

# A Verification System for Timed Interval Calculus

Chunqing Chen and Jin Song Dong and Jun Sun  
School of Computing  
National University of Singapore  
Computing 1, Law Link  
Singapore 117590  
{chenchun, dongjs, sunj}@comp.nus.edu.sg

## ABSTRACT

Timed Interval Calculus (TIC) is a highly expressive set-based notation for specifying and reasoning about embedded real-time systems. However, it lacks mechanical proving support, as its verification usually involves infinite time intervals and continuous dynamics. In this paper, we develop a system based on a generic theorem prover, Prototype Verification System (PVS), to assist formal verification of TIC at a high grade of automation. TIC semantics has been constructed by the PVS typed higher-order logic. Based on the encoding, we have checked all TIC reasoning rules and discovered subtle flaws. A translator has been implemented in Java to automatically transform TIC models into PVS specifications. A collection of supplementary rules and PVS strategies has been defined to facilitate the rigorous reasoning of TIC models with functional and non-functional (for example, real-time) requirements at the interval level. Our approach is generic and can be applied further to support other real-time notations.

## Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*Formal Methods, Validation*; D.2.1 [Software Engineering]: Requirements/Specifications—*Languages*

## General Terms

Verification

## Keywords

Real-Time Systems, Specification Language, Theorem Proving, PVS

## 1. INTRODUCTION

Real-time systems usually consist of computer-based controllers that interact with physical environment. As they have been used widely in safety-critical applications, it is necessary to formalize the properties of the environment for rigorously verifying these embedded real-time systems at early development stage [5]. Consequently, their formal models are desired to represent the discrete

logic of the controllers as well as the continuous dynamics of the environment [12]. It is also crucial that the modeling notation possesses powerful verification capability to formally check that the models fulfill requirements.

Timed Interval Calculus (TIC) [10] is a formal specification notation used to specify and reason about real-time systems. It is set-theory based and reuses the well-known formal notation  $Z$  [22] mathematical and schema notations. It adopts total functions of continuous time [13] to model analog and discrete state variables. In addition, differential and integral calculus are supported [9]. System properties (for example, periodic behavior) and requirements (for example, timing bounded response) are specified at interval level by measuring the set of intervals during which predicates over state variables hold everywhere. TIC includes a collection of reasoning rules which capture the generic features of sets of intervals. These rules and the support of mathematical analysis in TIC are useful to validate functional and non-functional (for example, real-time) requirements.

When analyzing complex systems in TIC, it is difficult to ensure the correctness of each proof step and to keep track of all proof details in a pencil-and-paper manner. It is necessary and important to develop a verification system for TIC to ease the proving. The verification usually requires mathematical analysis such as integral calculus used in modeling physical dynamics, and induction mechanism for dealing with infinite intervals and continuous time domain. These characteristics are not well supported by model checking [6] which usually applies discrete abstraction for infinite state space. The abstraction could decrease the analysis accuracy of continuous dynamics [15]. On the other hand, theorem proving [19] can handle infinite state space directly as well as support expressive specifications.

Instead of building a theorem prover from scratch, we choose one of the powerful generic theorem provers, Prototype Verification System (PVS) [18], because of its highly integrated environment for writing formal specifications and developing rigorous verification. The PVS specification language is based on higher-order logic associated with a rich type system. Its interactive theorem prover offers powerful automatic reasoning technique at low level such as arithmetic of *real numbers* and decision procedure over *sets*. Users can directly control proof development at high level, for example selecting proper user-defined strategies. A recently developed NASA PVS library [1] has formalized and validated the elementary calculus such as integration and derivation. The library has been successfully applied to verify a practical aircraft traffic system [15]. The above features of PVS are useful to achieve our goal of developing the mechanical proving support for TIC.

When constructing the TIC semantics in PVS, the higher-order logic eases the encoding, and the automatic type checking of PVS

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.



over intervals, concatenation operator  $\curvearrowright$  defined below connects two sets of intervals end-to-end, i.e., no gap and no overlap.

$$\frac{}{- \curvearrowright - : \mathbb{I} \times \mathbb{I} \mapsto \mathbb{I}}$$

$$\frac{\forall X, Y : \mathbb{I} \bullet X \curvearrowright Y = \{z : \mathbb{I} \mid \exists x : X; y : Y \bullet z = x \cup y \wedge (\forall t1 : x; t2 : y \bullet t1 < t2)\}}{}{}$$

By constraining the relation (e.g.,  $\subseteq$ ,  $=$ ) among sets of intervals, we can specify system properties and requirements at the interval level. For example, we can specify a periodic behavior below that a detector stores the input temperature  $Temp\_in$  every  $k$  time units.

$$\{\exists i : \mathbb{N} \bullet \alpha = i * k \wedge \omega = (i + 1) * k\}$$

$$= \{store = Temp\_in(\alpha)\}$$

Note that in the above TIC predicate, the TIC expression at the left decomposes the time domain into a sequence of left-closed, right-open intervals, and each interval lasts for  $k$  time units; the TIC expression at the right depicts the periodic update behavior of the value stored in the detector.

To manage TIC models in a structural manner, we adopt the Z schema notation to group a list of variables in its *declaration part* and specify the relations of the variables in its *predicate part*. For example, we represent the above detector in the following schema, where the continuity feature of the input temperature is denoted by the symbol  $\Leftrightarrow$  (defined in [9]).

