SimCheck: An Expressive Type System for Simulink[*]

Pritam Roy
University of California Santa Cruz
Santa Cruz, CA, USA
pritam.roy@gmail.com

Natarajan Shankar
SRI Computer Science Laboratory
Menlo Park CA 94025 USA
shankar@csl.sri.com

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 extend the type system of Simulink with annotations and dimensions/units associated with ports and links. These types can capture invariants on signals as well as relations between signals. We define a type-checker that checks the well-formedness of Simulink blocks with respect to these type annotations. The type checker generates proof obligations that are solved by SRI's *Yices* solver for satisfiability modulo theories (SMT). This translation can be used to detect type errors, demonstrate counterexamples, generate test cases, or prove the absence of type errors. Our work is an initial step toward the symbolic analysis of MATLAB Simulink models.

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 SimCheck capability that allows contracts [Mey97, ORSvH95] 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, unit types, generate test cases, capture interfaces and behaviors, monitor type conformance, and verify type correctness relative to such expressive types. The SimCheck type system can be used to specify and verify high-level requirements including safety objectives.

Unlike prior work [RdMH04, AC05, KAI+09, AKRS08, BHSO07, WCM+07] in the verification and testing of Simulink/Stateflow designs, we pursue an approach that is closer to the *correct-by-construction* paradigm. Our SimCheck tool is similar to the type-checker of the strongly-typed functional languages (e.g. OCaml, Haskell etc.). The SimCheck tool is written in Matlab language and is integrated with MATLAB Simulink. The designer can annotate the Simulink blocks with a type written in a simple annotation language. SimCheck tool can type-check the model with respect to the type 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 design to fail the type-checker. The designer 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 the Simulink diagrams provides a powerful, interactive tool to the designer.

There is a long tradition of work on adding units and dimensions to type systems [ACL$^+$05, Ken97, Ken96]. 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 1999, the Mars Climate Orbiter entered an incorrect orbit and was destroyed because of confusion between metric and imperial units of measurement. Though Simulink deals primarily with physical quantities, we know of no prior work on developing a type system with dimensions and units for it.

Each Simulink model consists of 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 synchronous 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 update 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 develop four analyzers for Simulink models:

1. An expressive type system that can capture range information on signals as well as relationships between signals. Type correctness can be proved using the Yices solver.

2. A test case generator that produces inputs that conform to the specified type constraints, or that illustrate type 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 properties using bounded model checking and *k*-induction.

While we make heavy use of Yices to solve the constraints involved in type inference, analysis, and test case generation, it is quite easy to plug in a different back-end solver. 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 in order to generate alerts.

2   Types and Simulink

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 enumerated types which contain string values. Fixed point real numbers are supported in an extension, but we do not consider them here. The datatype 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. Simulink has structured types which are vectors or matrices of the basic Simulink types. Simulink also supports objects and classes which are ignored here. We do not address basic typechecking for Simulink since this is already handled by MATLAB. We add a stronger type-checking to the built-in type-checker. In this paper, however, we focus on more expressive type annotations for Simulink covering dimensions and units, predicate subtypes and dependent types, and bounded model checking. 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. *Yices*[DdM06] is an SMT Solver developed at SRI International. Our annotations are written in the constraint language of the Yices SMT solver. We use Yices as our main tool in checking contracts and finding test cases and counterexamples. We provide a basic introduction to the Yices solver in the Appendix.

