Innovations Syst Softw Eng (2011) 7:73–83
DOI 10.1007/s11334-011-0145-4

SI: NFM 2010

SimCheck: a contract type system for Simulink

Pritam Roy · Natarajan Shankar

Received: 13 September 2010 / Accepted: 26 February 2011 / Published online: 24 March 2011

Abstract Matlab Simulink™ is a member of a class of visual languages that are used for modeling and simulating physical and cyber-physical system. A Simulink model consists of blocks with input and output ports connected using links that carry signals. We provide a contract-based type system of Simulink with annotations and dimensions/units associated with ports and links. These contract types can capture invariants on signals as well as relations between signals. We define a contract-based verifier that checks the well formedness of Simulink blocks with respect to these contracts. This verifier generates proof obligations that are solved by SRI's *Yices* solver for satisfiability modulo theories (SMT). This translation can be used to detect basic type errors and violation of contracts, demonstrate counterexamples, generate test cases, or prove the absence of contract-based type errors. Our work is an initial step toward the symbolic analysis of Matlab Simulink models.

Keywords Matlab · Simulink · Satisfiability modulo theories (SMT) · Dependent types · Units · Dimension checking · Contract-based type-checking

This research was supported by NSF Grant CSR-EHCS(CPS)-0834810 and NASA Cooperative Agreement NNX08AY53A.

P. Roy (✉)
University of California, Los Angeles, USA
e-mail: pritam.roy@gmail.com

N. Shankar
SRI International, Menlo Park, USA
e-mail: shankar@csl.sri.com

1 Introduction

The Matlab Simulink™ framework models physical and computational systems using hierarchical block diagrams. Simulink provides the framework for building and simulating systems from a set of basic building blocks and libraries. Blocks have input and output ports and the ports of different blocks are connected via links. Simulink has a basic type system for annotating ports and links, along with a limited capability for checking type correctness. There is, however, no systematic design-by-contract capability for Simulink. We have developed a tool called *SimCheck* that allows contracts [3,18] to be specified by means of type annotations associated with the ports and links. *SimCheck* can be used to carry out type inference for basic types, annotate links with expressive types. These expressive types are basically contracts on the values, units, dimensions, rows/columns of the block signals. These expressive types based on contracts are used to verify correctness of Simulink designs relative to such expressive types. In addition to that, the verifier can generate test cases and capture interfaces and behaviors. The SimCheck contract-type system can be used to specify and verify high-level requirements including safety objectives.

Unlike prior work [1,2,4,16,19,21] in the verification and testing of Simulink/ Stateflow designs, we pursue an approach based on static type-checking in a rich-type system similar to that of PVS [18]. The SimCheck tool is implemented within Matlab itself and is integrated with Matlab Simulink. Designers can annotate Simulink blocks with contracts written in a simple annotation language. SimCheck tool can typecheck the model with respect to those contract types and provide feedback to the designer whether the design behaves in the intended manner. If it does not, then the tool generates the inputs that are responsible for the type error or contract violation. Designers can simulate the

generated test inputs on the design via dynamic monitoring of type constraints during simulation. The designer can repeatedly fix the design and type-check the modified design in the design-cycle until the SimCheck type-checker passes the design. Thus the integration of the SimCheck tool with Simulink diagrams provides a powerful, interactive tool to the designer.

SimCheck also allows signals types to be annotated with values, dimensions and units, and checks models for well formedness with respect to these annotations. There is a long tradition of work on adding units and dimensions to type systems [10,12–14]. Many system errors occur because of incompatibilities between units. For example, in 1985, a strategic defense initiative (SDI) experiment went awry because the space shuttle pointed its mirror at a point 10,023 nautical miles, instead of feet, above the earth. In September 1999, NASA's Mars Climate Orbiter was placed into a 57 km orbit around Mars instead of a 140–150 km orbit because some calculations used pound force instead of Newton. Though Simulink deals primarily with physical quantities, we know of no prior work on developing a type system with dimensions and units for it. The target applications for SimCheck are in various fields including hardware/protocol verification, Integrated Vehicle Health Management (IVHM) and in cyber-physical systems (CPS). In these cases, the type system is used to capture the safety goals of the system to generate monitors for the primary software to generate alerts.

Each Simulink model consists of a network of blocks. Each block has some common parameters: the block label, the function, default initial values for state variables, and input and output ports. Blocks can be hierarchical so that they are composed from sub-blocks connected by links. Simulink can be used to model both continuous and discrete transition systems. Each output of a Simulink block is a function of its inputs. In a continuous-time model, the output of a block varies continuously with the input, whereas in a discrete-time block, the signals are updated synchronously in each time step. A block can also contain state variables so that the output can be a function, in both the discrete and continuous-time models, of the input and current state.

We focus on the mathematical, rather than computational, interpretation of the Simulink model and present a contract system. Here basic signals are either integers or reals. The output of a model captures the reaction of the state and outputs of a model in response to the input. We assume that there is some basic sampling time at which the model's signals are sampled. This allows us to view each signal as having a value at time t, where t is a non-negative integer corresponding to the tth sampling instant. We assume the models are causal in that once a block has delivered an output in one sampling cycle, it does not react to an input until the next cycle. In the simple version, the contract consists of a pair $\langle I, R\rangle$ where I is an invariant predicate on the inputs that is similar to a Floyd–Hoare precondition, and R is an invariant relation between the state/outputs of a model at time t and the inputs and other signals at times $t, t-1, \ldots, t-k$, for some bound k, and is similar to a post-condition.

To claim that a Simulink design (or a block) satisfies its contracts, we have to verify the claim statically at compile time. Simcheck extracts verification conditions (VC) from the Simulink designs and user provided contracts and send those VCs to a theorem prover. Here we have used Yices SMT solver to get an answer. If we get answer Yes, we know a function satisfies its contract. But if we get answer No, the SMT-solver generates a counter-example to show a failing test-case. While we make heavy use of Yices to solve the constraints involved in contract checking and test case generation, it is quite easy to plug in a different back-end solver.

