11-2007

Machine-assisted proof support for validation beyond Simulink

Chunqing CHEN

Jin Song DONG

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Machine-Assisted Proof Support for Validation Beyond Simulink

Chunqing Chen, Jin Song Dong, and Jun Sun

School of Computing
National University of Singapore
{chenchun,dongjs,sunj}@comp.nus.edu.sg

Abstract. Simulink is popular in industry for modeling and simulating embedded systems. It is deficient to handle requirements of high-level assurance and timing analysis. Previously, we showed the idea of applying Timed Interval Calculus (TIC) to complement Simulink. In this paper, we develop machine-assisted proof support for Simulink models represented in TIC. The work is based on a generic theorem prover, Prototype Verification System (PVS). The TIC specifications of both Simulink models and requirements are transformed to PVS specifications automatically. Verification can be carried out at interval level with a high level of automation. Analysis of continuous and discrete behaviors is supported. The work enhances the applicability of applying TIC to cope with complex Simulink models.

Keywords: Simulink, Real-Time Specifications, Formal Verification, PVS.

1 Introduction

Simulink [18] is popular in industry for modeling and simulating embedded systems. It is deficient to handle requirements of high-level assurance and timing analysis. Formal methods have been increasingly applied to the development of embedded systems because of their rigorous semantics and powerful verification capability [15]. Previously, we showed the idea of applying Timed Interval Calculus (TIC) [10], a formal notation of real-time systems to complement Simulink [5]: an automatic translation from Simulink models to TIC specifications preserves the functional and timing aspects; important timing requirements can hence be formally validated by the well-defined TIC reasoning rules and the strong support of mathematical analysis in TIC.

Currently, the validation is accomplished by hand. When verifying complex Simulink models, it becomes difficult to ensure the correctness of each proof step and to manage all proof details manually. Thus, developing machine-assisted proof support is necessary and important to ease the analysis beyond Simulink.

Simulink models usually involve continuous dynamics, and important timing requirements often concern behavior over arbitrary (infinite) intervals. These features make the automated verification of Simulink models challenging. An

approach, i.e., model checking [6] has successfully handled finite state transition systems with its fully automatic proving process. Nevertheless the discretization abstraction of infinite state transition systems can decrease the accuracy when analyzing properties of continuous dynamics (e.g., space distance [22]). On the other hand, theorem proving [4] can directly deal with infinite state transition systems with powerful proof methods (e.g. mathematical induction). Higher order theorem proving systems such as PVS [24], HOL [11], and Isabelle [23] support expressive input forms and automated proof capabilities (e.g., automated linear arithmetic reasoning over natural numbers). A recently developed NASA PVS library [1] formalizes and validates integral calculus based on the work [8] that supports elementary analysis. The library has been successfully used to verify a practical aircraft transportation system [21] which involves complex continuous behavior. In this paper, we apply PVS as a framework to encode and verify the TIC models generated from Simulink. The NASA PVS library allows us to rigorously represent and analyze continuous Simulink models.

We firstly construct the TIC denotational semantic models and validate the TIC reasoning rules in PVS. Based on the encoding, we define a collection of PVS *parameterized types* which correspond to the TIC library functions of Simulink library blocks. The TIC specifications are automatically transformed into PVS specifications. The transformation preserves the hierarchical structure. We define a set of rewriting rules to simplify the proving process and keep certain detailed TIC semantic encodings transparent to users. Hence we can formally validate Simulink models at interval level with a high grade of automation: powerful proving capability (including automatic type checking) of PVS guarantees the correctness of each reasoning step; proofs at low level can be automatically discharged, mainly by the decision procedures on sets and the propositional simplifications over real numbers in PVS. We have successfully validated continuous and hybrid systems represented in Simulink against safety and bounded liveness requirements.

The rest of the paper is organized as follows. In section 2, we brief the work on representing Simulink models in TIC followed by an introduction of PVS. The encoding of the primary TIC semantics and reasoning rules is presented in Section 3. Section 4 defines the library of PVS parameterized types. In the next section, the transformation strategy is illustrated with a non-trivial hybrid control system. Section 6 shows the benefits of the rewriting rules and the facilities of our approach by formally validating the control system in PVS. Related works are discussed in section 7. Section 8 concludes the paper with future work.

2   Background

2.1   Simulink in Timed Interval Calculus (TIC)

A Simulink [18] model is a wired block diagram that specifies system behavior by a set of mathematical functions over time. A block can be either an elementary block or a wired block diagram for a sub-model. An *elementary block*

denotes a primitive mathematical relationship over its inputs and outputs. Elementary blocks are generated from a rich set of Simulink *library blocks* by using the parameterization method. A *wire* depicts the dependency relationship between connected blocks. The source (destination) block can write (read) values to (from) a wire according to its *sample time* which is the execution rate of an elementary block during simulation. Simulink adopts *continuous time* as the unifying domain to support various systems (continuous, discrete or hybrid). Note that discrete systems behave piecewise-constantly continuously in Simulink.

Example 1. A brake control system is used as a running example to explain our idea and illustrate the results. The system aims to prevent a vehicle from over speeding by automatically enabling a brake device to decelerate the vehicle in time. The Simulink model is shown in Figure 1: each square box is an elementary library block, and each ellipse denotes an interface. The model consists of three subsystems, namely, subsystem *plant* depicting the physical speed behavior, subsystem *sensor* discretizing the speed, and subsystem *brake* controlling the brake device status based on the sensed speed. More details are provided in Section 5 where we translate the system with its requirements into PVS specifications, and here we select subsystem *sensor* to describe our previous work. The subsystem contains three components: two denote the interface (i.e., *speedS* and *speedR*), and block *detector* created from Simulink library block *ZeroOrderHold* stores the input value at each sample time point (the sample time is 1 second in the example) and keeps it till the next sample time point. Its simplified content is available in Figure 2.

We applied the Timed Interval Calculus (TIC) [10] to formally represent the Simulink denotational semantics, and developed a tool to automatically translate Simulink models into TIC specifications. The translation preserves the functional and timing aspects as well as the hierarchical structure [5].

TIC is set-theory based and reuses the Z [30] mathematical and schema notations. It extends the work in [17] by defining *interval brackets* to abstract time
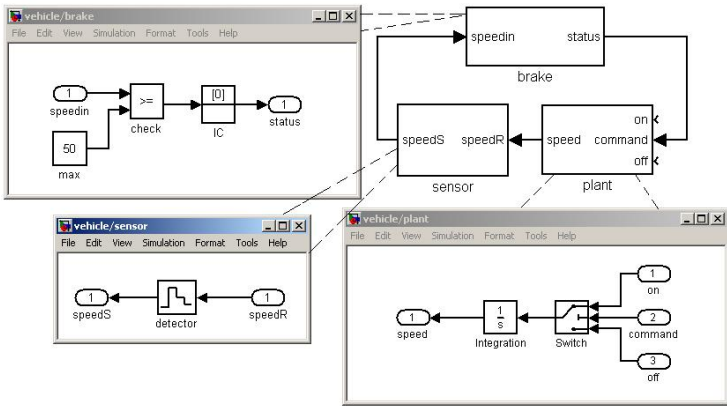

Fig. 1. The brake control system with its subsystems in Simulink

points and specifies properties at interval level. *Time domain* ($\mathbb{T}$) is non-negative real numbers, and *intervals* are continuous ranges of time points. In addition, α, ω, δ are *interval operators* in TIC to return the infimum, supremum and length of an interval. Note that they can be used to explicitly access endpoints. *Timed traces* defined in TIC are total functions of time to depict physical dynamics, and integration and differentiation have been rigorously defined in [9]. Each pair of interval brackets denotes a set of intervals during which an enclosed predicate holds at *all* time points. There are four *basic interval types* according to the endpoint inclusion. When the involvement of endpoints is unspecified, a *general interval type* is defined to cover all types of intervals. Hence system behavior can be modeled by predicates as relations over intervals. To manage the TIC specifications, *TIC schemas* are adopted: a TIC schema groups a collection of variables in the declaration part and constrains the relationships among the variables at the interval level in the predicate part. A set of well-defined TIC reasoning rules captures properties over sets of intervals and is used to verify complex systems.