3   Translation of Designs to the *Yices* Language

The translation scheme is illustrated with the example of the trajectory of a projectile. Figure 1 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 quantity *vy* denotes the vertical velocity of the projectile.
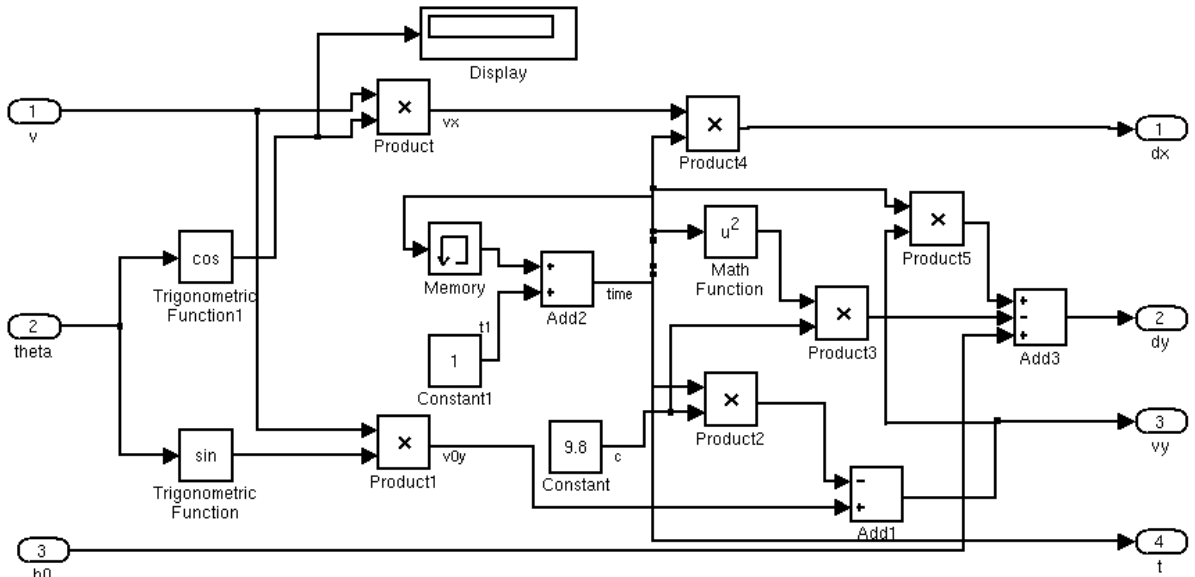
Figure 1: Projectile Subsystem

Parsing MDL Files :   Simulink diagrams are textually represented as *mdl* files. We have used the built-in Simulink functions to create an *abstract syntax tree (AST)*. Algorithm 1 resolves the types of every signal by block and connector information. Matlab blocks generally have polymorphic types. All the blocks in Figure 1 can take different types. However *Yices* is a strongly-typed language; so the algorithm needs to resolve the types of each signal before the translation process. The algorithm traverses the Simulink blocks from the source blocks (e.g. *Constant* blocks) and obtains the type information for the ports /signals gradually. For some signals and ports the concrete type information is not available and the tool creates an equivalence type-map for these variables. For example, type of *v* is equivalent to the type of first input of *Product* block in Figure 1. The algorithm traverses the blocks in the design in a *Breadth-First Search (BFS)* manner. The only interesting cases are when the algorithm dequeues a *subsystem* block or an outport of a subsystem block. In the first case, the algorithm appends all the inports of the subsystem block. The outport of a subsystem block appends the successor blocks of the parent subsystem.

Algorithm 1 MDL2YICES (model)

$visitedBlocks = \emptyset$

1. create the block transitions by parsing the connectors
2. push source blocks of the design into a queue
3. **while** queue is non-empty
4.   block x = dequeue()
5.   parse block x for description, type information and block-transitions
6. $visitedBlocks = visitedBlocks \cup \{x\}$
7.   **if** block is subsystem **then** append inports of x
8.   **else if** block is an outport **then** append the successors of its parent subsystem
9.   **else** append its successor of x
10.   **end if**
11.**end while**
12.dump2yices(model,visitedBlocks,1)

Block Translation: The function *dump2yices* (Algorithm 2) 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. The procedure also provides the transition-function of the block from the input ports to the output ports.

Example 1. *The type information and the block transitions of the* Divide *block are as follows:*

```
(define  Divide :: (-> real real real ))
(define  DivideIn1 :: real )
(define  DivideIn2 :: ( subtype (n :: real ) (/= n 0)))
(define  DivideOut :: real )
(assert (=  DivideOut ( Divide  DivideIn1 DivideIn2 )))
```

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. We provide translation schemes for all other basic blocks in the Appendix. The algorithm *parseDescription* parses the user-given annotations for the type-checking and verification step. We describe the annotation grammar in the next section.

Algorithm 2 dump2yices($model, visitedBlocks, depth$)

---

1. $descr = $ ' '
2.   **for** d $\in \{1, 2, \ldots, depth + 1\}$
3.   **for** block $\in visitedBlocks$
4. $descr = descr + parseDescription(block, d)$
5.   printBlockInfo(block,d)
6.   **end for**
7.   printConnectors(d)
8.   **end for**

---

4 An Expressive Type System

In the *SimCheck* tool, the user can provide the constraints 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 following syntax:

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

where, the terminal tokens are shown in bold. The start symbol of the grammar *blockannotation* consists of two parts : definitions and properties. 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 constraints are in either input-output invariant (tokens *iinv* and *oinv* respectively) style or the assume-guarantee (tokens *check* and *prop* 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(a) shows an example of user-given block description.

