SimCheck : An Expressive Type-system for Simulink [1]

Pritam Roy[1] Natarajan Shankar[2]

University of California, Los Angeles[1]
SRI International[2]

April 14, 2010

[1]supported by NSF Grant CSR-EHCS(CPS)-0834810 and NASA Cooperative Agreement NNX08AY53A

Outline

Simulink as Modeling Tool

- 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 as Modeling Tool

- 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

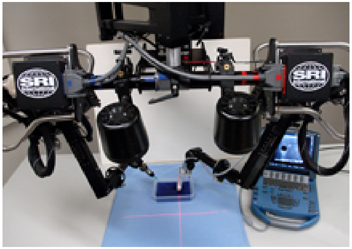


Figure: M7 Surgical Robot Arm

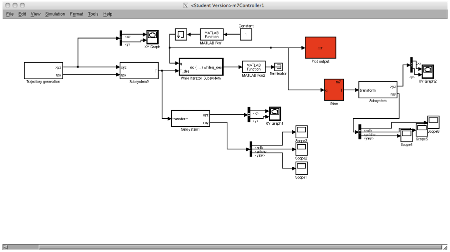
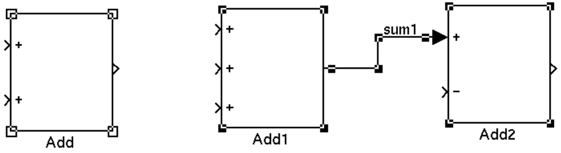


Figure: Simulink Model of M7

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,



Simulink Semantics

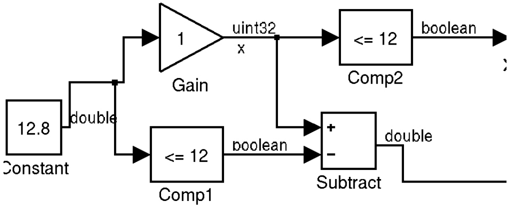
- Blocks may have states, which may consist of a discrete-time and a continuous-time part.
- The output of a block is computed by an output function, based on its input and its current state and time.
- An update function calculates the next discrete state.

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)

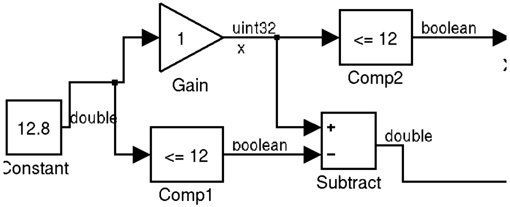
Types in Simulink

- 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.



- 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

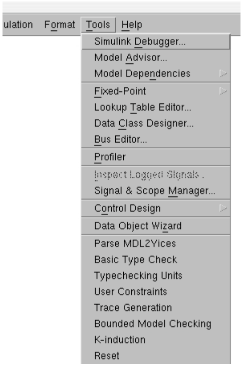
What we need?

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

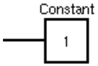
Yices - SMT Solvers

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

Basic Blocks

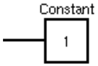
For example, *Constant* block is translated as



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

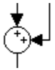
Basic Blocks

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 Blocks

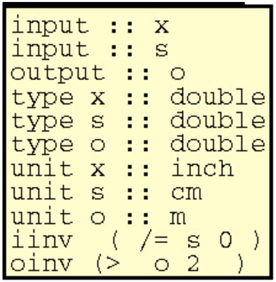
Memory with initial value 70 can be translated to two consecutive clock cycles as follows:

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

Outline

User Given Types

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

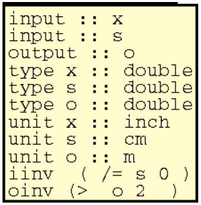


```
input  :: x
input  :: s
output  :: o
type x  :: double
type s  :: double
type o  :: double
unit x  :: inch
unit s  :: cm
unit o  :: m
iinv   ( /= s 0 )
oinv  (>  o 2   )
```