<div style="text-align: center; border-bottom: 1px solid black; margin-bottom: 5px;"><i>Detector</i></div> <div style="margin-bottom: 5px;"><math>Temp\_in : \mathbb{T} \Leftrightarrow \mathbb{R}; store : \mathbb{T} \rightarrow \mathbb{R}</math></div> <div><math>\{\exists i : \mathbb{N} \bullet \alpha = i * k \wedge \omega = (i + 1) * k\}</math>  <math>= \{store = Temp\_in(\alpha)\}</math></div>
---

TIC contains 13 primitive rules on capturing the timing properties of sets of intervals and their concatenations. These reasoning rules are used to carry out the TIC verification at the interval level. For example, the rule given below states that for any non-pointer interval in which a predicate  $P$  holds, the interval can be decomposed into two concatenated sub-intervals, and in each sub-intervals the predicate is still true everywhere.

if  $\alpha$ ,  $\omega$ , and  $\delta$  do not appear free in predicate  $P$ , then

$$\llbracket P \wedge \delta > 0 \rrbracket = \llbracket P \rrbracket \curvearrowright \llbracket P \rrbracket$$

The interval brackets  $\llbracket \cdot \rrbracket$  denote a union of four basic types of interval brackets. Namely,  $\llbracket P \rrbracket = (P) \cup (P] \cup [P) \cup [P]$ . They are often used when predicates are independent of interval endpoints. Note that the above rule is valid provided the time domain is continuous.

Using TIC, we can specify important requirements, such as *safety* and *bounded liveness* requirements. The verification in TIC is to show if a design logically implies requirements. A proof is usually decomposed into several sub-proofs, and each sub-proof concentrates on a subsystem. A proof is a deduction that starts from hypothesis and processes in a forward manner. Each deductive step is reached by rigorously applying a fact (as an axiom), a TIC reasoning rule, a mathematic law, or a pre-proved requirement.

## 2.2 Prototype Verification System (PVS)

PVS [18] is an integrated environment for formal specification and formal verification. The specification language of PVS is based on classic typed, higher-order logic. Built-in types in PVS include *Boolean* (`bool`), *real numbers* (`real`), *natural numbers* (`nat`), and so on. Standard operations of predicate logic and arithmetic, such as conjunction (`and`), less-inequality (`<`) and addition (`+`) on the built-in types are also defined in PVS.

Functions in PVS are total, and partial functions are supported by *predicate subtypes* and *dependent types*. In addition, functions in PVS can share the same name as long as they have different parameter types. PVS specifications are organized in *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 prover maintains a *proof tree*, and the objective is to construct a complete proof tree of which all leaves are trivially true. Each node in a proof tree is a *proof goal* which is a sequent consists of a list of formulas named *antecedents* and a list of formulas called *consequents*. The intuitive interpretation of a proof goal is that the conjunction of the antecedents implies the disjunction of the consequents.

The PVS prover provides a collection of primitive proof commands when applied such as expanding definitions (`expand`) and eliminating quantifies (`skosimp`), to manipulate proof trees. A frequently used powerful proof command is `grind`, which does skolemization, instantiation, propositional simplification, rewriting, and applying decision procedures. Users can introduce more powerful *proof strategies* which combine basic proof commands so as to enhance the automation of verification in PVS.

PVS contains many built-in theories about logics, sets, numbers, and so on. They offer much of the mathematics needed to support specification and verification in PVS. Recently the NASA PVS library [1] has formalized the elements of mathematical analysis covering *continuity*, *differentiation*, and *integration*. The library has also developed a number of validated theories to support the rigorous analysis of continuous dynamics.

## 3. ENCODING TIC SEMANTICS IN PVS

In this section, we encode the TIC semantics into PVS. The encoding forms a foundation from which we formalize the reasoning rules, and further verification of TIC models. An important requirement is that the resulting PVS specifications should be concise in a structural way and ease the proving. We construct the PVS theories of the TIC semantics in a bottom-up manner, and each subsection below corresponds to a PVS theory. Simple theories are used to compose complex ones (The complete PVS specifications are available online [4]). We also adopt the hierarchy concept to avoid sub-goals explosion problem occurred during reasoning process (in particular the way to define bracket intervals and concatenation operation illustrated later). In addition, the flexible type declaration technique of PVS can reduce the size of the PVS specifications for different types of intervals.

### Time and Interval Domains

The time domain is represented by the PVS built-in type `nnreal`. Namely, `Time : TYPE = nnreal`.

An interval is a tuple in PVS, where the first element indicates the interval type (for example, `CO` for left-Close, right-Open intervals), and the second element is also a tuple denoting the starting and ending points of an interval. The corresponding PVS specification is below.

<code>Interval_Type : TYPE = {OO, OC, CO, CC};</code> <code>GenInterval : TYPE = [invt : Interval_Type,</code> <code>  {stp : Time, etp : Time   stp &lt;= etp}]</code>
---

All types of intervals is defined below by the type, `II`, where the PVS projection operators ‘1’ and ‘2’ are used to access components of a tuple. Note that the definition relates specific interval types with their endpoints. By using the *predicate subtype* mechanism from PVS, specific interval type is easily constructed. For example, `COInterval` denotes the type of left-closed, right-open intervals.

```

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

```

### Timed Trace and Interval Operators

A timed trace is a function from time to real number. We further define a type of *discrete* timed traces to model Boolean-valued states. A discrete timed trace has just two values, 0 and 1.

```

Trace: TYPE = [Time -> real];
BTrace: TYPE = [Time -> {x:real | x = 0 or x = 1}]

```

In TIC, interval operators are functions from intervals to time point. For example, operator  $\alpha$  returns the infimum of an interval. Note that in its definition<sup>1</sup> below we can just specify the functionality of the general interval type `II` rather than listing out individual cases for each basic interval type.

```

Term: TYPE = [II -> Time];
ALPHA(i): Time = i`2`1 % i : II

```

### Expressions and Predicates

As a modeling feature, time points and intervals are abstracted in the predicate within TIC interval brackets. However, they are required to be explicit when interpreting expressions and predicates in PVS. Therefore, both expressions and predicates are declared to be functions where time points and intervals compose the domain.

```

TExp: TYPE = [Time, II -> real];
TPred: TYPE = [Time, II -> bool];

```

Primitive elements in TIC form expressions and in turn predicates. An element can be one of the three types, namely, a constant, a timed trace and an interval operator. By applying the *overloading* mechanism of PVS, we define below a function `LIFT` to execute respective functionalities according to the element type. For example, the function returns the value at a time point for a timed trace while evaluates an interval operator by a given interval.

```

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

