SimCheck : An Expressive Type-system for Simulink

Pritam Roy

Natarajan Shankar

University of California, Los Angeles

SRI International

April 14, 2010

support by NSF Grant CSR-EHCS(CPS)-0834810 and NASA Cooperative Agreement NNX08AY53A
1. Introduction

2. Model Translation to Yices

3. Type Annotation and Checking

4. Property Verification

5. Conclusion
Matlab/Simulink is very popular modeling language in the industry
Language of choice to design a complex embedded or control system
Easy to draw lines and boxes - drag and drop
A comprehensive set of function blocks
Matlab/Simulink is very popular modeling language in the industry
Language of choice to design a complex embedded or control system
Easy to draw lines and boxes - drag and drop
A comprehensive set of function blocks
Simulink Semantics

- A Simulink model is represented graphically by means of a number of interconnected blocks.
- Lines between blocks connect block outputs to block inputs and represent data flow signals.
- Blocks can be built from a large number of predefined library blocks, which can be nested in an arbitrary manner,
Blocks may have states, which may consist of a discrete-time and a continuous-time part.

The output of a block is computed by an output function, based on its input and its current state and time.

An update function calculates the next discrete state.
Why Type Checking is Important?

- type of a variable provide an abstraction of the data values the variable can store
- type checking: process of verifying and enforcing the constraints of types
- expressive type-checking catches most of the bugs eagerly (avoid crashing during execution)
Types in Simulink

- basic types are `Bool`, signed and unsigned integer types, tuples, vectors and matrices, and strings.
- Default type is `double`
- Ports in a block and connectors have types.
Types in Simulink

- basic types are Bool, signed and unsigned integer types, tuples, vectors and matrices, and strings.
- Default type is double
- Ports in a block and connectors have types.

Simulink has a limited capability for checking type correctness.

no systematic design-by-contract capability for Simulink.
What we need?

We need a capability that uses contracts to carry out

- Type inference for basic types,
- Annotate links with expressive types (over values, dimensions and units of physical variables)
- Generate test cases,
- Capture interfaces and behaviors,
- Monitor type conformance, and
- Verify type correctness relative to such expressive types

SimCheck : An Expressive Type-system for Simulink
What we need?

We need a capability that uses contracts to carry out

- Type inference for basic types,
- Annotate links with expressive types (over values, dimensions and units of physical variables)
- Generate test cases,
- Capture interfaces and behaviors,
- Monitor type conformance, and
- Verify type correctness relative to such expressive types

The SimCheck type system/tool can be used to specify and verify these requirements.
SimCheck Framework

- The whole package is built in Matlab language.
- integrated the type checker/verifier inside the Matlab environment editing customization file
- Simulink files have mdl extensions
- Used simulink APIs to obtain the parameters of blocks and connectors
Outline

1. Introduction
2. Model Translation to Yices
3. Type Annotation and Checking
4. Property Verification
5. Conclusion

Pritam Roy, N. Shankar

SimCheck: An Expressive Type-system for Simulink
SMT is the problem of determining satisfiability of formulas modulo background theories (e.g. linear arithmetic, arrays, data types, bit vectors etc.)

Yices is an SMT Solver developed at SRI International.

The input language of the Yices language in the semantics of a divide block:

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

For example, *Constant* block is translated as

```
(define ConstantOut1time1 :: real)
(assert (= ConstantOut1time1 1))
```
Basic Blocks

For example, *Constant* block is translated as

```
(define ConstantOut1time1 :: real )
(assert (= ConstantOut1time1 1))
```

*Sum* block is translated as

```
(define SumIn1time1 :: real )
(define SumIn2time1 :: real )
(define SumOut1time1 :: real )
(assert (= SumOut1time1 (+ SumIn1time1 SumIn2time1 )))
```
Memory with initial value 70 can be translated to two consecutive clock cycles as follows:

```
(define MemoryOut1time1 :: real)
(define MemoryIn1time1 :: real)
(assert (= 70 MemoryOut1time1))
(define MemoryOut1time2 :: real)
(define MemoryIn1time2 :: real)
(assert (= MemoryOut1time2 MemoryIn1time1))
```
User Given Types