We develop four analyzers for Simulink models:

1. An expressive contract-based type system that can capture range information on signals as well as relationships between signals. Type correctness can be proved using the Yices solver [5].
2. A test case generator that produces inputs that illustrate contract violations.
3. A unit analyzer that looks at the actual unit dimensions (length, mass, time, etc.) associated with a numeric type.
4. A verifier for temporal properties on state-based Simulink designs using bounded model checking (BMC) and k-induction.

The organization of the paper is as follows. Section 2 provides basic definitions and backgrounds. Section 3 provides a simple language for user to specify the contracts in the Simulink designs. In Sect. 4, we illustrate how the verification conditions are generated for a given Simulink design. In Sect. 5, we illustrate how the user provided constraints and contracts are translated into VCs. In Sect. 6, we illustrate how the user provided safety properties are verified [via BMC and k-induction] for Simulink designs till a user-given depth k. In Sect. 7, we provide details about the Yices SMT solver and how the VCs are translated into Yices input language. We also provide details how the Yices model is generated when VCs fail and user gets feedback.

2 Background

In the following sections, we use first-order logic as a language to describe contracts. Let V be a finite set of variables in the system. A property ϕ over V is a formula ϕ such that any free variable of ϕ is in V. We assume that all variables are typed. A variable x is of type T is shown as $x :: T$. In other words, every variable is associated with a specific domain. With the abuse of notation, we denote $x \in T$ to denote that

the domain of the variable x is the domain of type T. An assignment over a set of variables V is a (total) function mapping every variable in V to a certain value in the domain of that variable. A formula ϕ is satisfiable iff there exists an assignment a over the free variables of ϕ such that a satisfies ϕ, denoted $a \models \phi$. A formula ϕ is valid iff it is satisfied by every assignment. For a variable $x \in V, x(i)$ corresponds to the variable x at time step i. For a variable $x \in V, j \in \{1, 2, \ldots, 7\}, x.dim(j)$ corresponds to the jth dimension of variable x. For a variable $x \in V, x.rows$ and $x.cols$ denotes the number of rows and columns of matrix variable x. For $i \in \{1, \ldots, x.rows\}, j \in \{1, \ldots, x.cols\}, x[i][j]$ denotes the value of ith row and jth column.

2.1 Contracts

Constraints similar to *refinement types* or *predicate subtypes* [18] can be used to capture constraints on the inputs to a function, such as, the divisor for a division operation must be non-zero. They can also be used to capture properties on the outputs, so that, for example, the output of the absolute value function is non-negative. For example, if the value of a variable x of type T, is less than a constant k of type T, then the type of x can be inferred as $x : \{y \in T \mid y > k\}$.

Dependent types can be used to relate the type of an argument or a result to the value of an input. For example, the input signals *ref* and *houseTemp* of a thermostat are dependent in such a way that the value of *houseTemp* should be always higher or equal to 5° below of *ref*. For example, in PVS, dependent typing can be used to ensure that the reverse operation preserves the length of the list.

Contracts The type of a function contains the partial specification of the function. For example, $Sum :: Int \times Int \rightarrow Int$ says that Sum is a function that takes two integers and returns an integer. A *contract* of a function gives more detailed specification. For example, $Sum :: \{x \in Int \mid x > 0\} \times \{y \in Int \mid y > 0\} \rightarrow \{z \in Int \mid z = x + y\}$ says that Sum takes two positive integers and return their sum. The contracts are essentially relations between inputs and outputs.

3 Annotation language for user provided contracts

In the *SimCheck* tool, the user can provide the type information, contracts over types, and properties for type-checking as well as verification via annotations. These annotations can be written as block descriptions or Simulink annotation blocks. The simple grammar of the annotation language has the syntax given in Fig. 1 in which, the terminal tokens are shown in bold. The start symbol of the grammar *blockannotation* consists of two parts : definitions and properties.

```
blockannotation = def | constraints
def = scopedef | typedef | unitdef
scopedef = ( input | output | local) <varname>
typedef = type <varname> :: <vartype>
unitdef = unit <varname> :: <varunitvalue>
constraints= ( iinv| oinv| bmcprop| kind) <expressions>
```

Fig. 1 Syntax of user input

Fig. 2 Expressive types

```
input :: x
input :: s
output :: o
type x :: double
type s :: double
type o :: double
unit x :: inch
unit s :: cm
unit o :: m
iinv  ( /= s 0 )
oinv (>  o 2  )
```

The definitions can provide the details about the scope, datatype or units of the given signal. The tokens $< varname >$, $< vartype >$, $< varunitvalue >$ denote the scope, type, units of the signals, respectively. The contracts are in input–output invariant (tokens *iinv* and *oinv*, respectively) style. We use the tokens *bmcprop* and *kind* for bounded-model-checking and k-induction purposes. The token $< expressions >$ denotes a prefix notation of the expression over the signal variables (syntactically the same as Yices expressions). Figure 2 shows an example of user-given block description.

4 Verification conditions from Simulink design

Simulink diagrams are textually represented as *mdl* files. In SimCheck, we have used built-in Simulink functions to extract the details of the Simulink system. A Simulink design is recursively divided into sub-designs after parsing. The sub-designs are of two kinds—(1) Blocks and (2) connectors. Each block is instantiated with block specific information and user-provided constraints on values, units, and dimensions. Given a system model S, SimCheck internally saves the design into a recursive data structure G. The data structure is similar to graph, but contains various fields related to different fields of the block. For a block B, $B.Kind$ defines the type of the block. Currently we support the following basic blocks: (1) Constant, (2) Mathematical, (3) Comparison, (4) Logical, (5) Subsystem, (6) Switch, (6) Unit Delay/Memory, (7) Inport and Outport, (8) Integration (9) Derivative. $B.v$ denotes the value of block B (for Constant blocks), $B.x$ denotes the state of B (if B has a state). $B.x_0$ denotes the initial state value, $B.i_1$, $B.i_2$ denotes the 1st, 2nd input of block B, respectively. Similarly, $B.o_1$ and $B.o_2$ denotes the 1st and 2nd outputs of block B.