We defined a set of *TIC library functions* to capture the denotational semantics of Simulink library blocks, i.e., mathematical functions between their inputs and outputs over time. Each TIC library function accepts a collection of arguments that correspond to Simulink library parameters, and returns a TIC schema that specifies the functionality of an instantiated library block. For example, function *ZOH* shown in Figure 2 preserves the sample time value (i.e., variable *st*) and describes the discrete execution in each sample time intervals (where interval brackets   represent a set of *left-closed, right-open* intervals).

The translation from Simulink models into TIC specifications is in a bottom-up manner. Elementary blocks are translated into TIC schemas by applying appropriate TIC library functions to relevant Simulink parameters. For example, schema *vehicle_sensor_detector* below is constructed by passing the sample time value to the *ZOH* function. Note that symbol "*_*" is used to retain the hierarchical order in Simulink models (*vehicle_sensor_detector* indicates that block *detector* is a component of system *sensor* which is a subsystem of system *vehicle*). Simulink diagrams are converted into TIC schemas. Specifically, the schemas declare each component as an instance of a TIC schema that represents the corresponding component, and each wire is expressed by an equation that consists of variables from the declaration. For example, schema *vehicle_sensor* in Figure 2 captures its three components and the connections.

2.2 Prototype Verification Systems (PVS)

PVS [24] is an integrated environment for formal specification and formal verification. The specification language of PVS is based on the classic typed, higher-order logic. Built-in types in PVS include *Boolean*, *real numbers*, *natural numbers* and so on. Standard predicate and arithmetic operations, such as conjunction (`AND`) and addition ($+$) on these types are also defined in PVS. Types can be defined starting from the built-in types using the type constructions. For example, *record* types are of the form $[\#a_1 : t_1, \ldots, a_n : t_n\#]$, where the a_i are named

```
Block {
  BlockType        SubSystem
  Name             "sensor"
  System {
    Block {
      BlockType        Inport
      Name             "speedR" }
    Block {
      BlockType        ZeroOrderHold
      SampleTime       "1"
      Name             "detector"}
    Block {
      BlockType        Outport
      Name             "speedS" }
    Line {
      SrcBlock         "speedR"
      DstBlock         "detector" }
    Line {
      SrcBlock         "detector"
      DstBlock         "speedS" }}}
```

$$ZOH : \mathbb{T} \to \mathbb{P}[In_1, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T}]$$

$$\forall\, t : \mathbb{T} \bullet ZOH(t) = [In_1, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T}\ |$$
$$st > 0 \wedge st = t \wedge$$
$$\langle\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st\rangle =$$
$$\langle Out = In_1(\alpha)\rangle]$$

$$vehicle\_sensor\_detector \,\widehat{=}\, ZOH(1)$$

$$\underline{\quad vehicle\_sensor \quad\quad\quad\quad\quad\quad\quad\quad}$$
$$speedR, speedS : \mathbb{T} \to \mathbb{R}$$
$$detector : vehicle\_sensor\_detector$$
$$\rule{5cm}{0.4pt}$$
$$\mathbb{I} = \langle speedR = detector.In1\rangle$$
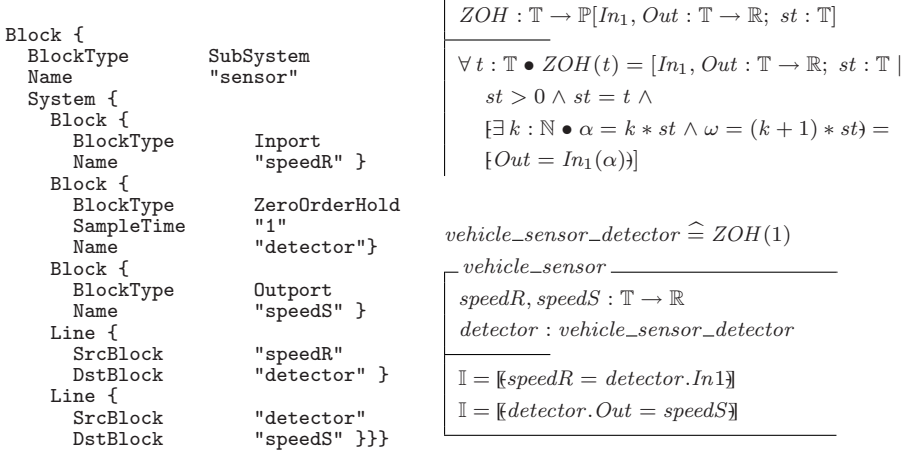$$\mathbb{I} = \langle detector.Out = speedS\rangle$$

Fig. 2. The *sensor* subsystem in Simulink and TIC

record accessors and the t_i are types. Elements of a record can be referenced by using the projection functions: 'a_i for instance.

Functions in PVS are total, and partial functions are supported by *predicate subtype* and *dependent types* which restrict the function domain. In addition, functions in PVS can share the same name as long as the types of their parameters are different. PVS specifications are organized into *theories*, which usually contain type declaration, axioms and lemmas. A theory can be reused in other theories by means of the *importing* clause.

The PVS theorem prover offers powerful primitive proof commands that are applied interactively under user guidance. Proofs are performed within a *sequent calculus* framework. A proof obligation consists of a list of assumptions $A_1, \ldots, A_n$ as *antecedents* and a list of conclusions $B_1, \ldots, B_m$ as *consequents*. It denotes that the conjunction of the assumptions implies the disjunction of the conclusions.

Primitive proof commands deal with propositional and quantifier rules, induction, simplification and so on. Users can introduce proof *strategies* which are constructed from the basic proof commands to enhance the automation of verification in PVS.

PVS contains many built-in theories as libraries which provide much of the mathematics needed (e.g. *real numbers* and *set*) to support verification. Recently, the NASA PVS library[1] extends the existing PVS libraries by providing means of modeling and reasoning about hybrid systems. The library formalizes the mathematical element analysis such as *continuity*, *differentiation* and *integration*, and contains many lemmas and theorems for manipulating these notations.

---

[1] It is available at `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`

3   Primitive Encoding of TIC

We construct the TIC denotational semantic model in PVS in a bottom up fashion. Each subsection below corresponds to a PVS theory. Complex theories can hence reuse the simple ones. The encoding forms a foundation to generate PVS specifications for TIC specifications and support TIC verification by formalizing and validating all TIC reasoning rules in PVS[2].

Time and Interval. The Time domain is represented by the PVS built-in type, i.e., *nnreal* for nonnegative real numbers. `Time: TYPE = nnreal`.