Figure: Annotation Block

Unit Checking

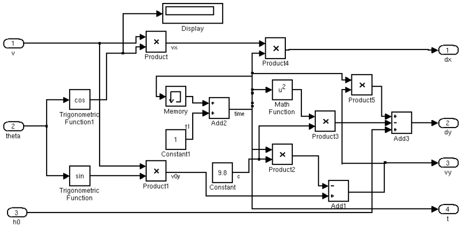
For an adder with inputs signals *x* and *s* and output *o*, we obtain

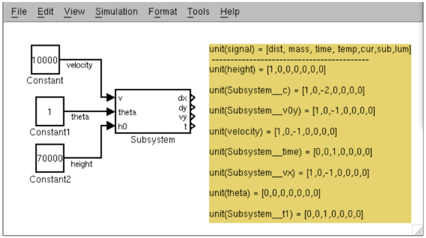


```
input :: x
input :: s
output :: o
type x :: double
type s :: double
type o :: double
unit x :: inch
unit s :: cm
unit o :: m
iinv  ( /= s 0 )
oinv (>  o 2  )
```

```
(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))))
```

Unit Checking

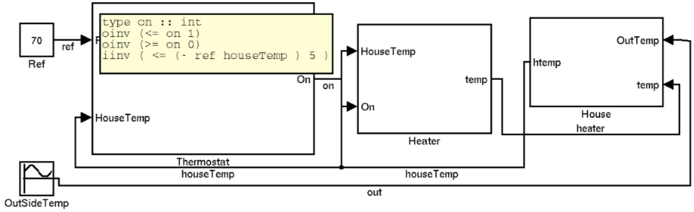




unit(signal) = [dist, mass, time, temp,cur,sub,lum]
----
unit(height) = [1,0,0,0,0,0,0]
unit(Subsystem__c) = [1,0,-2,0,0,0,0]
unit(Subsystem__v0y) = [1,0,-1,0,0,0,0]
unit(velocity) = [1,0,-1,0,0,0,0]
unit(Subsystem__time) = [0,0,1,0,0,0,0]
unit(Subsystem__vx) = [1,0,-1,0,0,0,0]
unit(theta) = [0,0,0,0,0,0,0]
unit(Subsystem__t1) = [0,0,1,0,0,0,0]

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

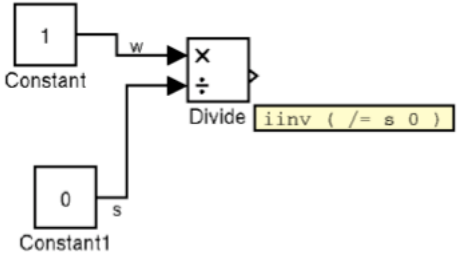


```
type on :: int
oinv (<= on 1)
oinv (>= on 0)
iinv ( <= (- ref houseTemp ) 5 )
```

Figure: Thermostat Example with Dependent and RefinementTypes

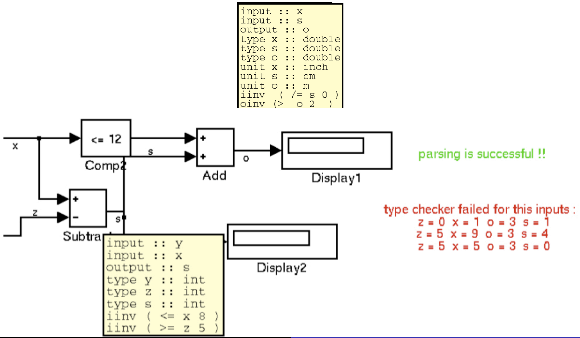
Typechecking Incompatibility

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



Test Case Generation

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



```
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  )
```

```
input :: y
input :: x
output :: s
type y :: int
type z :: int
type s :: int
iinv ( <= x 8 )
iinv ( >= z 5 )
```

```
parsing is successful !!
```

```
type checker failed for this inputs :
 z = 0  x = 1  o = 3  s = 1
 z = 5  x = 9  o = 3  s = 4
 z = 5  x = 5  o = 3  s = 0
```

Outline

Property Verification

Bounded Model Checking

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

Property Verification

Bounded Model Checking

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

K-induction

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-sat (transition,ψ)

Outline

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.

Related Tools

- 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