The connectors specify the how those block input and outputs are connected. Each connector has a source block and

port number and a destination block and port number. For example, $C(B_1.o_1, B_2.i_2)$ denotes that the connector connects 1-st output of Block B_1 to 2nd input of Block B_2. The third argument (if present) specifies the name of the connector.

4.1 Contracts from Simulink basic blocks

Matlab blocks generally have polymorphic types. In Simulink, parameters, ports and signals have types. The *basic* types range over $bool, int8, int16, int32, uint8, uint16, uint32$, and $single$ and $double$ floating point numbers. Complex numbers are represented as a pair of numbers (integer or floating point). We ignore non-numeric enumerated types. Simulink also supports objects and classes which are ignored here. The tool creates a fresh new symbolic term T every time the algorithm encounters an unspecified type. The symbolic type variable can be any basic Simulink type. The type inference engine will gather all the constraints from the model and the user-constraints. The data-type for a port can be specified explicitly or it can be inherited from that of a parameter or another port. The type of an output port can be inherited from a constant, an input port, or even from the target port of its outgoing link. For example, the type of the output of a *Product* block is the same as the type of the first input signal to the block. The table at Fig. 3 shows the contracts of basic blocks. These details are obtained from the semantics of Matlab and Simulink blocks. For constant type of block B, $B.v :: T$ implies $B.o_1 :: T$ and $B.o_1 :: T$ implies $B :: Unit \rightarrow T$ For sum type of block B, $B.i_1 :: T$ and $B.i_2 :: T$ implies $B.o_1 :: T$ and $B :: (T \times T) \rightarrow T$. For memory block. $B, B.x :: T$ implies $B.i_1 :: T$ and $B.o_1 :: T$ and $B :: T \rightarrow T$. For subsystem block B defined as $B.i_1 = B1.i_1$, $B.i_2 = B_1.i_2$, $B.o_1 = B_2.o_1$ with $B_1 :: (T_1 \times T_2) \rightarrow T_3$, and $B_2 :: (T_3 \rightarrow T_4)$, and a connector $C(B_1.o_1, B_2.i_1)$ with type $(T_3 \rightarrow T_3)$, we know that the type of B is $(T_1 \times T_2) \rightarrow T_4$.

BlockType	Properties	Block Contracts
Constant	$B.v :: T$	$B :: Unit \rightarrow T$
Math	$B.i_1, B.i_2 :: T$	$B :: (T \times T) \rightarrow T$
Comparison	$B.i_1, B.i_2 :: T$	$B :: (T \times T) \rightarrow bool$
Logical	$B.i_1, B.i_2 :: bool$	$B :: (bool \times bool) \rightarrow bool$
Memory	$B.i_1 :: T$	$B :: (T \rightarrow T)$
Subsystem	$B.i_1 = B_1.i_1$ $B.i_2 = B_1.i_2$ $B.o_1 = B_2.o_1$ $B_1 :: (T_1 \times T_2) \rightarrow T_3$ $B_2 :: (T_3 \rightarrow T_4)$	$B :: (T_1 \times T_2) \rightarrow T_4$

Fig. 3 Contracts from Simulink basic block

4.2 Refinement of Simulink basic types

In this paper, we view the Simulink data values as as refinement types over integers or reals. The Simulink variable x of type $bool$ can be represented as $x :: \{v \in Int \mid 0 \leq v \leq 1\}$. The Simulink variable x of type $int8$ can be represented as $x :: \{v \in Int \mid -128 \leq v \leq 127\}$. The Simulink variable x of type $int16$ can be represented as $x :: \{v \in Int \mid -2^{15} \leq v \leq 2^{15} - 1\}$. The Simulink variable x of type $int32$ can be represented as $x :: \{v \in Int \mid -2^{31} \leq v \leq 2^{31} - 1\}$. The Simulink variable x of type $uint8$ can be represented as $x :: \{v \in Int \mid 0 \leq v \leq 255\}$. The Simulink variable x of type $uint16$ can be represented as $x :: \{v \in Int \mid 0 \leq v \leq 2^{16} - 1\}$. The Simulink variable x of type $uint32$ can be represented as $x :: \{v \in Int \mid 0 \leq v \leq 2^{32} - 1\}$. In a similar manner, fixed point floating (i.e. *single*, *double*) types in Simulink can be represented as refinement of real types. Fixed point real numbers are supported in an extension, but we do not consider them in the paper. We use unbounded real numbers instead.

5 Verification conditions for user given contracts

The expressive type systems of the SimCheck tool are similar to the PVS type system and our tool supports predicate subtypes and dependent subtypes. We also have units and dimension types.

Types of 1-D Arrays and 2-D Matrices Simulink constants or variables can have either a scalar value, a vector or a matrix. In the translation process we assume the matrix of type t as a function of type $(int \times int) \rightarrow t$.

$$\frac{a : (int \times int) \rightarrow T,\ m : int, n : int}{a[m][n] : T}$$

The translation process can catch the following problems via type checking rows and columns- (1) row-column mismatches between blocks, (2) out-of-bounds indices (for vectors/arrays as well as matrices).

Example 1 Figure 4 shows the expressive power of the type systems via thermostat example. The output signal *on* can only take two integer values 0 and 1 (*predicate subtype*). The input signals *ref* and *houseTemp* are dependent in such a way that the value of *houseTemp* should be always higher or equal to 5° below of *ref* (*dependent types*).