An interval is a tuple made up by two elements: The first specifies the interval type, i.e., `InterVal_Type: TYPE = {OO, CO, OC, CC}` (e.g., CO indicates that the interval type is left-closed, right-open). The second is a pair which denotes the starting point (stp) and the ending point (etp).

```
GenInterVal: TYPE = [invt: Interval_Type, {stp: Time, etp: Time | stp <= etp}]
```

The *general* interval type (*II*) captures the relation between basic interval types and endpoints. For example, when the interval is both-closed ($gi'1 = CC$), its ending point can equal the starting point ($gi'2'1 <= gi'2'2$ where symbols '1 and '2 are the projection operators of PVS for accessing a tuple). Note that when an interval is *general*, it can be one of the four basic interval types. We apply the *predicate subtype* mechanism of PVS to define basic interval types (e.g. the type of left-closed, right-open intervals, *COInterVal* is given below).

```
II: TYPE = { gi : GenInterVal | (gi'1 = CC and (gi'2)'1 <= (gi'2)'2) or
      ((gi'1 = OO or gi'1 = OC or gi'1 = CO) and (gi'2)'1 < (gi'2)'2)};
COInterVal: TYPE = {i: II | i'1 = CO}
```

Timed Trace and Interval Operators. The type of timed traces is a total function from time to real number. The interval operators, α, ω, and δ are functions from interval to time (We show the encoding of α here due to the size limit, and variable i is a variable of type *II*).

```
Trace: TYPE = [Time -> real];
ALPHA(i): Time = (i'2)'1;   % i: var II
```

Expressions and Predicates. Though time and intervals are abstracted in TIC specifications for concise modeling, they need to be explicitly accessible when interpreting expressions and predicates in PVS.

A basic element of TIC can have different types. Specifically, it can be a timed trace, an interval operator, or a constant. To *unify* different types into one type during the encoding, function *LIFT* is defined in the way which accepts three kinds of parameters (where the second and third parameters are the time and intervals respectively) and returns real numbers. Note that this is accomplished

---

[2] The complete PVS specifications of the TIC semantics and the reasoning rules are available at `http://www.comp.nus.edu.sg/~chenchun/PVSTIC`

by the *overloading* mechanism of PVS. For example, a timed trace is evaluated at a given time point while a constant is unchanged regardless of the time points and intervals.

```
LIFT(x)(t, i): real = x;        % t : var Time, x: var real
LIFT(tr)(t, i): real = tr(t);   % tr : var Trace
LIFT(tm)(t, i): real = tm(i);   % tm : var Term
```

Expressions in TIC are constructed by applying mathematical operators (including calculus operators, e.g., " $\int$ ") to the basic elements and other subexpressions. As time and intervals are required by the *LIFT* function, they are passed down to the constituent sub-expressions. The *propagation* stops when all the sub-expressions are primitive elements. Similarly way is applied to analyze predicates which are formed from applying mathematical relation over expressions or predicate logic on constituent sub-predicates. We show below the type declarations of the expressions and predicates, a subtraction expression, and a disjunctive predicate:

```
TExp: TYPE = [Time, II -> real];
-(el, er)(t, i): real = el(t, i) - er(t, i);   % el, er: var TExp
TPred: TYPE = [Time, II -> real];
or(pl, pr)(t, i): bool = pl(t, i) or pr(t, i); % pl, pr: var TPred;
```

An important feature of TIC is that the elemental calculus operations are supported, in particular *integration* and *differentiation*. Their definitions are formalized precisely in the NASA PVS library, and hence we can directly represent them in PVS. For example, the integral operation of TIC encoded below uses function *Integral* from the NASA PVS library where expressions *el* and *er* denote the bounded points of an integral.

```
TICIntegral(el, er, tr)(t, i): real = Integral(el(t, i), er(t, i), tr);
```

Quantification in TIC is supported in PVS by defining a *higher-order function* from the range of the bounded variable to the quantified predicate. For example, if the range of the bounded variable is natural numbers, and then the type of the quantification is from natural numbers to TIC predicate. Note that the following representation adopts the existence quantifier (*EXISTS*) of PVS directly.

```
QuaPred: TYPE = [nat -> TPred]; qp: var QuaPred;
exNat(qp)(t, i): bool = EXISTS (k: nat): (qp(k)(t, i))
```

TIC Expressions. A TIC expression is either formed by the interval brackets or a set operation on other TIC expressions. We present the way of encoding the interval brackets and concatenation operation below as they are defined special in TIC. Other TIC expressions can be constructed by using the PVS *set* theory.

In TIC, a pair of interval brackets denotes a set of intervals during which an enclosed predicate holds *everywhere*. Firstly function t_in_i detects if a time point is within an interval according to the interval type. Next, function *Everywhere?* checks whether a predicate is true at all time points within an interval. Lastly

the set of desired intervals is formed by using the set constructor in PVS. For example, the definition of the interval brackets (⟦ ⟧) which return a set of general intervals is shown below:

```
t_in_i(t, i): bool = (i'1 = OO and t > (i'2)'1 and t < (i'2)'2) or
  (i'1 = OC and t > (i'2)'1 and t <= (i'2)'2) or
  (i'1 = CO and t >= (i'2)'1 and t < (i'2)'2) or
  (i'1 = CC and t >= (i'2)'1 and t <= (i'2)'2);
Everywhere?(pl, i): bool = forall t: t_in_i(t, i) => pl(t, i);
AllS(pl): PII =  {i | Everywhere?(pl, i)}; % PII: TYPE = setof[II];
```

In TIC, concatenations are used to model sequential behavior over intervals. A concatenation requires the connected intervals to meet *exactly*, i.e., no gap and no overlap. Note that there are eight correct ways to concatenate two intervals based on the inclusion of their endpoints. Here we just consider one situation that a set of left-closed, right-open intervals is the result of linking two sets of left-closed, right-open intervals: the absence of gap is guaranteed by the equivalence of the connected endpoints (i.e., the ending point of *co*1 equals the starting point of *co*2), and the overlap is excluded by restricting the types of the connected endpoints (i.e., *co*1 is right-open while *co*2 is left-closed).

```
concat(cos1, cos2): PCC = {c : COInterVal |   % cos1 is a set of CO intervals
  exists (co1 : cos1), (co2: cos2):           % cos2 is a set of CO intervals
    OMEGA(co1) = ALPHA(co2) and ALPHA(co1) = ALPHA(c) and OMEGA(co2) = OMEGA(c)}
```

Based on the above encoding, we can formalize and validate the TIC reasoning rules in PVS. They capture the properties of sets of intervals and the concatenations and used to verify TIC specifications at the level of intervals. We have checked all rules stated in [10, 2], and hence they can be applied as proved lemma when verifying TIC specifications of Simulink models in PVS in the following sections.

4   Constructing PVS Library Types

Simulink library blocks create elementary blocks by instantiating parameters specific values. Similarly, we previously defined a set of TIC library function to represent the library blocks. To be specific, an instantiation of a library block is modeled as an application of a TIC library function. In this section, we construct a library of PVS parameterized types for the TIC library functions. In this way we produce concise PVS specifications for Simulink elementary blocks, and keep a clear correspondence of mathematical functions denoted in different notations.

A TIC library function accepts a set of parameters and returns a TIC schema, where the inputs, outputs and relevant parameters of an elementary block are defined as variables with their corresponding types, and the functional and timing aspects are captured by constraints among the variables.