```

When representing an expression (a predicate), time points and intervals are the parameters passed to its constituent sub-expressions (sub-predicates or expressions). This parameter propagation stops until all sub-expressions are primitive elements. For example, the following PVS specifications model a subtraction and a disjunction of TIC. Note that the parameters such as `e1` and `p1` are also functions of time and intervals.

```

e1, er: var TExp
-(e1, er)(t, i): real = e1(t, i) - er(t, i)
p1, pr: var TPred;
or(p1, pr)(t, i): bool = p1(t, i) OR pr(t, i);

```

Elementary calculus is supported in TIC, in particular *integration* and *differentiation*. They can be modeled based on the PVS theories from the NASA PVS library which provides their formal definitions. For example, the integration ( $\int$ ) of a function `tr` over an interval is expressed in PVS below where the function `Integral` is adopted from the NASA PVS library.

```

TICIntegral(e1, er, tr)(t, i): real
= Integral(e1(t, i), er(t, i), tr);

```

<sup>1</sup>Characters following the symbol '%' are comments.

## TIC Expressions

A TIC expression represents a set of intervals. Here we illustrate the way of encoding the interval brackets and the concatenation operation. Other types of TIC expressions are constructed by the built-in functions from the PVS *set* theory.

In TIC, a pair of interval brackets denotes a set of intervals, and in each interval an enclosed predicate holds *everywhere*, namely, at every time point within the interval. Note that there are four basic types of interval brackets. We firstly model the semantics of the general interval brackets  $\llbracket \cdot \rrbracket$ , and then specify the encoding of specific basic types of interval brackets such as  $\{ \cdot \}$  by the predicate subtype technique. For example, function `ALLS` (for  $\llbracket \cdot \rrbracket$ ) and function `COS` (for  $\{ \cdot \}$ ) given below share the same function `Everywhere?` which checks if a predicate holds everywhere during an interval. The function `t_in_i` is used to detect if a time point is within an interval based on the interval type.

```

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): setof[II] = {i | Everywhere?(pl, i)};
COS(pl): setof[COInterval] =
{i: COInterval | Everywhere?(pl, i)};

```

A concatenation in TIC requires that any two connected intervals must meet *exactly*, namely, no overlap and no gap. There are thus eight correct concatenation ways from four basic interval types. Instead modeling each concatenation with their constraints, we represent all eight cases as one shown below. Function `concat` receives two sets of intervals, and each interval in the returned set is formed by two adjacent intervals.

```

ConcatType(l, r, re: II): bool =
(re`1 = OO AND ((l`1 = OC AND r`1 = OO) OR
(l`1 = OO AND r`1 = CO)))
OR (re`1 = CO AND ((l`1 = CC AND r`1 = OO) OR
(l`1 = CO AND r`1 = CO)))
OR (re`1 = OC AND ((l`1 = OO AND r`1 = CC) OR
(l`1 = OC AND r`1 = CO)))
OR (re`1 = CC AND ((l`1 = CO AND r`1 = CC) OR
(l`1 = CC AND r`1 = OC)));
concat(iisl, iisr: PII): PII =
{i | exists (i1, i2: II):
member(i1, iisl) AND member(i2, iisr) AND
OMEGA(i1) = ALPHA(i2) AND ALPHA(i1) = ALPHA(i) AND
OMEGA(i2) = OMEGA(i) AND ConcatType(i1, i2, i)};

```

Note that the adjacency of two intervals is checked by the set predicates which applies function `ConcatType`. The function captures all correct situations of concatenation in terms of the types of connected intervals. We hence create a hierarchical structure to represent the constraints. Namely, the function `ConcatType` can be treated as a lower level of the function `concat`. This way is useful in practice to avoid sub-goal explosion problem encountered in reasoning process in PVS: if we directly list the eight constraints of interval types in `concat`, the PVS prover will automatically split a current proof goal into eight sub-goals when expands the concatenation definition `concat`, although usually there are many repetitive proof commands in the proofs of eight sub-goals.

So far, we have demonstrated how to construct the TIC semantics in PVS, while the way of handling schemas and requirements will be given in Section 5. During the encoding, the overloading mechanism allows a function to execute different functionalities (for example, the function `LIFT`), and the higher-order logic of PVS makes the constructions of expressions and predicates easier

with the bottom-up manner. The encoding forms a base to validate TIC reasoning rules and support mechanical verification as follows.

#### 4. CHECKING TIC REASONING RULES

TIC reasoning rules capture the properties of sets of intervals. They are used to verify TIC models at the interval level. Hence guaranteeing their correctness is important and necessary. In this section, we first explain the challenge with an illustration by validating a specific rule. Next, we demonstrate the flaws discovered during the checking, and provide the solutions. In addition, we derive a new rule which simplifies proving process.

Checking TIC reasoning rules is not trivial. Though some of them are automatically proved by the PVS prover, others involve complex proofs which may cover all interval types (for example, analyzing concatenations over three sets of intervals can lead to 16 different cases) and various types of predicates (for example, if a predicate relies on interval operators). Taking the rule given in Section 2.1 as an example, its PVS specification is formed based on the encoding in the previous section, where function `No_Term?` returns true when predicate `p1` contains no free interval operators.

```
CONC_CONC: LEMMA No_Term?(p1) =>
  AllS(p1 AND LIFT(DELTA) > LIFT(0)) =
  concat(AllS(p1), AllS(p1))
```

The checking considers all types of intervals as well as the concatenation over all types of intervals. During the proving process, human interactions are helpful to increase the efficiency. For example, one simplified proof sub-goal shown below needs to instantiate two intervals to form a left-closed, right-open interval  $x!1$ . The *Skolem* constant  $x!1$  is generated by the PVS prover. We can choose the middle point of  $x!1$  as the connecting point of the concatenation by applying proof strategy `assignconcat` to explicitly assign two left-closed, right-open intervals.

```
[-1] PROJ_1(x!1) = CO
[-2] AllS(p1!!1 AND LIFT(DELTA) > LIFT(0))(x!1)
[-3] No_Term?(p1!!1)
|-----
[1] concat(AllS(p1!!1), AllS(p1!!1))(x!1)
Rule? (assignconcat 1
" (CO, (ALPHA(x!1), (ALPHA(x!1) + OMEGA(x!1))/2)) "
" (CO, ((ALPHA(x!1) + OMEGA(x!1))/2, OMEGA(x!1))) ")
```

Note that the above instantiation is validated by the PVS prover for its correctness. In other words, we are required to show that two specified intervals fulfill the constraints of a concatenation. The PVS prover can hence prevent potential mistakes introduced by users such as instantiating incorrect types of intervals.