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

Example 2. *For example, Figure 2(b) 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. The translation algorithm adds the following lines to the codes of Example 1:*

```
(assert (= DivideIn2 0))
(check)
```

$\rightarrow$ *unsat*

Dependent Types and Refinement Types: The expressive type-systems of the SimCheck tool are similar to the PVS type-system and our tool supports predicate subtypes and dependent subtypes. Predicate subtyping 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. Finally, dependent typing can be used to relate the type of an argument or a result to the value of an input. For example, we can constrain the contents array of a stack data type to have a size equal to the size slot. We can also ensure that the length of the result of appending two lists is the sum of the lengths of these two lists.

Example 3. *Figure 3 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 degrees below of* ref *(dependent 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  )
```

Constant 1

Constant1 0 s

Divide iinv (/= s 0)
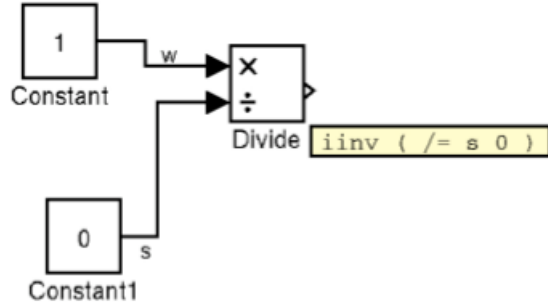
(a) Annotation Block   (b) Compatibilty

Figure 2: Expressive Types