We represent each TIC library function by a PVS *parameterized* type, which declares a type based on parameters. The parameters are the ones of a TIC library function, and the declared type is a *record type*. The record type models a generated schema by a TIC library function: variables are the *record accessors*;

and each predicate is represented by a constraint restricting the type domain of an associated accessor which is used to construct a set of records. Note that there are two categories of schema predicates: one indicates the relations between the declared variables and the TIC function parameters (or constants); the other captures the relations between variables denoting the inputs and outputs of Simulink blocks. For the first category of predicates, they constrain the domains of the corresponding variables; and the predicates of the second category restrict the domains of the outputs.

Taking TIC library function *ZOH* from Section 2.1 as an example, the PVS library type, *ZOH* below represents a record type that contains three accessors, i.e. *st*, *In*1 and *Out*. The TIC predicates constrain the type domains of relevant accessors. To be specific, the first two predicates (that belong to the first category) are the criteria for assigning sample time value correctly, and the last predicate (which satisfies the second category) is used to express the behavior of timed trace *Out*. Note that the PVS library type closes to the TIC library function in terms of the structure (where operator "o" is a function composition defined in PVS).

```
ZOH (t: Time): TYPE = [# st : {temp: Time | temp > 0 AND temp = t},
  In1: Trace,
  Out: {temp: Trace |
     COS(exNat(lambda(k: nat): LIFT(ALPHA) = LIFT(k) * LIFT(st) AND
                               LIFT(OMEGA) = (LIFT(k) + LIFT(1)) * LIFT(st)))
     = COS(LIFT(temp) = (LIFT(In1) o LIFT(ALPHA)))} #]
```

Continuous library blocks are important in Simulink modeling. They are directly represented in TIC with the well-defined operators [9] on elementary analysis. Using the recently developed NASA PVS library, these features can be preserved in PVS. For example, a continuous Simulink library block *Integrator* that performs an integration operation over its input is modeled formally in the TIC library function *Integrator*:

$$Integrator : \mathbb{R} \to \mathbb{P}[IniVal : \mathbb{R}; \ In_1 : \mathbb{T} \to \mathbb{R}; \ Out : \mathbb{T} \twoheadrightarrow \mathbb{R}]$$

$$\forall \, init : \mathbb{R} \bullet Integrator(init) =$$
$$[IniVal : \mathbb{R}; \ In_1 : \mathbb{T} \to \mathbb{R}; \ Out : \mathbb{T} \twoheadrightarrow \mathbb{R} \mid IniVal = init \land$$
$$Out(0) = IniVal \land \mathbb{I} = [\![Out(\omega) = Out(\alpha) + \int_\alpha^\omega In_1]\!]]$$

Operator $\twoheadrightarrow$ in the TIC specification indicates that the output is continuous. We retain this feature explicitly by function *continuous* from the NASA PVS library (specifically, *continuous(temp)* in the set constraint below). Note that PVS variable *fullset* denotes all valid intervals and maps to the TIC symbol, $\mathbb{I}$.

```
Integrator (init: real): TYPE = [# IniVal: {temp: real | temp = init},
  In1: Trace,
  Out: {temp: Trace | temp(0) = IniVal and continuous(temp) and
     fullset = AllS((LIFT(temp) o LIFT(OMEGA)) = (LIFT(temp) o LIFT(ALPHA)) +
                 TICIntegral(LIFT(ALPHA), LIFT(OMEGA), In1))}          #]
```

We found it useful to formalize functions as type declarations instead of conventional functions, although the second seems more intuitive. The reason is that

type information is available to the PVS prover, and hence we can minimize the number of type correctness conditions generated which are side effects of proof steps during the type checking in PVS. The benefit has also been investigated by Stringer-Calvert et al. [27]. They applied PVS to prove Z refinements for a compiler development. Their work focused on supporting partial functions of Z in PVS, however the way of handling schemas was missing. As we will show in the following sections, representing schemas as record types can facilitate both transformation and verification of TIC specifications.

5   Transformation of TIC Specifications

In this section, we present a strategy to transform TIC specifications which represent both Simulink models and requirements. The transformation preserves the hierarchical structure and has been implemented in Java for the automation.

5.1   Transforming TIC Schemas of Simulink Models

A Simulink model is a wired block diagram, and a block can be another wired block diagram. This hierarchical structure modeling feature eases the challenge of handling large scale systems. In the TIC specifications of Simulink models, using schemas as types is the way to preserve the hierarchical structure. When verifying these TIC specifications in PVS, it is important and necessary to retain the same hierarchical structure: on the one hand, we can support large scale systems in PVS, on the other hand the diagnostic information obtain at the level of PVS can be reflected back to the level of Simulink. The goal is achieved by using the record type of PVS as illustrated below.

The TIC specifications of Simulink models are TIC schemas and can be classified into two groups. One group represents the elementary blocks, and each schema is formed by an application of a TIC library function with relevant Simulink parameters. The transformation of this type of schemas is direct because of the PVS library types defined in the previous section. Namely, each schema is converted to a PVS record type which is an application of an appropriate PVS library type (i.e., the parameterized record type). The selection criterion is the name of the TIC library function by the one-to-one relationship between the TIC library functions and the PVS library types. For example, schema *vehicle_sensor_detector* of elementary block *detector* in Figure 1 is transformed to the following PVS specification:

$$vehicle\_sensor\_detector \mathrel{\widehat{=}} ZOH(1)$$

```
vehicle_sensor_detector: TYPE = ZOH(1);
```

The other group represents (sub)diagrams. A schema of this group models the diagram components in the declaration part and the connections in the predicate part. Taking the brake control system (see Figure 1) as an example, the whole system is made up of three subsystem, where each is represented by a variable

of schema type, and the wires between components are expressed as equalities in terms of intervals.

$$
\begin{array}{|l|}
\hline
\quad vehicle \\
\hline
plant : vehicle\_plant;\ sensor : vehicle\_sensor;\ brake : vehicle\_brake \\
\hline
\mathbb{I} = (\!| plant.speed = sensor.speedR |\!) \\
\mathbb{I} = (\!| sensor.speedS = brake.speedin |\!) \\
\mathbb{I} = (\!| brake.status = plant.command |\!) \\
\hline
\end{array}
$$

Similar to the way of dealing with the first group, each schema is transformed into a PVS record type. However, the difference is that the predicates constrain *all* accessors together rather then *some* accessors. The main reason is that unlike Simulink elementary blocks which denote relationships between inputs and outputs, the predicates of Simulink diagrams denote the connections among components, and it is thus difficult to determine which accessor should be constrained, especially when the wires form a cycle. For example, if we adopt the previous way, one possible PVS specification of schema *vehicle* is below:

```
vehicle: TYPE = [# plant: vehicle_plant,
  sensor: {temp: vehicle_sensor |
       fullset = AllS(LIFT(temp'speedR) = LIFT(plant'speed))},
  brake: {temp: vehicle_brake |
       fullset = AllS(LIFT(temp'speedin) = LIFT(sensor'speedS) AND
       fullset = AllS(LIFT(temp'status) = LIFT(plant'command))}     #]
```

It is not hard to observe that above PVS specification forces a dependency relation among three subsystems, and the correspondence between the PVS type declaration and the TIC library function is loose. To solve the problem, we apply the *predicate subtype* mechanism of PVS to define a set of records which represent the schema variables as accessors and satisfy the restrictions denoted by the schema predicates. In this way the transformed PVS specifications follow closely the schemas. Regarding the previous example, the schema is converted to the following PVS type declaration:

```
vehicle: TYPE = { temp: [#
  plant: vehicle_plant, sensor: vehicle_sensor, brake: vehicle_brake #] |
  fullset = AllS(LIFT(temp'plant'speed) = LIFT(temp'sensor'speedR)) AND
  fullset = AllS(LIFT(temp'sensor'speedS) = LIFT(temp'brake'speedin)) AND
  fullset = AllS(LIFT(temp'brake'status) = LIFT(temp'plant'command))       }
```

We remark that representing TIC schemas by the PVS record types supports the popular modeling technique in Z that uses schemas as types. As shown in the above PVS specification, the projection function (') acts like the selection operator (.) in Z to access a component. We remark that our way is different from Gravell and Pratten [12] who discussed some issues on embedding Z into both PVS and HOL. They interpreted Z schemas as Boolean functions of record types and it is thus difficult to handle the case where schemas declared as types.