- User can annotate the type constraints or constraints on the signals
- Constraints can be written on the blocks or globally

```
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 )
```
For an adder with inputs signals $x$ and $s$ and output $o$, we obtain

```scheme
(define dist(x) :: int)
(define dist(s) :: int)
(define dist(o) :: int)
(define x:: real)
(define s:: real)
(define o:: real)
(assert (= dist(x) dist(s)))
(assert (= dist(x) dist(o)))
(assert (= o (+ (* 254/10000 x) (* 1/100 s))))
```

simcheck : An Expressive Type-system for Simulink
Unit Checking

SimCheck: An Expressive Type-system for Simulink
Dependent Types and Refinement Types

- **Refinement types**: capture constraints on the signals to a function e.g. odd integer, non-zero divisor
- **Dependent types**: capture dependency between input ports or output port and input ports

![Thermostat Example with Dependent and Refinement Types](image)

**Figure**: Thermostat Example with Dependent and Refinement Types
Typechecking Incompatibility

- Two components may have different input assumptions and output guarantees
- May not work together
- If SMT solver returns UNSAT then there is no environment

![Simulink Diagram](image-url)
Test Case Generation

- For each signal $s$ and constraint $Ps(s)$, we want to show that $\neg Ps(s) \land \land_{r \neq s} Pr(r)$.
- Yices solver returns either a negative answer (unsat) or a positive answer with a satisfying variable assignment.
Property Verification

Bounded Model Checking

1. $\text{transition}(1, k + 1) = \text{dump2yices}(\text{model}, \text{vBlocks}, k)$
2. $\psi = \bigvee_{d \in \{1, 2, \ldots, k+1\}} \neg \phi_d$
3. check-sat(transition, $\psi$)
Property Verification

Bounded Model Checking

1. \( transition(1, k + 1) = dump2yices(model, vBlocks, k) \)
2. \( \psi = \forall d \in \{1, 2, \ldots, k + 1\} \neg \phi_d \)
3. \( \text{check-sat}(transition, \psi) \)

K-induction

1. \( transition(1, k + 1) = dump2yices(model, vBlocks, k) \)
2. \( \psi = (\land d \in \{1, 2, \ldots, k\} \phi_d) \land \neg \phi_{k+1} \)
3. \( \text{check-sat}(transition, \psi) \)
Related Tools

- Reactis: test generation and simulation of test cases including embedded C code.
- Mathworks SDV: test generation and prove and refute functional properties.
- CheckMate: analysis of properties of hybrid systems using a finite state abstraction of the dynamics.
HiLiTE: use of static analysis of input range for generation of test-cases, detection of ambiguities and divide-by-zero errors, robustness, and unreachable code.

Simulink-to-Lustre: translation for type inference on Simulink models and the resulting Lustre output can be analyzed using model checkers.

Gryphon: a number of tools, including model checkers, for analysis of Simulink/Stateflow models.

kind: bounded model checking, k-induction
Future Directions

- **Interface types** to capture temporal input-output behavior of Stateflow
- Map the bounded integers to their **fixed-point representations** or bit vectors.
- Reason about **Matlab functions blocks** via static analysis
- Cover **other properties** such as the robustness, error bounds for floating point computations,
- Verification of **model/code correspondence** through the use of test cases
- Translate Simulink models to **SAL** and **HybridSAL** for symbolic analysis
Conclusions

- Type system capture constraints, dimensions and units of signals, and the relationships between signals.
- Annotations expressed in the constraint language of Yices
- Proof obligations translated to constraint solver
- Obligations used to check compatibility w.r.t. dimensions, generate counterexamples and test cases, and to prove type correctness.
- Yices performs BMC and $k$-induction to refute or verify type invariants.
- Eventual goal is to use this capability to certify the correctness of such systems and monitor type conformance.
Thank You