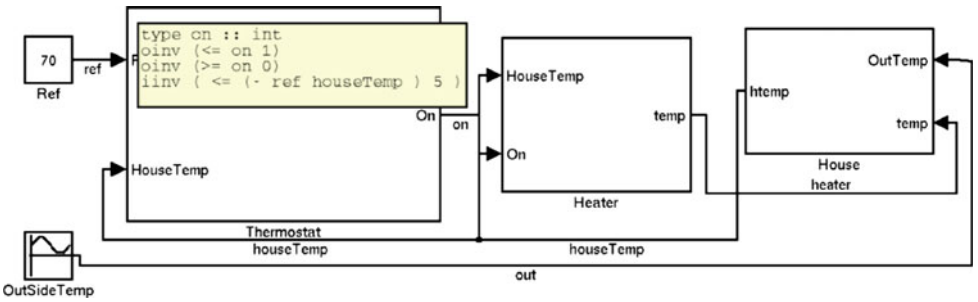


Fig. 4 Thermostat example with dependent and refinement-types

5.1 Constraints due to units and dimensions

Most types in Simulink are numeric. Dimensions add interpretation to such types. A signal may have a numeric type without indicating whether the numeric quantity represents force or volume. Dimension checking can also highlight errors. For instance, the time derivative of distance should yield velocity, and the time derivative of velocity should yield acceleration. Multiplying a mass quantity with an acceleration quantity should yield a force. Each of these, distance, velocity, acceleration, mass, and force can also be represented in different units. We use Novak's classification [17] of units into a 7-tuple consisting of *distance*, *mass*, *time*, *temperature*, *current*, *substance*, and *luminosity*. The dimension of a quantity is represented as a 7-tuple of integers. For example, $\langle 1, 0, 0, 0, 0, 0, 0\rangle$ represents $distance$, $\langle 0, 0, 1, 0, 0, 0, 0\rangle$ represents $time$. $1, 0, -1, 0, 0, 0, 0$ represents velocity since it is of the form $\frac{distance}{time}$. A dimension of the form $\langle 0, 0, 0, 0, 0, 0, 0\rangle$ represents a dimensionless scalar quantity. We infer dimensions for Simulink signals from the annotations for the input signals and the outputs of constant blocks. When a summation operation is applied to a set of input signals, the dimensions of these signals must match, and the resulting summation has the same dimension. When two or more inputs are multiplied, the dimension of the corresponding output is given by the point-wise summation of the dimensions. Even if the dimensions are matched, the units could also be different. In that case the basic types and the dimensions are considered together.

A variable is represented in the Simulink as a record with several fields—rows, columns, values, scaling factors, and unit dimensions. Each field denotes different aspect of variable. For example $op1$ variable has scaling factor 0.0254, 1 row, 1 column, value x, unit dimensions $\langle 1, 0, 0, 0, 0, 0, 0\rangle$, respectively. Similarly, $op2$ variable has scaling factor 0.01, 1 row, 1 column, value s, unit dimensions $\langle 1, 0, 0, 0, 0, 0, 0\rangle$, respectively. The verification conditions are generated for the Add block in Simulink on those records. Let Add block has two inputs $op1$ and $op2$ and one output $res1$. The first condition is to check whether the units of the operands have same dimensions. The next conditions check whether the rows and columns of the operands are equal. The final condition checks whether the values of operands multiplied with their respective scaling factors are equal to the product of the scaling factor and values of the result. For $i \in \{1, 2, \ldots, 7\}$,

$$op1.dim(i) = op2.dim(i),\ op1.dim(i) = res1.dim(i)$$
$$op1.rows = op2.rows,\quad op1.rows = res1.rows$$
$$op1.cols = op2.cols,\quad op1.cols = res1.cols$$

In addition to the dimension matching, value-based constraints can be generated from the units. SimCheck keeps a scaling-factor map from known units to standard SI/metric units. For example, 1inch = 0.0254m and 1cm = 0.01m. For example, if the adder block input values are $op1.v$ inch. and $op2.v$ cm. and output value is $Sum.v$ m., then the verification condition will be

$$(0.0254 \cdot op1.v + 0.01 \cdot op2.v) = Sum.v.$$

The verification conditions generated for the $Product$ block are similar to the Add block, however, there are some new conditions. Let $Product$ block also has two inputs $op1$ and $op2$ and one output $res2$. The units of the product is pairwise summation of the operands. The number of columns of first operand should be equal to the number of rows of the second operand. The number of rows of product is same as the number of rows of the first operand, number of columns of the product is same as the number of columns of the second operand. For $i \in \{1, 2, \ldots, 7\}$,

$$op1.dim(i) + op2.dim(i) = res2.dim(i)$$
$$op1.cols = op2.rows,\ op1.rows = res2.rows$$
$$op2.rows = op1.cols,\ op2.cols = res2.cols$$

6 Property verification via BMC and k-induction

Most practical designs contain memory elements and state variables; thus the effect of inputs spans multiple clock-cycles. Hence, for a design with state-variables, the errors can only be detected with a sequence of test inputs. For a transition system with variables X, initial states I and transition relation T, SimCheck can generate VCs to verify a safety property till a given bound. For BMC with depth k, we need

to unroll the transition relation, check whether the property satisfies. To find a counter-example, SimCheck unrolls the transition checks the negation of property P.

$$(I\ (x_0) \wedge T\ (x_0, x_1) \wedge \ldots T\ (x_{k-1}, x_k)) \\ \wedge (\neg P\ (x_0) \vee \ldots \neg P\ (x_{k-1}))$$

Usual induction to prove invariant property P. The following conditions has to be satisfied:

- Base case : $I(x_0) \rightarrow P(x_0)$
- Induction step : $P(x_0) \wedge T(x_0, x_1) \rightarrow P(x_1)$

The following conditions has to be satisfied to prove invariant property P by k-induction:

- Base case : $I(x_0) \wedge T(x_0, x_1) \wedge \ldots \wedge T(x_{k-2}, x_{k-1}) \rightarrow P(x) \wedge \ldots \wedge P(x_{k-1})$
- Induction step : $P(x_0) \wedge T(x_0, x_1) \wedge \ldots \wedge P(x_{k-1}) \wedge T(x_{k-1}, x_k)) \rightarrow P(x_k)$

To find a counter-example, SimCheck checks negates these two conditions and tries to find an assignment that fails either base case or induction step.