During the validation of all TIC reasoning rules, two subtle flaws of the original reasoning rules have been discovered. Below we list the problematic rules with counterexamples, followed by the solutions that have been validated.

- Rule *True and False* is frequently used to reason about safety requirements. The original rule states that predicate  $P$  is true everywhere *iff* its negation is true nowhere. Namely,  $\llbracket P \rrbracket = \mathbb{I} \Leftrightarrow \llbracket \neg P \rrbracket = \emptyset$

The implication  $\llbracket \neg P \rrbracket = \emptyset \Rightarrow \llbracket P \rrbracket = \mathbb{I}$  fails in certain circumstances. For example, let timed trace  $x$  have value 1 from time points 5 to 7, and value 0 elsewhere, and predicate  $\neg P$  be  $x = 1 \wedge \delta = 3$ , it is obvious that the predicate fails everywhere, although its negation  $P$ ,  $x \neq 1 \vee \delta \neq 3$ , does not hold in every interval such as the interval  $[5 \dots 8]$ .

To solve the problem, a stronger hypothesis is a must. The predicate within interval brackets should be independent on

interval characteristics, namely, the starting time, ending time and length of an interval. The modified rule is represented in PVS below, where sets `emptyset` and `fullset` denote the empty set and the set of all intervals respectively.

```
Emp_to_All: LEMMA No_Term?(p1) =>
  AllS(not p1) = emptyset => AllS(p1) = fullset
```

- Rule *Concatenation Duration* is useful to deal with proofs involving concatenation. The rule can decompose a set of intervals into two concatenated sets of intervals with specified interval lengths. Namely, given a predicate  $P$  where interval operators do not occur freely, if we have  $r, s : \mathbb{T}$  and  $r > 0 \vee s > 0$ , then,  $\llbracket P \wedge \delta = r + s \rrbracket = \llbracket P \wedge \delta = r \rrbracket \cap \llbracket P \wedge \delta = s \rrbracket$ .

The above equality of two sets of intervals does not always hold. For example, if  $r = 0$ , then any interval of  $\llbracket P \wedge \delta = r \rrbracket$  must be both-closed by the interval definition. However, it is possible that  $\llbracket P \wedge \delta = r + s \rrbracket$  contains intervals which are left-open, and hence type conflict occurs. The conflict is removed by a stronger assumption, namely,  $r > 0 \wedge s > 0$ .

We remark that it is the first time to discover the first flaw (while the second flaw has also been observed by Dawson and Goré [7]). In other words, the discoveries show one benefit of exploiting a theorem prover for rigorous verification.

Moreover, based on the lemma `Emp_to_All`, we derive a new rule `EmpCC_to_All` to reduce the proving complexity. When applying lemma `Emp_to_All`, we have to check the correctness of four basic interval types respectively, and each takes similar proof commands. In contrast, the new rule defined below requires only one type of intervals, namely, both-closed intervals to be checked.

```
EmpCC_to_All: LEMMA No_Term?(p1) =>
  CCS(not p1) = emptyset => AllS(p1) = fullset
```

We have checked all TIC reasoning rules in PVS. Two flaws have been found out and fixed. These rules can thus be applied as lemmas in PVS to verify TIC models at the interval level.

#### 5. TRANSFORMING TIC MODELS TO PVS

Based on the encoding of TIC semantics in PVS (refer to Section 3), we present in this section an algorithm to automatically transform TIC models into PVS specifications. The transformed PVS specifications follow closely to original TIC models so the diagnostic information obtained at the level of PVS can be reflected back to the level of TIC. The algorithm has been implemented in Java. We adopt an adaptive temperature control system [16, 21] as a running example to demonstrate the transformation here as well as the verification in the next section.

##### 5.1 Representing TIC Models in PVS

Using TIC, system properties are modeled by TIC schemas, and requirements are specified as TIC predicates. In the following, we explain the way of representing them in PVS respectively.

Schemas are used to structure and compose models; collating pieces of information, encapsulating them and naming them for reuse. A TIC schema consists of two parts, where variables are defined with their types in the declaration part, and their values are constrained in the predicate part. Each schema denotes a composite type which is made up of a set of bindings, and each binding relates a declared variable with its restrictive values. This modeling feature allows schemas to be used as types so as to support component-based design [22].

Each schema is represented by a set of PVS *records*. Each record stores the declared variables by *record accessors* with their corresponding types in PVS, and expresses the constraints over the record accessors as set predicates. Additional constraints are generated to capture implicit properties of certain kinds of functions of TIC, in particular continuity, differentiability and integrability.

A TIC predicate specifying a requirement of a system or some components is formed based on the TIC schemas which correspond to the system or components. Each requirement is represented by a PVS *theorem* formula, and its contents are specified by the PVS specifications accordingly.

Informally, the above relationships between TIC models and PVS specifications can be illustrated below, where variable `temp` used in the construction of records is auxiliary to access record accessors in the generated `Predicate` specification.

