option −type *isabelle_type* with an Isabelle type parameter, which adds a new definition where it instantiates all type variables to the type parameter.

For example, if we apply the translation using the options −const, −generic, and −type real to the rightmost diagram from Fig. 1, we obtain:

$$\begin{aligned}
&\text{ConstA}(x : 'a : \text{simulink}) := \text{Const}(\text{s_bool}(2 : 'a))\\
&\text{ConstB}(y : 'a : \text{simulink}) := \text{Const}(3 : 'a)\\
&\text{Integral}(dt : 'a : \text{simulink}) := [s, x \rightsquigarrow s, s + x \cdot dt]\\
&\text{Add} := [(x : 'a : \text{simulink}), y \rightsquigarrow x + y]\\
&\text{A}(x, y, dt) := (((\text{ConstA}(x) \parallel \text{ConstB}(y)) \circ \text{Add}) \parallel \text{Id}) \circ \text{Integral}(dt)\\
&\text{A_type}(dt) := \text{A}(0 : \text{real}, 0 : \text{real}, dt : \text{real})
\end{aligned}$$

and also the simplified versions A and A_type:

$$\text{A}(x, y, dt) = [s \rightsquigarrow s, s + (1 + 3) \cdot dt] \quad \text{and} \quad \text{A\_type}(dt) = [s \rightsquigarrow s, s + 4 \cdot dt]$$

In the generic version $\text{s\_bool}(2)$ is automatically simplified to 1 using the definition of s_bool and the assumption numeral_nzero, and in A_type the expression 1+3 is further simplified to 4. In A_type we can eliminate the variables providing types for constants because these types are now instantiated to real.

8 Implementation and Validation

The mechanism presented above for translating Simulink diagrams is implemented in the Refinement Calculus of Reactive Systems framework, available from http://rcrs.cs.aalto.fi. In this framework, Simulink diagrams are translated into Isabelle theories, where diagrams are modeled using predicate transformers. The framework allows the user to perform various analyses on the formal model such as simplification, compatibility checking, safety property verification and simulation.

In order to handle a large set of diagrams, we introduced three translation options: −const, −generic, and −type *isabelle_type*, where each solves different possible corner cases. These options allow some control over the translation

process. More details about these options are available in the extended version of this work [14].

We have extensively tested all combinations of interactions of numeric and Boolean blocks, and we carefully implemented the observed behavior. We have also tested our technique on several examples, including an industrial case study: the Fuel Control System (FCS) benchmark from Toyota [9]. All examples presented in this paper are excerpts from the FCS model. The latter contains 1 constant-related problem as described in §4, 5 implicit conversions, and 5 explicit conversions from which 1 has the inherited output type. Our approach allowed to detect and correct the implicit bool to real conversion present in the FCS model.

Simulink's type system is not formalized, thus it is difficult to make formal claims about its relation to our work. Our experience shows that in most cases our translation results in types that are more general than those in the original diagram.[4] Therefore, instantiating the remaining type variables can be done such that the types of the translation match the types inferred by Simulink.

9 Conclusions and Future Work

We presented a type inference technique for Simulink diagrams which relies on Isabelle's type inference. The main advantage of our technique is that it treats the basic blocks of the diagram *compositionally*, i.e., locally and without knowledge of their context.

Our work is not necessarily restricted to Simulink and could also be used to translate from other weakly typed languages and/or other hierarchical block diagram notations. It could in principle be also applicable to similar in style dataflow languages, with synchronous or asynchronous semantics, which are standard in modeling and reasoning about distributed systems (e.g., see [22]). Our work could also help in implementing translations *into* other systems than Isabelle (e.g., PVS, Z3, Coq), although several challenges need to be overcome as mentioned in §2. The investigation of all these possibilities is part of future work.

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. 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.
3. L. Damas and R. Milner. Principal Type-schemes for Functional Programs. In *POPL '82*, pages 207–212. ACM, 1982.
4. L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340. Springer, 2008.

[4] The only exception is when Boolean values are used in numeric expressions, as discussed in §7, in which case *true* and *false* are modeled as the numbers 1 and 0.

5. E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457, 1975.
6. I. Dragomir, V. Preoteasa, and S. Tripakis. Compositional Semantics and Analysis of Hierarchical Block Diagrams. In *SPIN*, pages 38–56. Springer, 2016.
7. B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI International, 2006.
8. F. Haftmann and M. Wenzel. Constructive Type Classes in Isabelle. In *TYPES*, pages 160–174. Springer, 2007.
9. X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts. Powertrain control verification benchmark. In *HSCC*, pages 253–262. ACM, 2014.
10. B. Meenakshi, A. Bhatnagar, and S. Roy. Tool for Translating Simulink Models into Input Language of a Model Checker. In *ICFEM*. 2006.
11. S. Minopoli and G. Frehse. SL2SX Translator: From Simulink to SpaceEx Models. In *HSCC*, pages 93–98. ACM, 2016.
12. T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.
13. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *CADE*, pages 748–752. Springer, 1992.
14. V. Preoteasa, I. Dragomir, and S. Tripakis. Type Inference of Simulink Hierarchical Block Diagrams in Isabelle. *CoRR*, abs/1612.05494, 2016.
15. V. Preoteasa and S. Tripakis. Refinement Calculus of Reactive Systems. In *EMSOFT*, pages 2:1–2:10. ACM, 2014.
16. V. Preoteasa and S. Tripakis. Towards Compositional Feedback in Non-Deterministic and Non-Input-Receptive Systems. In *LICS*. ACM, 2016.
17. R. Reicherdt and S. Glesner. Formal Verification of Discrete-Time MATLAB/Simulink Models Using Boogie. In *SEFM*, pages 190–204. Springer, 2014.
18. P. Roy and N. Shankar. SimCheck: a contract type system for Simulink. *Innovations in Systems and Software Engineering*, 7(2):73–83, 2011.
19. V. Sfyrla, G. Tsiligiannis, I. Safaka, M. Bozga, and J. Sifakis. Compositional translation of Simulink models into synchronous BIP. In *SIES*, pages 217–220. IEEE, 2010.
20. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.6*, Dec. 2016.
21. 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, 2011.
22. S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincentelli, P. Caspi, and M. D. Natale. Implementing Synchronous Models on Loosely Time-Triggered Architectures. *IEEE Transactions on Computers*, 57(10):1300–1314, Oct. 2008.
23. S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating Discrete-time Simulink to Lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4):779–818, 2005.
24. 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.
25. 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.
26. 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, 2013.