7 Translation of VCs to the *Yices* language

Our annotations are written in the constraint language of the *Yices* SMT solver [5]. We use Yices as our main tool in checking contracts and finding test cases and counterexamples. The non-linear-constraints are solved using *HySAT* [11] solver. A solver for *Satisfiability Modulo Theories (SMT)* can determine if a given formula has a model relative to background theories that include linear arithmetic, arrays, data types, and bit vectors.

The formula language for Yices is similar to that of PVS and is quite expressive. It includes a higher-order logic with predicate subtypes and dependent types. Interaction with Yices occurs through some basic commands

1. `(define` $\langle identifier \rangle :: \langle type \rangle \langle expression \rangle$ `)` : Defines a constant.
2. `(assert` $\langle formula \rangle$ `)` : Asserts a formula to the context.
3. `(check)` : Checks if the context is satisfiable.

As a example, Yices responds with `unsat` to the input

```
(define x::real)
(assert (< (+ x 1) x))
```

Yices can handle large problems with thousands of variables and constraints involving Booleans, arrays, bit vectors, and uninterpreted function symbols.

7.1 Translation procedure

The type system at Yices is different from the type systems at Matlab. Hence *double* is converted to *real*. The integers *int8*, *int16*, *int32*, *int64* and their unsigned counterparts are converted to *int*. The name of a variable contains the name of the context, name of the block, the type of port, port number and the clock cycle number. One sample name would be

```
Projectile__Product__In1__time1.
```

The context provides the name of the model and the subsystems that encapsulates the block where the variable exists.

The function $dump2yices$ (Algorithm 1) translates every visited block and connector transition into the Yices language. The procedure *printBlockInfo(block,d)* dumps the type information of the ports to the *Yices* file. We exploit the Yices operators directly to translate basic arithmetic and logical operator blocks. The procedure also provides the transition-function of the block from the input ports to the output ports. The algorithm *parseDescription* parses the user-given annotations for the type-checking and verification step. The algorithm follows the annotation grammar described in Fig. 1. The algorithm *printConnectors* translates the connector assignments. Each connector signal obtains the value and type from its source block. Similarly, the signal variables propagate the type and value to its destination block inputs.

Algorithm 1 $dump2yices(G, depth)$

```
1.  (B, B_0, C) = G, descr = ' '
2.  for d ∈ {1, 2, ..., depth + 1}
3.      for b ∈ B
4.          descr = descr + parseDescription(b, d)
5.          printBlockInfo(b, d)
6.      end for
7.      printConnectors(C, d)
8.  end for
```

The user can add various contracts over Simulink blocks. The tool first checks the compatibility of the design with user-provided constraints. Let C_{model} and C_{user} denote the set of VCs from the model and user-provided constraints, respectively. If the user-given constraints are not satisfiable with respect to the Simulink design, then the SimCheck tool declares that the design blocks are not compatible with the constraints. In other words, the Yices solver returns *unsat* result for the query $\cap_{cs \in (C_{model} \cup C_{user})} cs$. There is no environment (i.e. test input) that can satisfy the given user-constraints and the design.

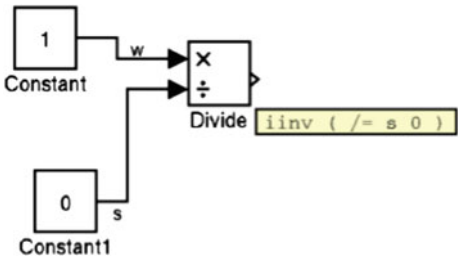


Fig. 5 Compatibility

For example, Fig. 5 illustrates that the divider block only accepts non-zero inputs. If the design connects constant zero to the second input of the divider block the design fails the compatibility checking.

```
(define  Divide ::(-> real real real ))
(define  Divide__In1__time1 :: real)
(define  Divide__In2__time1 :: real)
(define  Divide__Out__time1 :: real )
(assert (=  Divide__Out__time1
   (Divide  Divide__In1__time1 Divide__In2__time1)))
(assert (= DivideIn2 0))
(check)
-> unsat
```

With this kind of expressive typing, we can generate test cases that conform to input constraints. The test criterion can itself be expressed as a type constraint.

7.2 Test case generation

When a type constraint does not hold, we can construct test cases that illustrate the violation. If no such violations are found, this validates type correctness relative to the expressive type annotations. Of course, this assumes that we have a sound and complete decision procedure for the proof obligations arising from such type constraints, but this is not always the case. Many Simulink models are finite-state systems and expressive type checking can be decidable, even if it is not practical. However, for the purpose of expressive type checking, we assume that the signal types are taken from mathematical representations such as integers and real numbers.

A test-case generator produces inputs that conform to the specified type constraints, or illustrate contract violations. Let S and C denote the set of signals and user-constraints in a design, respectively. For every signal $s \in S$ and for every constraint $cs \in C$ on signal s, the tool asks for the satisfiability of the $(\neg cs) \cap (C \backslash cs)$. The Yices solver returns either a negative answer (*unsat*) or a positive answer with a satisfying variable assignment. In the former case, we learn that there is no input that can satisfy $(\neg cs) \cap (C \backslash cs)$. In the latter case, the variable assignment can provide a test-case that fails exactly one of the user-given constraints and satisfies the other constraints.

Example 2 Figure 6 shows a design with various user-given input assumptions $C = \{(/ = \ s\ 0), (<= \ x\ 8), (>= \ z\ 5)\}$ of a design. Figure 2 shows the constraints of the *Add* block in Fig. 6. We can verify that each of the three test-cases represents one case where all except one constraint is satisfied.

8 Case studies

SimCheck verification capabilities are illustrated with the following examples.