<i>sch_name</i>	SchName : TYPE = {temp:
<i>declaration</i>	[# Declaration #]
<i>predicate</i>	Predicate }
<i>req_name</i> ==	ReqName : THEOREM
<i>predicate</i>	Predicate

## 5.2 Transformation Algorithm

The transformation algorithm consists of three main steps, scanning, parsing, and transforming<sup>2</sup>. A scanner splits TIC models into a sequence of meaningful tokens such as TIC interval brackets, mathematical operators and so on. A parser constructs an abstract syntax tree (AST) for TIC models based on the tokens from the scanner. The root of an AST is a list of TIC models, and each element represents a TIC schema or a TIC predicate (but not both). From an element, we can find out the relevant information stored in its child nodes, such as the name, declarations and predicate of a schema, or the name and predicate of a requirement. From these child nodes, we can explore deeper to obtain more information, for example expressions. The leaves of an AST are the primitive elements of TIC, namely, constants, timed traces or interval operators. We develop an algorithm below to generate the corresponding PVS specifications by traversing an AST in the top-down manner.

### Algorithm 1 Transforming TIC models

**Require:** A list of TIC models *ModList*

```

1: for i = 1 to ModList.length do
2:   if ModList[i] instanceof TICSchema then
3:     anaSchName(ModList[i].Name)
4:     DeclList ← ModList[i].Declarations
5:     for j = 1 to DeclList.length do
6:       anaDeclName(DeclList[j].Name)
7:       InSchema ← true
8:       LIFTED ← false
9:       anaExpr(DeclList[j].Expr, InSchema, LIFTED)
10:    end for
11:    anaPredicate(ModList[i].Predicate)
12:    createNewPred(DeclList)
13:  else
14:    anaReqName(ModList[i].Name)
15:    anaPredicate(ModList[i].Predicate)
16:  end if
17: end for

```

In the above algorithm, method *anaExpr* at line 8 analyzes 14 types of expressions, and method *anaPredicate* at lines 10 and 14 supports 8 types of predicates. These are sufficient to handle the TIC models in practice. Note that the method *anaExpr* is applied

<sup>2</sup>We emphasize the transforming phase here, and the tool is available from the website [4].

in the method *anaPredicate*. Two flags, *InSchema* and *LIFTED*, as the parameters of the method *anaExpr* indicate the environment of an expression, and hence assist proper transformation. For example, when analyzing a variable expression, if *LIFTED* is true, that means the variable is within a pair of interval brackets, then we apply the function `LIFT` to make the references of time and intervals to the variable explicit. Method *createNewPre* at line 12 generates additional constraints which are implicitly denoted in TIC models. For example given a continuous timed trace  $v$  (indicated by symbol  $\Rightarrow$  in TIC), we adopt the function *continuous* from the NASA PVS library to capture the continuity feature, namely, *continuous*( $v$ ).

## 5.3 Temperature Control System

A temperature control system [16, 21] is a hybrid application to control the temperature by turning a heater on or off. We adapt it to fit for the demonstration in this paper. The modified system involves discrete logic and continuous dynamics, and important requirements including safety requirements and timing requirements.

Two components compose the system. A *plant* represents the physical environment where the temperature changes *continuously* following different integration functions based on the heater status. A *controller* turns on or off the heater according to the temperature from the plant. Initially, the heater is *off* and the temperature is 30. The TIC models of the system are available in Appendix A. We use the controller here to demonstrate the transformation.

The properties of the controller are modeled below. For any interval in which the temperature *tmp\_in* is not larger than the minimum value 20, the heater *heater\_out* must be on (denoted by value 1). Similarly for interval during which *tmp\_in* is not smaller than the maximum value 40, the heater is off.

<i>Controller</i>
$tmp\_in : \mathbb{T} \Rightarrow \mathbb{R}; heater\_out : \mathbb{T} \rightarrow \{0, 1\}$
$\llbracket tmp\_in \leq 20 \rrbracket \subseteq \llbracket heater\_out = 1 \rrbracket$
$\llbracket tmp\_in \geq 40 \rrbracket \subseteq \llbracket heater\_out = 0 \rrbracket$

In the following transformed PVS specification, the schema is represented by a PVS record type. The declaration part is captured by the record accessors with their corresponding types in PVS. The predicate part is mapped into a conjunction of three constraints. Note that the last constraint which is generated by the method *createNewPred* in algorithm 1, models the continuous characteristic of the timed trace *tmp\_in*.

```

Controller: TYPE = {
  temp: [# tmp_in: Trace, heater_out: BTrace #] |
  subset?(AllS(LIFT(temp'tmp_in) <= LIFT(20)),
    AllS(LIFT(temp'heater_out) = LIFT(1))) AND
  subset?(AllS(LIFT(temp'tmp_in) >= LIFT(40)),
    AllS(LIFT(temp'heater_out) = LIFT(0))) AND
  continuous(temp'tmp_in)
}

```

One safety requirement of the system is that the temperature is always within the valid range from 20 to 40.

$$\begin{aligned}
\text{Safety} &== \forall s : \text{System} \bullet \\
&\mathbb{I} = \llbracket s.pla.tmp\_out \leq 40 \wedge s.pla.tmp\_out \geq 20 \rrbracket
\end{aligned}$$

```

Safety: THEOREM forall (s: System): fullset =
  AllS(LIFT(s'pla'tmp_out) <= LIFT(40) and
    LIFT(s'pla'tmp_out) >= LIFT(20));

```

Using TIC, we can specify important timing requirements. A requirement *Length* describes that for any interval which starts from time point 0, the accumulation of the lengths of the intervals in which the heater is on is always less than three-fourths of the length of the interval. We show below its TIC model and the transformed

PVS specifications respectively. We remark that the requirement is not supported in the original papers [16, 21] which lack the support for modeling continuous behavior.

$$\text{Length} == \forall s : \text{System} \bullet \\ \llbracket \alpha = 0 \rrbracket \subseteq \llbracket \int_{\alpha}^{\omega} s.\text{con.heater\_out} \leq \frac{3}{4} * \delta \rrbracket$$

```
Length: THEOREM forall (s: System): subset?(
  Alls(LIFT(ALPHA) = LIFT(0)),
  Alls(TICIntegral(LIFT(ALPHA), LIFT(OMEGA),
    s`con`heater_out) <= LIFT(3/4) * LIFT(DELTA)));
```

## 6. VERIFYING TIC MODELS IN PVS

Verification of TIC models is nontrivial, as usually systems contain continuous dynamics and requirements involve infinite intervals. When verifying TIC models in PVS, certain grade of automation is desired. In this section, we first define a set of supplementary rules and PVS proof strategies to ease the reasoning process. Next we present a general proving procedure to systematically analyze TIC models in PVS. The procedure application is then illustrated through our example studies. In the end we discuss how to enhance our system to support other notations of real-time systems.