5.2   Transforming Requirements

Requirements are predicates formed from the TIC specifications of Simulink models. They are directly converted into PVS *theorem* formulas based on the PVS specifications of the TIC schemas. With the primitive encoding of TIC semantics mentioned in Section 3, the way of transforming requirements is similar to the one analyzing schema predicates explained in the previous section. Below we skip details of the transformation due to the page limit, and provide the transformed PVS specifications of two requirements of the brake control system. They are used to illustrate the verification of TIC specifications in the next section.

One requirement checks the computational accuracy of the sensed speed. Namely, at any time the sensor should measure the speed within an accuracy of 10 meters/second. The TIC predicate and the translated PVS specification (where the used PVS variables such as *plant* can be found in Appendix B) are given below respectively.

$$Approximation == \forall \, v : vehicle \bullet \mathbb{I} = [\![\, | v.sensor.speedS - v.plant.speedR | \leq 10]\!]$$

```
Approximation: THEOREM forall (v: vehicle): fullset =
  AllS(LIFT(v'sensor'speedS) - LIFT(v'plant'speed) <= LIFT(10) AND
       LIFT(v'sensor'speedS) - LIFT(v'plant'speed) >= LIFT(-10));
```

Another requirement concerns the response time within which the brake device should respond. To be specific, if an interval of which the length is more than 1 second and during which the speed in the plant is not less than 50 meters/second, the brake must be enabled within 1 second and keep on till the end. The requirement is represented by the TIC predicate followed by the transformed PVS specifications:

$$Response == \forall \, v : vehicle \bullet [\![v.plant.speed \geq 50 \wedge \delta > 1]\!] \subseteq$$
$$[\![\delta < 1)\!\frown [\![v.brake.status = 1]\!]$$

```
Response: THEOREM forall (v: vehicle): subset?(
   CCS(LIFT(v'plant'speed) >= LIFT(50) AND LIFT(DETLA) > LIFT(1)),
   concat( COS(LIFT(DELTA) < LIFT(1)),
           CCS(LIFT(v'brake'status) = LIFT(1))));
```

We have demonstrated the strategy to automatically transform system design and requirements into PVS specifications. Important issues about different ways to represent TIC schemas have been discussed as well. The transformation preserves the hierarchical structure denoted in TIC specifications. In other words, systems specified in three notations (i.e. Simulink, TIC, and PVS) share the same viewpoint of structure. This feature improves the traceability when analyzing systems in different formalisms, for example, verifying TIC specifications in PVS can follow a similar proving procedure in TIC.

6   Validation Beyond Simulink in PVS

After transforming system designs and requirements into PVS specifications, we can formally verify Simulink models at the interval level (by the encoded TIC

reasoning rules) with a high grade of automation (by powerful proving support of PVS). In this section, we first define and validate a set of rewriting rules dedicated for Simulink modeling features to make verification more automated, followed by an illustration of verifying the brake control system to show that our developed tool supports the validation beyond Simulink such as dealing with open systems and checking timing properties.

6.1   Rewriting Rules for Simulink

Wires in Simulink models are represented by equations in TIC. Each equation consists of two timed traces that denote the connected block ports. When verifying TIC specifications in PVS, it is often to replace one timed trace by another when they both are in an equation. However, the substitution could be tedious in PVS since we need to expand the TIC semantic encoding thoroughly to make both time and interval explicitly for allowing the PVS prover to automatically discharge the proof obligation. To simplify the process as well as keep the detailed encoding transparent to users, we define a set of rewriting rules to easily handle the replacement of two equivalent timed traces. For example, rule *BB_ge_sub* substitutes two timed traces *tr1* and *tr2* over an inequality at the interval level.

```
BB_ge_sub: LEMMA forall (tr1, tr2, tr3: Trace):
  fullset = AllS(LIFT(tr1) = LIFT(tr2)) =>
  AllS(LIFT(tr1) > LIFT(tr3)) = AllS(LIFT(tr2) > LIFT(tr3));
```

Time domain of discrete systems in Simulink is decomposed into a sequence of left-closed, right-open intervals. We can hence define rewriting rules to facilitate the analysis of the discrete Simulink models. For example, rule *CO_to_All* states that for a predicate that is interval operator free[3], if it holds on *all* sample intervals (a *sample interval* is a left-close, right-open interval of which the endpoints are a pair of adjacent sample time points.), it is true in *any* interval. We remark that the rule is useful to check safety requirements of discrete systems.

```
st: var Time; tp: TPred;
CO_to_All: LEMMA st > 0 AND No_Term?(tp) =>
   subset?(COS(exNat( lambda(k: nat): LIFT(ALPHA) = LIFT(k) * LIFT(st) AND
                       LIFT(OMEGA) = (LIFT(k) + LIFT(1)) * LIFT(st))),
         COS(tp))
   => fullset = AllS(tp);
```

6.2   Reasoning About the Brake Control System

The brake control system is *open* as the exact function of the acceleration change is difficult to known. Simulink can check functional behavior for just closed systems by simulation. Moreover, timing requirements are difficult to specify in Simulink. In the following, we show how our approach can formally validate the nontrivial system with a high level of automation.

---

[3] We define PVS function *No_Term?* to check if the predicate is dependent on the interval endpoints or interval length.

Adding Environment Assumptions. When analyzing open physical environment, it is often that the exact functions of environment variables is unknown while loose information is available such as the ranges of environment inputs. The loose information can be easier modeled as constraints in TIC than in Simulink, and we can hence check the open systems represented in Simulink with the new constraints. For example, the range of the acceleration (the output of block *switch* in Figure 1) is known as below: input port *on* which denotes the value when the brake is enabled has a range between -10 and 0 $meters/second^2$; input port *off* which indicates the acceleration when the brake is disabled has a range from 0 to 10 $meters/seond^2$. The loose information is thus expressed by the following TIC predicate as well as the corresponding PVS specification.

$$InputAssump == \forall\, v : vehicle \bullet$$
$$\mathbb{I} = [\![-10 \le v.plant.on \le 0]\!] \land \mathbb{I} = [\![0 \le v.plant.off \le 10]\!]$$

```
InputAssump: LEMMA FORALL (v: vehicle):
      fullset = AllS(LIFT(v'plant'on) >= LIFT(-10) AND
                     LIFT(v'plant'on) <= LIFT(0))
  AND fullset = AllS(LIFT(v'plant'off) >= LIFT(0) AND
                     LIFT(v'plant'off) <= LIFT(10))
```