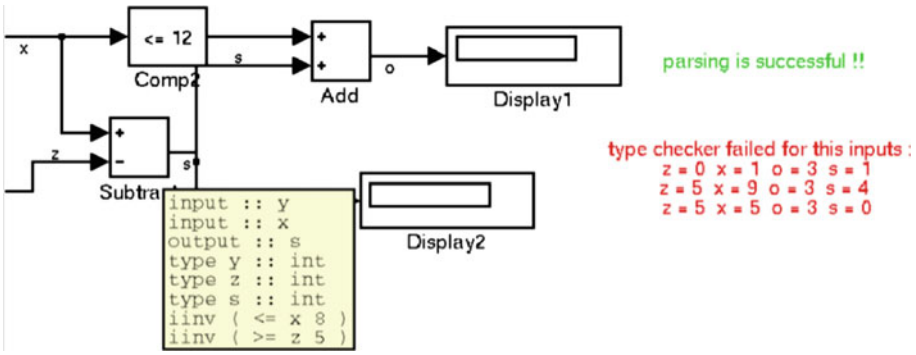
8.1 Modulo-3 counter

Figure 7 shows a design of a modulo-3 counter using 2 memory bits. The transition relation for ith cycle i.e. $transition_i$ is

$$(m_0\,(i+1) = f_1 \wedge m_1\,(i+1) = f_2)$$

where, $f_1 := \neg(m_0(i) \vee m_1(i))$ and $f_2 := (m_0(i) \wedge \neg(m_1(i)))$. According to the specification, both bits cannot become 1 together. Hence in the example the invariant property is $\phi_i := \neg(m_0(i) \wedge m_1(i))$. If the design is correct then the counter will never reach 3 and the given property will hold for any k. For any user-given depth k, the BMC tool asks the following query to the Yices solver:

Fig. 6 Test case generation



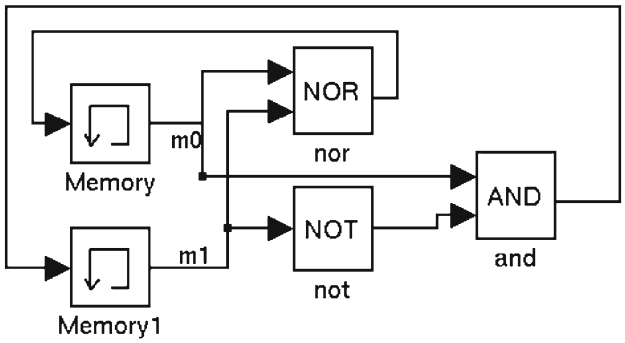


Fig. 7 Bounded model checking

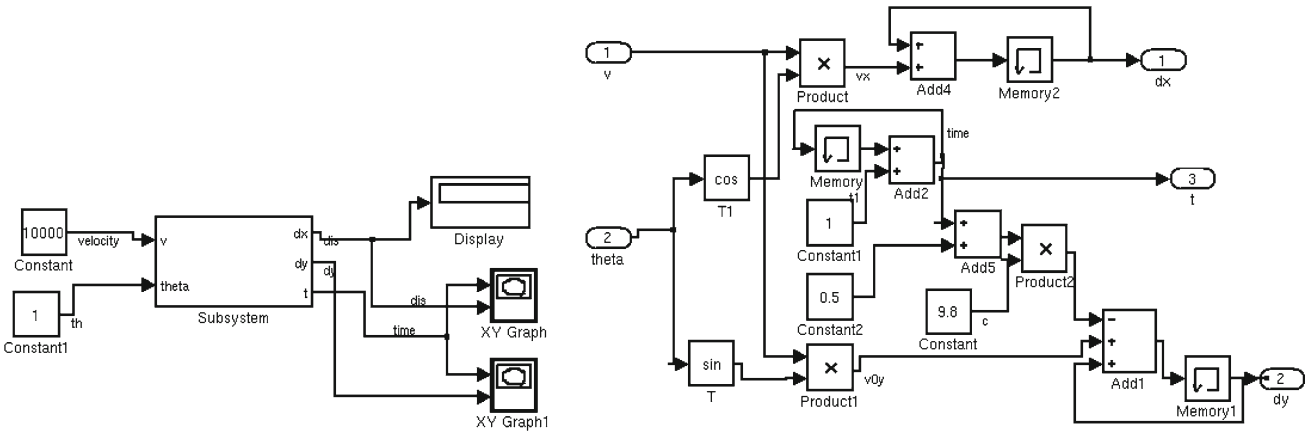
$\wedge_{i\in\{1,2,\ldots,k\}} transition_i \wedge (\vee_{i\in\{1,2,\ldots,k\}} \neg\phi_i)$. For a correct design, the negated invariant property will never get satisfied and the BMC will return the *unsat* result.

8.2 Projectile system

Figure 8a shows the Matlab model and it takes three inputs: the firing angle θ with respect to the horizontal, the initial velocity v, and the height h_0 above the ground. The model computes the horizontal and vertical distance of the projectile at time t from the origin (0, 0) via dx and dy, respectively. The actual computation is carried out within a subsystem shown in Fig. 8b. Figure 9 shows the Hysat input file for the non-linear design of projectile. Figure 10 the results of BMC with the counter-example with respect to the safety invariant ($yh >= 0$). Figure 11 shows the projectile design and the output of the dimension checker tool.

9 Related work

There is a substantial body of work [1,2,7,16,19–21,23] in all of the above areas. The Reactis tool [20] generates and simulates test cases for Simulink/State-flow models including embedded C code. The Mathworks Simulink Design Verifier also performs test generation. The Simulink Design Verifier is also capable of generating test cases and for proving and refuting functional properties of Simulink blocks. The CheckMate tool [7] analyzes properties of hybrid systems represented by Simulink/State-flow models using a finite state abstraction of the dynamics. The Honeywell Integrated Life-cycle Tools and Environment (HiLiTE) [4] from Honeywell Research uses static analysis to constrain the ranges of inputs to generate test cases, detect ambiguities and divide-by-zero errors, and unreachable code. The HiLiTE tool bases its analysis on numeric techniques on a floating-point representation, whereas our approach is based on symbolic constraint solving over the real numbers. The Simulink-to-Lustre translator defined by Tripakis et al. [23], performs type inference on Simulink models and the resulting Lustre output can be analyzed using model checkers. The Gryphon tool suite [16] integrates a number of tools, including model checkers, for analyzing properties of Simulink/State-flow models. There is a long tradition of work on adding units and dimensions to type systems. These are surveyed in Kennedy's dissertation [13] where he presents a polymorphic dimension inference procedure. Kennedy [14] also proves that these programs with polymorphic dimension types are parametric so that the program behavior is independent of the specific dimension, e.g., whether it is weight or length, or its unit, e.g., inches or centimeters. The resulting type system, in which the numeric types are tagged by dimension, has been incorporated into the language F#. The recently designed Fortress programming language for parallel numeric computing also features dimension types [10]. We use an approach to dimensions that is based on a fixed set of seven basic dimensions. Operations are parametric with respect to this set of dimensions. The type is itself orthogonal