### 6.1 Supplementary Rules and Proof Strategies

The TIC reasoning rules validated in Section 4 capture primitive properties of sets of intervals. However, they are inadequate to support special characteristics of specific domains such as continuous functions. For example, if a continuous timed trace  $tr$  crosses a threshold  $TH$  at an interval  $i$  in a way  $tr(\alpha(i)) < TH \wedge tr(\omega(i)) > TH$ , then we can infer that the interval can be decomposed into *three* connected intervals, where the values of the trace are larger than the threshold everywhere in the *last* subinterval and equal to the threshold in the *middle* subinterval. We declare below a PVS lemma `mid_ivl_exi`, where  $\circ$  is the function composition operator, to express the special feature of continuous timed traces.

```
mid_ivl_exi: LEMMA continuous(tr) => subset?(
  Alls((LIFT(tr) o LIFT(ALPHA)) < LIFT(TH) and
    (LIFT(tr) o LIFT(OMEGA)) > LIFT(TH)),
  concat(Alls(TRUE), concat(Alls(LIFT(tr) = LIFT(TH)),
    Alls(LIFT(tr) > LIFT(TH))));
```

The above feature has not been captured by any existent TIC reasoning rule. It is derived from the classic *intermediate value* theorem of continuous functions. We have reasoned about its correctness in PVS, and hence we can apply the lemma when analyzing continuous dynamics.

To make proving process more automated, and ease users from grasping detailed TIC encoding in PVS, we have developed several PVS proof strategies. Each strategy combines repetitive proof commands which are used frequently in practice. The strategies mainly cope with quantified PVS formulas during the reasoning process, as the PVS prover possesses powerful capabilities (such as automatic deduction, simplification) on reasoning about primitive formulas which are represented in the propositional logic. Based on the quantifier type, the strategies are classified into two categories. One eliminates the *universal* quantifier by *skolemization*, and the other removes the *existential* quantifier by proper instantiation. In addition, the strategies usually automatically expand definitions of TIC semantics in PVS, and hence we can keep the detailed encoding of TIC transparent to users. Below we illustrate the strategy `AssignInterval` which offers a flexible way to instantiate specific interval and time point to a user-specified formula.

```
1: (defstep AssignInterval (fnum &OPTIONAL ivl pt)
2:   (try (else (expand "OOS" fnum)
3:         (else (expand "OCS" fnum)
4:               (else (expand "COS" fnum)
5:                     (else (expand "CCS" fnum)
6:                           (else (expand "Alls" fnum)
7:                                 (skip))))))
8:   (then (if ivl (inst fnum ivl)
9:         (inst? fnum))
10:  (expand "Everywhere?" fnum)
11:  (if pt (inst fnum pt) (inst? fnum)))
12:  (skip)))
```

In the above strategy, users can provide explicit values of the interval and the time point denoted by the parameters `ivl` and `pt` respectively (following the keyword `&OPTIONAL` at line 1), or let the PVS prover automatically fix an interval and a time point by applying the PVS proof command `inst?` (at lines 9 and 11). The strategy is able to handle different types of intervals by repeatedly using a basic PVS strategy `else` (which applies the first step, and if that does nothing, then the second step is applied) to expand appropriate definition of the interval brackets in the specified formula. Note that the strategy automatically expands the `Everywhere?` function (at line 10). In other words, it keeps the definition which encodes TIC semantics transparent to users.

We have constructed 25 supplementary rules used frequently in practice, and 11 PVS strategies. They facilitate the verification of TIC models by elevating the grade of automation.

### 6.2 General Proving Procedure

In general, verification in TIC is to show logical implication relationships among TIC models, namely, to check whether TIC schemas that represent system properties imply TIC predicates that denote requirements. The proving procedure is a deduction which starts from hypotheses (for example, TIC schemas) and processes in a forward manner. Each deductive step is reached by applying a TIC reasoning rule or a mathematical law to a hypothesis. Usually a TIC proof can be decomposed into several sub-proofs of subsystems to simplify the proving.

In our system, we can verify the PVS specifications that represent TIC models in a close manner with a high degree of automation. During a proving process, a proof goal is a sequent initially expressed in terms of intervals. The objective is to concretize the sequent by assigning proper values to intervals and time points to eliminate quantified formulas. Hence the PVS prover can directly manipulate the sequent and automatically discharge most of tedious proof steps such as reasoning about arithmetic and sets. We sketch a proving procedure below for a proof goal.

1. Introducing new antecedents that represent system properties. As only system names are referenced in the PVS specifications that depict requirements in the beginning, system properties can be introduced to the sequent as new antecedents by applying the PVS proof command `typepred`.
2. Applying TIC reasoning rules and supplementary rules. It can change the sequent in two directions. (1) *Backward proof*: generate new consequents provided some of the current consequents match the conclusion of a rule. (2) *Forward proof*: create new antecedents if some of the current antecedents satisfy the premise of a rule.
3. Using proved lemmas. These lemmas are often sub-goals which capture simpler requirements over subsystems, and they can be used to compose complex proofs.
4. Instantiating intervals and time points. This step makes the intervals and time explicit. They can be assigned by users or automatically by the PVS prover.

5. Adopting mathematical laws. As TIC models often contain continuous dynamics represented by integral and differential calculus, we can select special lemmas from the NASA PVS library to support the analysis.
6. Applying PVS proof command `grind`. It is the last step to automatically discharge the proof goal.