Checking the Approximation Requirement. The requirement concerns the functional behavior. It involves the analysis of continuous dynamics (e.g., the vehicle speed is the integration of the acceleration.), and it requires the checking over *all* types of intervals. The reasoning process is sketched below[4]:

1. We apply the rewriting rule, i.e., *CO_to_All* to reduce the type of intervals to be checked. Namely, we only need to observe the behavior over the sample intervals instead of all types of intervals. This is motivated by the discrete components, i.e., block *detector* is discrete in the *sensor* subsystem.
2. By the functionality of block *detector* (specified by the *ZOH* type defined in Appendix A) and the connection within the *vehicle* system and its *sensor* subsystem, we need to compare the output value of subsystem *plant* at the beginning of a sample interval and other values at *all* time points in the sample interval. The PVS *skolemization instantiation* mechanism allows to replacing all time points by an arbitrarily fixed time point, so we can just analyze the output values within a both-closed interval formed by two time points, i.e., the beginning endpoint and the fixed time point.
3. Since variable *speed* of subsystem *plant* is the output of continuous block *integration*, the analysis of continuous dynamics is needed. Based on the assumption of the environment encoded early, we can apply a lemma from the NASA PVS library to show that the difference of the speed at two specific time points mentioned in Step 2 is between -10 and 10. The lemma named *Integral_bound* relates the bound of the integration of a function and the bound of the function over a closed interval.

---

[4] The complete verification of both requirements in PVS is available at `http://www.comp.nus.edu.sg/~chenchun/brakecontrol`

```
Integral_bound: LEMMA a < b AND Integrable?(a,b,f) AND
        (FORALL (x: Closed_interval(a,b)): m <= f(x) AND f(x) <= M )
   IMPLIES m*(b-a) <= Integral(a,b,f) AND Integral(a,b,f) <= M*(b-a)
```

In the above analysis, the rewriting rules and the powerful proving capability of PVS facilitate the verification, and the NASA PVS library enhances the capability of our tool to handle continuous dynamics.

Checking the Response Requirement. Verifying Simulink models against timing requirements is non-trivial: the models usually involve continuous dynamics; and the timing requirements often investigate system behavior over arbitrary (infinite) intervals. Here we demonstrate how a typical timing requirement, *response* requirement of the brake control system can be validated.

The requirement involves all three subsystems, where subsystem *sensor* acts as a converter to pass the sensed speed to the brake. Note that the detector updates its output only at sample time points. The verification thus becomes nontrivial as each endpoint of an arbitrary interval may not be a sample time point. We adopt the *proof by exhaustion* method to solve the difficulty. An informal proof procedure is given below:

Firstly, we show that any arbitrary interval can be classified into one of *finite* cases. The following lemma, *Endpoints_general_form* states that given a positive sample time (ST), the interval endpoints can be expressed in a uniform format. The lemma has been checked correctly in PVS. Therefore we can group all intervals into *four* cases according to the values assigned to variables n and q (either 0 or positive real number).

```
Endpoints_general_form: LEMMA FORALL (i: II):
  EXISTS (m, p: nat), (n, q: nnreal): n < ST AND q < ST AND
    ALPHA(i) = m * ST + n AND OMEGA(i) = p * ST + q;
```

Next, we check that the validity of the *response* requirement in all cases. We consider the case where the intervals consist of multiple sample intervals as the basic case, as variables n and q are 0. Other types of intervals from the left three cases can be formed by appending an interval which lasts less than one sample time to the front or the back of a multiple sample intervals.

Lemma *Mult_Sample_Intervals* is defined specially for the basic case, and it facilitates the analysis over other three cases. The lemma allows a proof over a multiple sample intervals to be accomplished by reasoning about sub-proofs over every constituent sample interval. To be specific, the lemma checks the consequence relation between two predicates (*tp1* and *tp2*) which both are interval operator free. Note that by the skolemization instantiation method of PVS, we can only check the proof over one sample interval instead of every sample interval, and hence reduce the complexity of proving the lemma correctness.

```
Mult_Sample_Intervals: LEMMA No_Term?(tp1) AND No_Term?(tp2) AND x < y =>
    ((FORALL (k: {n: nat | x <= n AND n < y}):
      subset?( COS( tp1 AND LIFT(ALPHA) = LIFT(k) * LIFT(ST) AND
                    LIFT(OMEGA) = LIFT(k) * LIFT(ST)), COS(tp2)))
 => subset?( COS( tp1 AND LIFT(ALPHA) = LIFT(x) * LIFT(ST) AND
                  LIFT(OMEGA) = LIFT(y) * LIFT(ST)), COS(tp2)));
```

We remark that the lemma is generic to be applied to other systems which involve *periodical* behavior. For example, it can check the *exportable interval properties* defined in Interval Temporal Logic [20] which is a linear-time temporal logic with a discrete model of time.

The verification of the requirement is nontrivial: initial proof without using auxiliary lemmas takes more than 1000 steps. The complexity can be reduced by half after applying five proved lemmas: Four of five represent the validity of the requirement over four cases mentioned above according to lemma *Endpoints_general_form*, and the fifth captures the behavior over the primitive interval (i.e., the requirement holds everywhere in the interval of which the starting point is a sample time point and the interval length is not longer than one sample time). The reason for the decrease is that using lemmas we can save proof steps in many repeated sub-proofs.

Besides the method, proof by exhaustion, used here, we have also applied other powerful methods, such as *proof by contradiction* and *proof by induction* to verify safety requirements of continuous and hybrid Simulink models.

7   Related Works

Recently, there are a number of works on reasoning about Simulink models. Meenakshi et al. [19] used a model checker to analyze single-rate discrete Simulink models. Tripakis et al. [29] applied synchronous programming language *Lustre* to support multi-rate discrete Simulink models. Tiwari et al. [28] discretizing differential equations denoted by Simulink models into difference equations to construct discrete transition systems. Different from theirs our approach can directly represent and analysis continuous Simulink models. Gupta et al. [13] developed a tool to increase the modeling capability of Simulink. The tool emphasized on checking functional behavior which is also the main concern of the works mentioned previously, and hence timing analysis lack support. Jersak et al. [16] translated Simulink models into SPI models for timing analysis, although the translation abstracts the functional aspect. In contrast, our approach supports the validation over functional and timing behavior.

There were two preliminary works on supporting TIC using theorem provers. Dawson and Goré [7] validated TIC reasoning rules in Isabelle/HOL [23]. But the encoding of TIC semantics is incomplete, and it is hence difficult to support verification of TIC in general. Cerone [2] described many axioms on interpreting TIC expressions and predicates. However, the interpretation of the *concatenation* operator differed from the original [10], and his work dealt with just five reasoning rules. Our approach encoded the complete TIC semantics and handled all TIC reasoning rules in PVS. Some researchers have investigated the machine-assistant proof for a similar formal notation, Duration Calculus (DC) [31]. Skakkebaek and Shankar [26] developed a proof checker upon PVS, and Heilmann [14] applied Isabelle to support the mechanized proof. Chakravorty and Pandya [3] digitized a subclass of DC (i.e. Interval Duration Calculus) into another subclass for just discrete systems. As DC and its extensions [33, 32] describe systems behavior

without explicit reference to absolute time, they are limited to represent the constraints which are relevant to the values of interval endpoints. For example, the function of Simulink library block *Zero Order Hold* relies on specific sample time points. We remark that continuous behavior which is usually involved in Simulink models lacks of support in above works on TIC and DCs. For example, the *integral* function is either ignored or captured by a few axioms of limited properties. This is different from ours, as our approach can handle the analysis of continuous behavior.