```
type on :: int
oinv (<= on 1)
oinv (>= on 0)
iinv ( <= (- ref houseTemp ) 5 )
```
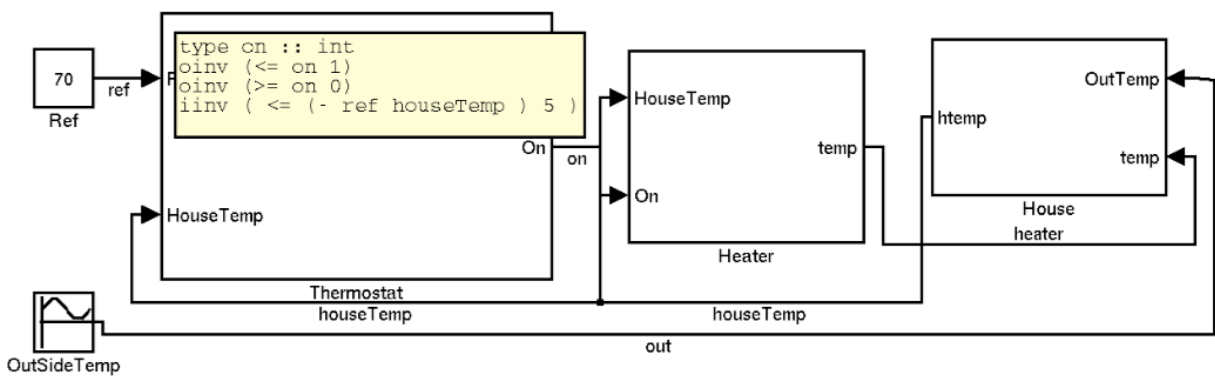
Figure 3: Thermostat Example with Dependent and RefinementTypes

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. 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.

Test Case Generation A test-case generator produces inputs that conform to the specified type constraints, or illustrate type 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 $(\neg cs) \cap (C \setminus cs)$. 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 $(\neg cs) \cap (C \setminus cs)$. 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 4. *Figure 4 shows a design with various user-given input assumptions $C = \{(/ = s\ 0), (<= x\ 8), (>= z\ 5)\}$ of a design. Figure 2(a) shows the constraints of the* Add *block in the Figure 4. We can verify that each of the three test-cases represents one case where all except one constraint is satisfied.*
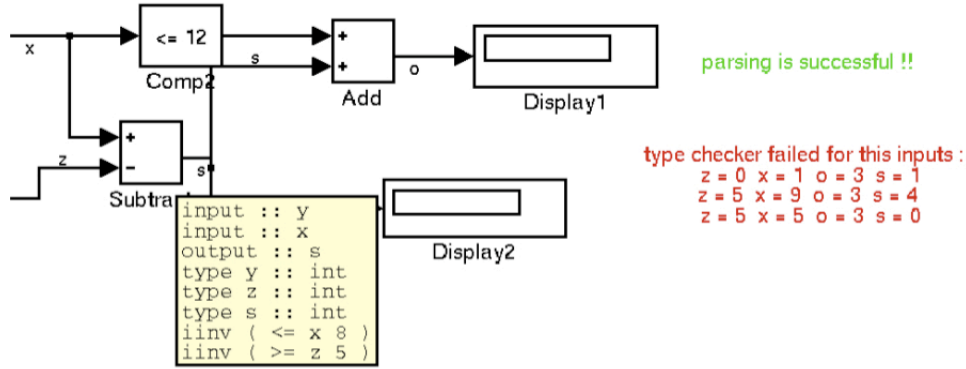
Figure 4: Test Case Generation

5 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. The type checking for the sequential Simulink models can be extended to the verification of the systems by bounded model-checking and k-induction algorithms. Algorithm 3 and Algorithm 4 provide the pseudo-code for the bounded model checking and k-induction respectively for a property ϕ and a user-given depth k.

Algorithm 3 BMC($k, vBlocks, \phi, model$)

1. $transition(1, k+1) = dump2yices(model, vBlocks, k)$
2. $\psi = \vee_{d \in \{1,2,\ldots,k+1\}} \neg \phi_d$
3. check(transition, ψ)

Example 5. *Figure 5 shows a design of a modulo-3 counter using 2 memory bits and an invariant property that both bits cannot become 1 together. We have*

$$transition(i, i+1) = (m_0(i+1) = \neg(m_0(i) \vee m_1(i))) \wedge (m_1(i+1) = (m_0(i) \wedge \neg(m_1(i))))$$

and $\phi_i = \neg(m_0(i) \wedge m_1(i))$. 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 bounded model checking (BMC) tool asks the following query to the Yices solver : $\wedge_{i \in \{1,2,\ldots,k\}} transition \wedge (\vee_{i \in \{1,2,\ldots,k\}} \neg \phi_i)$. For a correct design, the negated invariant property will never get satisfied and the BMC will return the unsat *result .*

Algorithm 4 K-Induction($k, vBlocks, \phi, model$)

1. $transition(1, k+1) = dump2yices(model, vBlocks, k)$
2. $\psi = (\wedge_{d \in \{1,2,\ldots,k\}} \phi_d) \wedge \neg \phi_{k+1}$
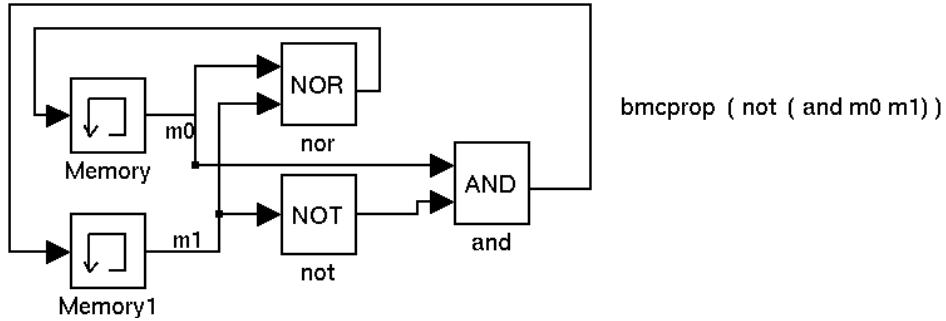3. check (transition, ψ)

Figure 5: Bounded Model Checking

6 Specifying and Checking 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. Confusion about units has led to some software-related failures. In September 1999, NASA's Mars Climate Orbiter was placed into a 57km orbit around Mars instead of a 140-150km orbit because some calculations used pound force instead of Newtons. We use Novak's classification [Nov95] of units into a 7-tuple consisting of *distance*, *mass*, *time*, *temperature*, *current*, *substance*, and *luminosity*. We have other dimensions like *angle* and *currency*, but these are distinct from the basic ones considered above. The dimension of a quantity is represented as a 7-tuple of integers. For example, $\langle 1, 0, -1, 0, 0, 0, 0 \rangle$ 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. Figure 6 shows the projectile design and the output of the dimension checker tool. The projectile subsystem block already shown in the Figure 1 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 pointwise summation of the dimensions.
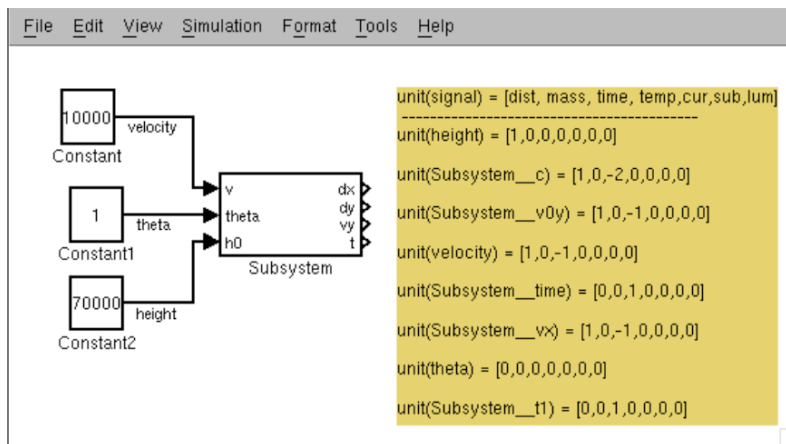

Figure 6: Unit and Dimensions

7 Related Work

Our work covers the verification and validation of MATLAB Simulink models through type annotations. We use the Yices SMT solver to both verify type correctness as well as to generate test cases. We focus in particular on expressive types that go beyond signal datatypes to capture dynamic ranges and relationships between signals. We also employ types to describe and check dimension and unit information. Finally, we employ bounded model checking and *k*-induction to verify invariant properties and generate counterexamples.

There is a substantial body of work [RdMH04, AC05, KAI$^+$09, AKRS08, CK03, Sys, TSCC05, WCM$^+$07] in all of the above areas. The Reactis tool [Sys] generates and simulates test cases for Simulink/Stateflow models including embedded C code. The Mathworks Simulink Design Verifier also performs test generation. The Simulink Design Verifier is also capable of generating test cases and for proving and refuting functional properties of Simulink blocks. The CheckMate tool [CK03] analyzes properties of hybrid systems represented by Simulink/Stateflow models using a finite state abstraction of the dynamics. The Honeywell Integrated Lifecycle Tools and Environment (HiLiTE) [BHSO07] from Honeywell Research uses static analysis to constrain the ranges of inputs in order to generate test cases, detect ambiguities and divide-by-zero errors, and unreachable code. The HiLiTE tool bases its analysis on numeric techniques on a floating-point representation, whereas our approach is based on symbolic constraint solving over the real numbers. The Simulink-to-Lustre translator defined by Tripakis, *et al* [TSCC05], performs type inference on Simulink models and the resulting Lustre output can be analyzed using model checkers. The Gryphon tool suite [WCM$^+$07] integrates a number of tools, including model checkers, for analyzing properties of Simulink/Stateflow models.