(a) Trajectory **(b)** Subsystem

Fig. 8 Projectile

```
DECL
    int [0, 100] t;       -- global time
    float [-100.0, 100.0] dx;  -- x position
    float [-100.0, 100.0] dy;  -- y position
    define g  = 9.8;-- gravitational acceleration
    define v = 10;
    define theta = 0.84;
INIT
    t = 0.0;
    dx = 0;
    dy = 50;
TRANS
    dx' = dx + v*cos(theta);
    dy' = dy + v*sin(theta)-g*(t+0.5);
    t' = t+1;
TARGET
    dy < 0;
```

Fig. 9 HySat input

to the dimension, so that it could be an `int32`, `float`, or `double`, and still have a dimension corresponding to the weight. We use an SMT solver to perform type inference with dimensions, and use scaling to bridge mismatched units.

The work on Model Ontology [15] defines types for signals over a complete concept lattice. Signal data types are arranged in a lattice with the empty type at the bottom and the most general type at the top so that $S \leq T$, when any element of type S can be losslessly converted to one of type T. A similar complete lattice is given for dimensions like speed and acceleration. Type constraints are viewed as monotone operators over a concept lattice, and the Rehof–Mogensen algorithm is used to compute a least fixpoint of a set of constraints.

Chen et al. [6] define a semantics for Simulink diagrams using a Timed Interval Calculus which is formalized in PVS. The timed interval calculus is used to capture relations between timed traces in terms of constraints on intervals.

We solve for dimensions using a constraint solving algorithm, where the constraints are expressed over a domain of n-tuples of integers. Since the constraint solver, Yices in this case, does not return symbolic solutions, we do not capture dimension polymorphism, but we can detect dimension errors. Tripakis et al. [23] provide a translation from discrete-time Simulink to the data flow language Lustre. They also provides a basic typechecking algorithm that is similar to the one we use—we have omitted the details from this paper.

Tripakis et al. [22] present a *relational interface* as a map Ξ from a sequence of valuations of the input/output variables (the history) to a contract on the new values of input/output variables. It is stateless if Ξ ignores the history. They constrain their interfaces to be input contravariant but not output

```
=> ./hysat projectile.hys --nostats --max_depth 5
# This is HySAT 0.8.5, compiled on 25.11.2009.
Reading input file 'projectile.hys'.
Preprocessing input formulae.
SOLVING:
    k = 0
RESULT:
    unsatisfiable
....
SOLVING:
    k = 5
RESULT:
    candidate solution box found
SOLUTION:
    t (int):
        @0: [0, 0]
        @1: [1, 1]
        @2: [2, 2]
        @3: [3, 3]
        @4: [4, 4]
        @5: [5, 5]
    dx (float):
        @0: [0, 0]
        @1: [6.6746282584030813823, 6.6746282584230813839]
        @2: [13.349256516806162765, 13.349256516846162768]
        @3: [20.023884775209243259, 20.02388477526924504]
        @4: [26.698513033612321976, 26.698513033692329088]
        @5: [33.373141292015397141, 33.373141292115413137]
    dy (float):
        @0: [50, 50]
        @1: [52.546431199698588443, 52.546431199718604432]
        @2: [45.292862399397179729, 45.292862399437204601]
        @3: [28.239293599095770304, 28.239293599155807613]
        @4: [1.3857247987943654977, 1.3857247988744045841]
        @5: [-35.267844001507043572, -35.267844001406992049]
```

Fig. 10 Hysat results

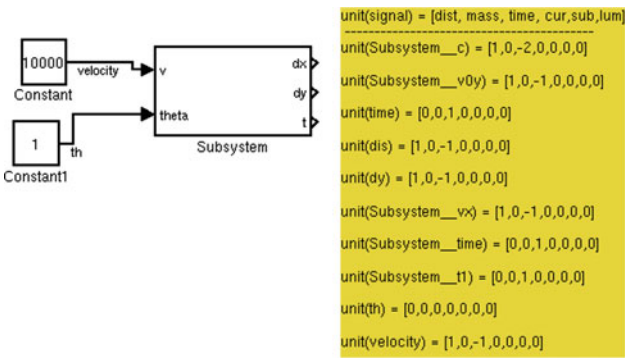


Fig. 11 Unit and dimensions

covariant, and prove a substitutability theorem for their interfaces that one interface can be replaced with a more refined

one in any context including feedback loops. Our contracts are different from relational interfaces in that we separate them into an input invariant precondition and a relational invariant between the input and output signals that can only reference a bounded number of prior values of the signals.

10 Future directions

This work is our first step toward type-based verification of physical and cyber-physical models described in Simulink. Our work can be extended in a number of directions. Many models also contain Simulink blocks to represent discrete control in the form of hierarchical state machines. For static type checking and BMC, we can extract the relationships between inputs and outputs of the State-flow descriptions. We can also associate interface types that capture the temporal input–output behavior of State-flow state machines. Our current translation to Yices maps the Simulink bounded integer types to the unbounded integer types of Yices, and the floating point types of Simulink are mapped to the real numbers in Yices. This makes sense for checking the idealized behavior of the models. We can also map these types to their bounded representations or even directly to bit vectors. The latter translations are more useful when checking the execution behavior of the models or in validating the code generated from the models. The SimCheck type system and Yices translation can also be extended to capture more extensive checking of Simulink models. We would like to extend our checks to cover Simulink S-functions which are primitive blocks defined in the M language. We also plan to cover other properties such as the robustness of the model with respect to input changes, error bounds for floating point computations, and the verification of model/code correspondence through the use of test cases [4]. The type system can also be extended to handle interface types [8,9] that specify behavioral constraints on the input that ensure the absence of type errors within a block. Finally, we plan to translate Simulink models to SAL and HybridSAL representations for the purpose of symbolic analysis.