8   Conclusion

In this paper, we extended our previous work which applied TIC to capture functional and timing aspects of Simulink diagrams as well as preserve the hierarchical structure. We developed a tool based on PVS to support the machine-assisted proofs for the Simulink models represented in TIC. A strategy has been implemented in Java to automatically transform the TIC specifications to PVS specifications. The transformed PVS specifications follow closely the hierarchical structured denoted by TIC specifications. Hence we can relate the diagnostic information of validation at the level of PVS to the level of Simulink.

We define a set of rewriting rules to simply the proving process and capture special characteristics of Simulink modeling. With the support of the NASA PVS library, we can directly analyze continuous dynamics which are usually involved in Simulink models. Validation in our framework can be carried out at the interval level with a high grade of automation. Open systems which are not checkable in Simulink can be reasoned about by specifying assumptions in TIC. Powerful mathematical proof methods (e.g. proof by induction) are useful to verify timing requirements (of safety and bounded liveness) beyond Simulink.

Currently, we are enhancing our framework in several directions. One is to develop graphical user interface (GUI) on top of the framework so as to facilitate the usability of our framework: transformation or proving can be executed by clicking buttons, and systems modeled in different notations can be shown in a better layout with colors to highlight the correspondence. Another is to improve the automation of the validation. Though verification of complex Simulink models is challenging, we are constructing more rewriting rules for special features of specific domain (e.g. hybrid control systems, the primary domain of Simulink modeling), and developing more PVS strategies to simplify the proving process. Extending the framework to support real-time systems development [25] of other formal notations is also one of our goals in the future.

Acknowledgements

References

1. Butler, R.W.: Formalization of the integral calculus in the pvs theorem prover. Technical report, NASA Langley Research Center, Hampton, Virginia (October 2004)
2. Cerone, A.: Axiomatisation of an interval calculus for theorem proving. Electr. Notes Theor. Comput. Sci. 42 (2001)
3. Chakravorty, G., Pandya, P.K.: Digitizing interval duration logic. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 167–179. Springer, Heidelberg (2006)
4. Chang, C.-L., Lee, R.C., Lee, R.C.-T.: Symbolic Logic and Mechanical Theorem Proving. Academic Press, Inc., London (1997)
5. Chen, C., Dong, J.S.: Applying Timed Interval Calculus to Simulink Diagrams. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 74–93. Springer, Heidelberg (2006)
6. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. on Prog. Lang. and Sys.s 16(5), 1512–1542 (1994)
7. Dawson, J.E., Goré, R.: Machine-checking the timed interval calculus. In: Sattar, A., Kang, B.-H. (eds.) AI 2006. LNCS (LNAI), vol. 4304, pp. 95–106. Springer, Heidelberg (2006)
8. Dutertre, B.: Elements of mathematical analysis in PVS. In: von Wright, J., Harrison, J., Grundy, J. (eds.) TPHOLs 1996. LNCS, vol. 1125, pp. 141–156. Springer, Heidelberg (1996)
9. Fidge, C.J., Hayes, I.J., Mahony, B.P.: Defining differentiation and integration in Z. In: ICFEM 1998, pp. 64–73 (1998)
10. Fidge, C.J., Hayes, I.J., Martin, A.P., Wabenhorst, A.: A Set-Theoretic Model for Real-Time Specification and Reasoning. In: Jeuring, J. (ed.) MPC 1998. LNCS, vol. 1422, pp. 188–206. Springer, Heidelberg (1998)
11. Gordon, M.J.C., Melham, T.F.: Introduction to HOL: A theorem proving environment for higher order logic. Cambridge University Press, Cambridge (1993)
12. Gravell, A.M., Pratten, C.H.: Embedding a formal notation: Experiences of automating the embedding of z in the higher order logics of pvs and hol. In: Grundy, J., Newey, M. (eds.) TPHOLs 1998. LNCS, vol. 1479, pp. 73–84. Springer, Heidelberg (1998)
13. Gupta, S., Krogh, B.H., Rutenbar, R.A.: Towards formal verification of analog designs. In: ICCAD 2004, pp. 210–217 (2004)
14. Heilmann, S.T.: Proof Support for Duration Calculus. PhD thesis, Department of Information Technology, Technical University of Denmark (1999)
15. Henzinger, T.A., Sifakis, J.: The embedded systems design challenge. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 1–15. Springer, Heidelberg (2006)
16. Jersak, M., Cai, Y., Ziegenbein, D., Ernst, R.: A transformational approach to constraint relaxation of a time-driven simulation model. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) ISSS 2003. LNCS, vol. 3233, pp. 137–142. Springer, Heidelberg (2004)
17. Mahony, B.P., Hayes, I.J.: A case-study in timed refinement: A mine pump. IEEE Transactions on Software Engineering 18(9), 817–826 (1992)
18. The MathWorks. Simulink - Simulation and Model-based Design - Using Simulink Version 6 (2004)
19. Meenakshi, B., Bhatnagar, A., Roy, S.: Tool for translating simulink models into input language of a model checker. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 606–620. Springer, Heidelberg (2006)

20. Moszkowski, B.C.: A complete axiomatization of interval temporal logic with infinite time. In: LICS 2000, pp. 241–252 (2000)
21. Muñoz, C., Carreño, V., Dowek, G.: Formal analysis of the operational concept for the Small Aircraft Transportation System. In: REFT 2006, pp. 306–325 (2006)
22. Muñoz, C., Carreño, V., Dowek, G., Butler, R.W.: Formal verification of conflict detection algorithms. I. Jour. on Soft. Tools for Tech. Trans. 4(3), 371–380 (2003)
23. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Springer, Heidelberg (2002)
24. Owre, S., Rushby, J.M., Shankar, N.: PVS: A Prototype Verification system. In: Kapur, D. (ed.) CADE-11. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)
25. Rischel, H., Cuéllar, J., Møk, S., Ravn, A.P., Wildgruber, I.: Development of safety-critical real-time systems. In: Bartosek, M., Staudek, J., Wiedermann, J. (eds.) SOFSEM 1995. LNCS, vol. 1012, pp. 206–235. Springer, Heidelberg (1995)
26. Skakkebæk, J.U., Shankar, N.: Towards a duration calculus proof assistant in pvs. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994. LNCS, vol. 863, pp. 660–679. Springer, Heidelberg (1994)
27. Stringer-Calvert, D.W.J., Stepney, S., Wand, I.: Using pvs to prove a z refinement: A case study. In: Jones, C.B. (ed.) FME 1997. LNCS, vol. 1313, pp. 573–588. Springer, Heidelberg (1997)
28. Tiwari, A., Shankar, N., Rushby, J.M.: Invisible Formal Methods for Embedded Control Systems. Proceedings of the IEEE 91(1), 29–39 (2003)
29. Tripakis, S., Sofronis, C., Caspi, P., Curic, A.: Translating discrete-time simulink to lustre. Trans. on Embedded Computing Sys. 4(4), 779–818 (2005)
30. Woodcock, J., Davies, J.: Using Z: Specification, Refinement and Proof. Prentice Hall International, Englewood Cliffs (1996)
31. Zhou, C.C., Hoare, C.A.R., Ravn, A.P.: A calculus of durations. Information Processing Letters 40, 269–276 (1991)
32. Zhou, C.C., Li, X.S.: A mean value calculus of durations. In: A classical mind: essays in honour of C. A. R. Hoare, Prentice-Hall International, Englewood Cliffs (1994)
33. Zhou, C.C., Ravn, A.P., Hansen, M.R.: An extended duration calculus for hybrid real-time systems. In: Hybrid Systems, pp. 36–59. Springer, Heidelberg (1993)