There is a long tradition of work on adding units and dimensions to type systems. These are surveyed in Kennedy's dissertation [Ken96] where he presents a polymorphic dimension inference procedure. Kennedy [Ken97] 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 centimetres. 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 [ACL$^+$05]. 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 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.

8 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 bounded model checking, we can extract the relationships between inputs and outputs of the Stateflow descriptions. We can also associate interface types that capture the temporal input-output behavior of Stateflow 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 [BHSO07]. The type system can also be extended to handle interface types [dAH01] 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.

9 Conclusions

We have outlined an approach to the partial verification of Simulink models through the use of an expressive type system. This type system can capture constraints on signals, the dimensions and units of signals, and the relationships between signals. Our annotations are already expressed in the constraint language of Yices. The resulting proof obligations are translated to the Yices constraint solver which is then used to check compatibility with respect to dimensions, generate counterexamples and test cases, and to prove type correctness. We also use Yices to perform bounded model checking and k-induction to refute or verify 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 cyber-physical systems. Our eventual goal is to use this capability to certify the correctness of such systems.

References

[AC05] M. M. Adams and Philip B. Clayton. ClawZ: Cost-effective formal verification for control systems. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005, Proceedings*, volume 3785 of *Lecture Notes in Computer Science*, pages 465–479. Springer, 2005.

[ACL+05] Eric Allen, David Chase, Victor Luchango, Jan-Willem Maessen, Sukyoung Ryu, Jr. Guy L. Steele, and Sam Tobin-Hochstadt. The Fortress language specification, version 0.618. Technical report, Sun Microsystems, Inc., 2005.

[AKRS08] Rajeev Alur, Aditya Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In *Proceedings of the 8th ACM & EMSOFT 2008, Atlanta, USA, October 19-24*, pages 89–98. ACM, 2008.

[BHSO07] D. Bhatt, S. Hickman, K. Schloegel, and D. Oglesby. An approach and tool for test generation from model-based functional requirements. In *Proc. 1st International Workshop on Aerospace Software Engineering*, 2007.

[CK03] A. Chutinan and B. Krogh. Computational techniques for hybrid system verification. *IEEE Transactions on Automatic Control*, 48(1):64–75, Jan. 2003.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120, 2001.

[DdM06] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *18th CAV 2006, Seattle, WA, USA, August 17-20*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[KAI+09] Aditya Kanade, Rajeev Alur, Franjo Ivancic, S. Ramesh, Sriram Sankaranarayanan, and K. C. Shashidhar. Generating and analyzing symbolic traces of Simulink/Stateflow models. In *21st CAV, Grenoble, France, June 26 - July 2, 2009*, volume 5643 of *LNCS*, pages 430–445. Springer, 2009.

[Ken96] A. J. Kennedy. *Programming Languages and Dimensions*. PhD thesis, University of Cambridge, 1996. Published as University of Cambridge Computer Laboratory Technical Report No. 391.

[Ken97]   Andrew J. Kennedy. Relational parametricity and units of measure. In *The 24th ACM POPL '97)*, pages 442–455, January 1997.

[Mey97]   Bertrand Meyer. Design by contract: Making object-oriented programs that work. In *TOOLS (25)*, page 360, 1997.

[Nov95]   Gordon S. Novak. Conversion of units of measurement. *IEEE TSE*, 21(8):651–661, 1995.

[ORSvH95]   Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[RdMH04]   John Rushby, Leonardo de Moura, and Grégoire Hamon. 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.

[Sys]   Reactive Systems. Model based testing and validation with Reactis, Reactive Systems inc., http://www.reactive-systems.com.

[TSCC05]   Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating discrete-time simulink to lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4):779–818, 2005.

[WCM+07]   Michael W. Whalen, Darren D. Cofer, Steven P. Miller, Bruce H. Krogh, and Walter Storm. Integration of formal analysis into a model-based software development process. In Stefan Leue and Pedro Merino, editors, *FMICS*, volume 4916 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2007.

A   Appendix

A.1   Satisfiability Modulo Theories and Yices

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 ⟨*identifier*⟩ :: ⟨*type*⟩⟨*expression*⟩): Defines a constant.

2. (assert ⟨*formula*⟩): Asserts a formula to the context.

3. (check): Checks if the context is satisfiable.

There are other commands for resetting the context, pushing and popping contexts, setting various flags, and invoking weighted satisfiability. As a simple example, Yices responds with unsat to the input

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

Yices also returns *unsat* on the following integer constraints.