In the above procedure, the first three steps manipulate a proof sequent at the interval level, and the last step eases the workload on checking low level proofs by exploiting the PVS powerful reasoning capability. The proposed procedure can result in an efficient and effective way to reason about TIC models in our verification system. In the following section we demonstrate how it facilitates the analysis of our example studies.

### 6.3 Evaluation with Example Studies

We have experienced the proving procedure with two applications. One is the temperature control system mentioned in Section 5.3. The other is a brake control system which aims to prevent vehicle from dangerous speed by turning on brakes in time. It is necessary for both systems to satisfy important requirements such as safety requirements and real-time constraints.

Due to the page limit, we just show how to check two requirements stated in Section 5.3. The first checking focuses on illustrating the advantages of the general proof procedure. The second checking emphasizes on demonstrating the capability of handling induction proofs in PVS. At the end of this section, we summarize the experimental results of both example studies. The detailed specifications of the proved lemmas with their complete proofs can be found online [4].

#### Proof of Safety Requirement

Instead of checking the temperature in *every* interval, we adopt the *proof by contradiction* mechanism to show that there is no interval during which the temperature is outside the valid range. Moreover, by the reasoning rule, `EmpCC_to_All` defined in Section 4, we just need to prove that no such *both-closed* interval exists.

The proof is divided into two sub-proofs which check whether the temperature is greater (or lower) than the maximum (or minimum). Each sub-proof relies on a lemma which depicts the continuous behavior of the temperature respectively. We illustrate below how a lemma *Decreasing* is verified by using the above proving procedure. The lemma denotes that for any both-closed interval in which the temperature is not lower than 40 the temperature at the ending point is not greater than that at the starting point.

$$\begin{aligned} \text{Decreasing} &== \forall s : \text{System} \bullet \\ &\{s.pla.tmp\_out \geq 40\} \subseteq \\ &\{s.pla.tmp\_out(\omega) \leq s.pla.tmp\_out(\alpha)\} \end{aligned}$$

The verification of the proof commands used is shown below.

```

1: ((skosimp)
2: (expandsubset)
3: (typepred "s!1") (typepred "s!1`con")
4: (assignsubset -1)
5: (("1" (typepred "s!1`pla")
6: (assignsubset -1)
7: ("1" (lemma "Integral_ge_0")
8: (inst - "ALPHA(x!1)" "OMEGA(x!1)"
"s!1`pla`tmp_out")
9: (ground)
10: ("1" (grind)) ("2" (grind))
11: ("3" (use "cont_Integrable?") (grind))
12: ("4" (skosimp) (grind)))
13: ("2" (rewrite "Invariant_True_R")
14: (expintervaltotime 1) (grind)))
15: ("2" (rewrite "Invariant_True_R") (grind)))

```

- Properties (for example, the connections between components) of the whole system `s!1` and its subsystems, `s!1`con` and `s!1`pla`, are added as new antecedents by applying the proof command `typepred` to corresponding names (at lines 3 and 5).
- The proof strategies developed by us are used to manipulate the proof sequent at the interval level (at lines 2, 4, 6, 14). For example, the application `(assignsubset -1)` at line 4 directs the PVS prover to automatically instantiate an interval to the first antecedent formula.
- As the temperature changes continuously, we need to apply special lemmas from the NASA PVS library (at lines 7 and 11) to cope with mathematical analysis. For instance, lemma `Integral_ge_0` at line 7 represents a property that if an integrable function has nonnegative values over a closed interval its integral over the interval is nonnegative.
- In each branch, the proof command `grind` completes the proof (at lines 10, 11, 12, 14 and 15).

Note that during the verification, all instantiations of intervals and time points are automatically accomplished. The only one manual instantiation (at line 8) requires the values of the bounds of an interval and the integral function. Users help the PVS prover to increase the reasoning efficiency by assigning appropriate values rather than letting the prover try all possible instantiations. Nevertheless the prover still checks the validity of the values.

#### Proof of Length Requirement

Based on the assumption of *finite variability*, we apply the *mathematical induction* mechanism to show that the requirement holds in any arbitrary interval. The finite variability restricts [24] that within any bounded interval a discrete-valued state can change only finitely many times. In other words, we assume that for a discrete timed trace, an interval is classified into two groups: (1) the discrete timed trace is constant over the whole interval; or (2) the interval can be decomposed into a sequence of sub-intervals, and the discrete timed trace has different values in any adjacent sub-intervals. The property has been formalized in PVS and is helpful to deal with the analysis of intervals by induction.

To prove the requirement, we decompose any interval of  $\llbracket \alpha = 0 \rrbracket$  into a sequence of subintervals, of which the heater is *off* in the first subinterval (by the system initiation) and can be *off* or *on* in the last subinterval. As  $\int \text{heater\_out} = 0$  when the heater is *off*, we can hence concern the analysis over special intervals in which the starting time point is 0 and the heater is *on* in the end. These intervals can be constructed by the following recursive function *superCon*. The function receives three parameters, a counter  $k$ , a set of interval as a *base*, and a set of interval as a *unit*. It returns a set of intervals by repeatedly appending the unit for  $k$  times to the base.

$$\begin{array}{l} \text{superCon} : \mathbb{N} \times \mathbb{P}\mathbb{I} \times \mathbb{P}\mathbb{I} \rightarrow \mathbb{P}\mathbb{I} \\ \hline \forall k : \mathbb{N}; \text{base}, \text{unit} : \mathbb{P}\mathbb{I} \bullet \text{superCon}(k, \text{base}, \text{unit}) = \\ \quad \text{if } k = 0 \text{ then } \text{base} \\ \quad \text{else } \text{superCon}(k - 1, \text{base}, \text{unit}) \curvearrow \text{unit} \end{array}$$

Hence we can represent the special intervals below by the application  $\text{superCon}(k, \text{base}, \text{unit})$ , where *base* denotes the first time when heater turns on from its initial state *off*, and *unit* expresses a sequential behavior of the heater, specifically, from *off* to *on*.