A   PVS Library Types of the Brake Control System

```
ZOH(t: Time): TYPE = [# st: {temp: Time | temp > 0 and temp = t}, In1: Trace,
   Out: {temp: Trace | COS(exNat(lambda (k: nat): LIFT(ALPHA) = LIFT(k) * LIFT(st) AND
                                          LIFT(OMEGA) = (LIFT(k) + LIFT(1)) * LIFT(st)))
                      = COS(LIFT(temp) = (LIFT(In1) o LIFT(ALPHA)))} #]

Integrator(x: real): TYPE = [# IniVal: {temp: real | temp = x}, In1: Trace,
   Out: {temp: Trace| temp(0) = IniVal AND
                    AllTrue((LIFT(temp) o LIFT(OMEGA)) = (LIFT(temp) o LIFT(ALPHA))
                              + TICIntegral(LIFT(ALPHA), LIFT(OMEGA), In1)) AND
                    continuous(temp)} #]

Switch_G(t: Time, x: real): TYPE = [# st: {temp: Time | temp = t},
   TH: {temp: real | temp = x}, In1, In2, In3: Trace,
   Out: {temp: Trace |
     IF st = 0 THEN AllS(LIFT(In2) > LIFT(TH)) = AllS(LIFT(temp) = LIFT(In1)) AND
                    AllS(LIFT(In2) <= LIFT(TH)) = AllS(LIFT(temp) = LIFT(In3))
     ELSE COS(exNat(lambda (k: nat): LIFT(ALPHA) = LIFT(k) * LIFT(st) AND
                                     LIFT(OMEGA) = (LIFT(k) + LIFT(1)) * LIFT(st)))
           = COS( ((LIFT(In2) o LIFT(ALPHA)) > LIFT(TH) =>
                      LIFT(temp) = (LIFT(In1) o LIFT(ALPHA)))
                 AND ((LIFT(In2) o LIFT(ALPHA)) <= LIFT(TH) =>
                      LIFT(temp) = (LIFT(In3) o LIFT(ALPHA))))
     ENDIF} #];
```

```
Relation_GE(t: Time): TYPE = [# st: {temp: Time | temp = t}, In1, In2: Trace,
  Out: {temp: BTrace |
    IF st = 0 THEN AllS(LIFT(In1) >= LIFT(In2)) = AllS(LIFT(temp) = LIFT(1)) AND
                   AllS(LIFT(In1) < LIFT(In2)) = AllS(LIFT(temp) = LIFT(0))
    ELSE COS(exNat(lambda (k: nat): LIFT(ALPHA) = LIFT(k) * LIFT(st) AND
                                    LIFT(OMEGA) = (LIFT(k) + LIFT(1)) * LIFT(st)))
       = COS( ((LIFT(In1) o LIFT(ALPHA)) >= (LIFT(In2) o LIFT(ALPHA))=>
               LIFT(temp) = LIFT(1))
              AND
              ((LIFT(In1) o LIFT(ALPHA)) < (LIFT(In2) o LIFT(ALPHA)) =>
               LIFT(temp) = LIFT(0)))
    ENDIF}  #];

InitCond(t: Time, x: real): TYPE = [# st: {temp: Time | temp = t},
   IniVal: {temp: real | temp = x}, In1: Trace,
   Out: {temp: Trace |
           IF st = 0 THEN subset?(AllS(LIFT(ALPHA) = LIFT(0)),
                                  AllS((LIFT(temp) o LIFT(0)) = LIFT(IniVal)))
                     AND  subset?(AllS(LIFT(ALPHA) > LIFT(0)),
                                  AllS(LIFT(temp) = LIFT(In1)))
           ELSE COS(LIFT(ALPHA) = LIFT(0) AND LIFT(OMEGA) = LIFT(st)) =
                COS(LIFT(temp) = LIFT(IniVal))
            AND COS(exNat1(lambda (k: posint): LIFT(ALPHA) = LIFT(k) * LIFT(st) AND
                                   LIFT(OMEGA) = (LIFT(k) + LIFT(1)) * LIFT(st)))
               = COS(LIFT(temp) = (LIFT(In1) o LIFT(ALPHA)))
           ENDIF} #];

Constant(x: real): TYPE = [# IniVal: {IniVal: real | IniVal = x},
   Out: {temp: Trace| AllTrue(LIFT(temp) = LIFT(IniVal))}  #];
```

B   Transformed PVS Specifications of the Brake Control System

For subsystem *brake*:

```
vehicle_brake_max: TYPE = Constant(50);
vehicle_brake_check: TYPE = Relation_GE(0);
vehicle_brake_IC: TYPE = InitCond(0, 0);
vehicle_brake: TYPE = {temp: [# speedin, status: Trace,
     max: vehicle_brake_max, check: vehicle_brake_check, IC: vehicle_brake_IC #] |
        fullset = AllS(LIFT(temp'speedin) = LIFT(temp'check'In1)) AND
        fullset = AllS(LIFT(temp'max'Out) = LIFT(temp'check'In2)) AND
        fullset = AllS(LIFT(temp'check'Out) = LIFT(temp'IC'In1)) AND
        fullset = AllS(LIFT(temp'IC'Out) = LIFT(temp'status))}
```

For subsystem *sensor*:

```
vehicle_sensor_detector: TYPE = ZOH(1);
vehicle_sensor: TYPE = {temp: [# speedS, speedR: Trace,
     detector: vehicle_sensor_detector #] |
        fullset = AllS(LIFT(temp'speedR) = LIFT(temp'detector'In1)) AND
        fullset = AllS(LIFT(temp'detector'Out) = LIFT(temp'speedS))}
```

For subsystem *plant*:

```
vehicle_plant_Integration: TYPE = Integrator(0);
vehicle_plant_Switch: TYPE = Switch_G(0, 0);
vehicle_plant: TYPE = {temp: [# on, off, command, speed: Trace,
     Switch: vehicle_plant_Switch, Integration: vehicle_plant_Integration #] |
        fullset = AllS(LIFT(temp'on) = LIFT(temp'Switch'In1)) AND
        fullset = AllS(LIFT(temp'command) = LIFT(temp'Switch'In2)) AND
        fullset = AllS(LIFT(temp'off) = LIFT(temp'Switch'In3)) AND
        fullset = AllS(LIFT(temp'Switch'Out) = LIFT(temp'Integration'In1)) AND
        fullset = AllS(LIFT(temp'Integration'Out) = LIFT(temp'speed))}
```

For whole system *vehicle*:

```
vehicle: TYPE = { temp: [# plant: vehicle_plant, sensor: vehicle_sensor,
     brake: vehicle_brake #] |
        fullset = AllS(LIFT(temp'plant'speed) = LIFT(temp'sensor'speedR)) AND
        fullset = AllS(LIFT(temp'sensor'speedS) = LIFT(temp'brake'speedin)) AND
        fullset = AllS(LIFT(temp'brake'status) = LIFT(temp'plant'command))}
```