```
(define i::int)
(assert (> i 0))
(assert (<= (* 2 i) 1))
```

On the other hand, if we relax the second constraint to a non-strict inequality, we get sat with an assignment of 0 for i.

```
(define i::nat)
(assert (<= (* 2 i) 1))
(check)
```

Yices can of course handle large problems with thousands of variables and constraints involving Booleans, arrays, bit vectors, and uninterpreted function symbols. We can run *Yices* in the batch mode : *./yices ex*1*.ys*. Yices also provides a basic typechecker.

A.2 Block Translation Details

Memory Blocks One memory element with initial value 70 can be translated in two consecutive clock cycles as follows:

```
(define Memory__Out1__time1 :: real)
(define Memory__In1__time1 :: real)
(assert (= 70 Memory__Out1__time1))
(define Memory__Out1__time2 :: real)
(define Memory__In1__time2 :: real)
(assert (= Memory__Out1__time2 Memory__In1__time1))
```

Basic Blocks We exploit the Yices operators directly to translate basic arithmetic and logical operator blocks. For example, *Product* block in Figure 1 is translated as

```
(define  Product__In1__time1 :: real )
(define  Product__In2__time1  :: real )
(define  Product__Out1__time1  :: real )
(assert (=  Product__Out1__time1  (*  Product__In1__time1
               Product__In2__time1 )))
```

Constants Simulink constant blocks 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 $int \rightarrow int \rightarrow t$ The translation process can catch the following problems via type checking rows and columns- (1) row-column mismatches between blocks, (2) calling out of indices (for vectors/arrays as well as matrices). For example, *Constant1* block in Figure 1 is translated as

```
(define  Constant1__Out1__time1  :: real )
(assert (=  Constant1__Out1__time1  2))
```

Signal Connectors For example, the signal c in Figure 1 adds the following translations

```
(define  c__time1  :: real )
(assert (=  c__time1  Constant__Out1__time1 ))
(define  Product3__In2__time1  :: real )
(assert (=   Product3__In2__time1   c__time1 ))
(define  Product2__In2__time1  :: real )
(assert (=  Product2__In2__time1  c__time1 ))
```