11 Conclusions

We have outlined an approach to the partial verification of Simulink models through the use of a contract system. This contract annotation language can capture constraints on signals, the dimensions and units of signals, and the relationships between signals. These annotations are translated into verification conditions and finally into the constraint language of Yices. The resulting proof obligations are translated to the Yices/HySat constraint solver which is then used to check compatibility with respect to dimensions, generate counterexamples and test cases, and to prove type correctness. SimCheck tool also performs BMC and k-induction to refute or verify contract and type invariants. SimCheck represents a preliminary step in exploiting the power of modern SAT and SMT solvers to analyze the Simulink models of physical and CPS. Our eventual goal is to use this capability to certify the correctness of such systems.

References

1. Adams MM, Clayton Philip B (2005) ClawZ Cost-effective formal verification for control systems. In: Kung-Kiu L, Richard B (eds) 7th international conference on formal engineering methods, ICFEM, Manchester, UK, 1–4 November, proceedings of lecture notes in computer science, vol 3785. Springer, Manchester, pp 465–479
2. Aditya K, Rajeev A, Franjo I, Ramesh S, Sriram S, Shashidhar KC (2009) Generating and analyzing symbolic traces of Simulink/Stateflow models. In: 21st CAV, Grenoble, France, June 26–July 2, 2009, LNCS vol 5643. Springer, France, pp 430–445
3. Bertrand M (1997) Design by contract: making object-oriented programs that work. In: TOOLS 1997: 25th international conference on technology of object-oriented languages and systems, 24–28 November 1997, Melbourne, Australia, vol 25, p 360
4. Bhatt D, Hickman S, Schloegel K, Oglesby D (2007) An approach and tool for test generation from model-based functional requirements. In: Proceedings of the 1st international workshop on aerospace software engineering, Minneapolis, USA, 21–22. May
5. Bruno D, de Moura L (2006) A fast linear-arithmetic solver for DPLL(T). In: 18th CAV 2006, Seattle, WA, USA, 17-20 August, Lecture notes in computer science, vol 4144. Springer, Berlin, pp 81–94
6. Chen C, Dong JS, Sun J 0001 (2009) A formal framework for modeling and validating simulink diagrams. Formal Asp Comput 21(5):451–483
7. Chutinan A, Krogh B (2003) Computational techniques for hybrid system verification. IEEE Trans Autom Control 48(1):64–75
8. de Alfaro L, Henzinger TA (2001) Interface automata. SIGSOFT Softw Eng Notes 26(5):109–120
9. de Alfaro L, Henzinger TA (2001) Interface automata. In: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering. ESEC/FSE-9, Vienna, Austria. ACM, New York, pp 109–120, ISBN: 1-58113-390-1
10. Eric A, David C, Victor L, Jan-Willem M, Ryu S, Steele GL Jr, Tobin-Hochstadt S (2005) The Fortress language specification, version 0.618. Technical report, Sun Microsystems, Inc, San Antonio
11. Fränzle M, Herde C (2007) Hysat: an efficient proof engine for bounded model checking of hybrid systems. Form Methods Syst Des 30(3):179–198
12. Hayes IJ, Mahony BP (1995) Using units of measurement in formal specifications. Formal Asp Comput 7(3):329–347
13. Kennedy AJ (1996) Programming languages and dimensions. PhD thesis, University of Cambridge, 1996. Published as University of Cambridge Computer Laboratory Technical Report No. 391

14. Kennedy AJ (1997) Relational parametricity and units of measure. In: The 24th ACM POPL '97, January 1997, pp 442–455
15. Man-Kit LJ, Thomas M, Edward LA, Elizabeth L, Charles S, Stavros T, Ben L (2009) Scalable semantic annotation using lattice-based ontologies. In: 12th international conference on model driven engineering languages and systems, October 2009. ACM/IEEE, New York, pp 393–407 (recipient of the MODELS 2009 Distinguished Paper Award)
16. Michael WW, Cofer DD, Miller SP, Krogh BH, Storm W (2007) Integration of formal analysis into a model-based software development process (2007) In: Stefan L, Pedro M (eds) FMICS, Lecture notes in computer science, vol 4916. Springer, Berlin, pp 68–84
17. Novak GS (1995) Conversion of units of measurement. IEEE TSE 21(8):651–661
18. Owre S, Rushby J, Shankar N, von Henke F (1995) Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. IEEE Trans Softw Eng 21(2):107–125
19. Rajeev A, Aditya K, Ramesh S, Shashidhar KC (2008) Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In: Proceedings of the 8th ACM & EMSOFT 2008, Atlanta, USA, 19–24 October. ACM, New York, pp 89–98
20. Reactive Systems. Model based testing and validation with Reactis, Reactive Systems inc., http://www.reactive-systems.com
21. Rushby J, de Moura L, Hamon G (2004) Generating efficient test sets for Matlab/Stateflow designs. Task 33b report for Cooperative Agreement NCC-1-377 with Honeywell Tucson and NASA Langley Research Center, Computer Science Laboratory, SRI International, Menlo Park, CA, May 2004
22. Tripakis S, Lickly B, Henzinger T, Edward LA (2009) On relational interfaces. In: Embedded software (EMSOFT'09), October 2009
23. Tripakis S, Sofronis C, Caspi P, Curic A (2005) Translating discrete-time simulink to Lustre. ACM Trans Embed Comput Syst (TECS) 4(4):779–818