$$\begin{aligned} \text{base} &= \llbracket \alpha = 0 \wedge s.con.heater\_out = 0 \rrbracket \curvearrow \\ &\llbracket s.con.heater\_out = 1 \rrbracket; \\ \text{unit} &= \llbracket s.con.heater\_out = 0 \rrbracket \curvearrow \llbracket s.con.heater\_out = 1 \rrbracket \end{aligned}$$

In the following we demonstrate how to handle an inductive proof in our system by proving a simple lemma *end\_with\_On*, which expresses that at the end of the special intervals the heater is *on*.

$$\begin{aligned} \text{end\_with\_On} &== \forall s : \text{System} \bullet \forall k : \mathbb{N} \bullet \\ &\text{superCon}(k, \text{base}, \text{unit}) \sqsubseteq \\ &[\text{true}] \curvearrowright [\text{s.con.heater\_out} = 1] \end{aligned}$$

The verification of the above lemma is below.

```

1: ((skosimp)
2: (induct "k")
3: (("1" (expandsubset)
4: (expand "superCon")
5: (expandconcat -1)
6: (assignconcat 1 "i1!1" "i2!1") (grind))
7: ("2" (skosimp)
8: (expandsubset)
9: (expand "superCon")
10: (expandconcat -2 1) (expandconcat -1)
11: (assignconcat 1 "TwoTOneIv1(i1!1, i1!2)" "i2!2")
12: (grind)))

```

In the above proof since the function *superCon* is recursive, we need to perform the induction on the variable *k*. The PVS command *induct* invokes an inductive proof. Its application at line 2 automatically generates two sub-goals. One denotes the base case where  $k = 0$  (at line 3), and the other corresponds to the inductive case (at line 7). Function *TwoTOneIv1* defined by us creates a new interval from two adjacent intervals (at line 11). We remark that the analysis of the inductive case considers two connections of three sets of intervals (indicated by expanding function *concat* twice at line 10). If the verification is manual, we have to check 16 circumstances in terms of the interval types. In contrast, the tedious work is automatically completed in our system.

### Experimental Results

Table 1 presents the lemmas of the temperature control system, the number of the proof commands entered from users, and the PVS system execution time (in seconds) for each lemma. We have mentioned some lemmas, namely, *Decreasing*, *Safety*, *end\_with\_On*, and *Length*. Other lemmas also represent important properties. For instances, lemma *Off\_On\_Off* reasons about the *interval length* when the heater is on between any two consecutive *off* states, and its verification involves the mathematical analysis of integral calculus (for example, the calculation of the temperature); lemma *invariant* proves that the special intervals represented by *superCon(k, base, unit)* fulfills the *length* requirement by induction.

**Table 1: Verification of the temperature control system**

Lemma Name	Steps	Time	Lemma Name	Steps	Time
Decreasing	21	14.45	Off_On_Off	44	20.92
Increasing	24	16.66	end_with_On	14	21.88
Safe1	28	14.15	invariant	79	458.8
Safe2	27	14.68	super_and_BT	23	10.59
Safety	7	12.88	Length_Cover	98	562.32
On_Off_On	46	22.47	Length	26	59.67

Besides the experiment on the temperature control system presented, we also apply our verification system to another hybrid application, a brake control system. The system involves discrete logic (a controller controlling the brake), periodic behavior (a sensor sampling the real speed every 1 time unit) and continuous-time dynamics (the vehicle speed is an integration of its acceleration).

We have successfully proved three important requirements by using our verification system. Requirement *Approximation* checks that at any time the sensor measures the speed within an accuracy range. Requirement *Response* guarantees that the system will react

on time whenever the vehicle is over dangerous speeding. Requirement *Safety* shows that a vehicle will be always within a safe speed. Table 2 summarizes the proved lemmas, the number of proof commands needed, and the execution time (in seconds).

**Table 2: Verification of the brake system**

Lemma Name	Steps	Time	Lemma Name	Steps	Time
ACC_Range	11	77.73	Response_L2	68	26.26
Plant_Speed	43	73.98	Response_L3	49	26.40
Approximation	26	13.78	Response_L4	53	28.70
V_Initial	8	10.46	Response	18	15.74
Brake_Prop	27	45.75	ACC_On	16	66.66
V_at_sample	22	23.26	overlimit1	42	112.20
V_within_sample	27	21.56	overlimit2	55	185.88
Response_L1	46	24.83	Safety	48	103.45

We remark that in the proof of the *Response* requirement, we adopt the *proof by cases* mechanism. Specifically any arbitrary interval can be classified into one of the four groups by the criteria that whether its endpoints are sample time points, and each group is analyzed as a lemma (for example, lemma *Response\_L1*).

## 6.4 Supporting Other Notations

By the highly expressive power of TIC, we can encode other notations of real-time systems in TIC and hence support their machine-assisted proof in our tool. Below we show an application of our verification system to a specific notation, Duration Calculus (DC) [24].

DC is a popular formal notation, and builds on the idea of measuring the time points where a state assertion holds within an interval. Note that the encoding of time intervals and integral operation in TIC provides a theoretic base to construct the DC semantics. Here we briefly discuss how important features of DC can be denoted by TIC. Detailed transformation from DC to TIC is available online [4], where a case study of a gas burner is given as well.

Primitive state variables of DC are functions from time to a set  $\{0, 1\}$ . They are exactly the discrete timed traces of TIC. Hence, the frequently used abbreviation in DC,  $[S]$ , which holds in a non-pointer interval if state assertion  $S$  is true *almost* everywhere, can be represented by a TIC expression  $[\int_{\alpha}^{\omega} S = \delta \wedge \delta > 0]$ . In DC, predicate values at endpoints are irrelevant. That is to say, they can be either true or false at the endpoints of intervals. This feature is captured by the interval brackets  $[$ ] of TIC.

The chop operator  $\frown$  of DC can be converted to the concatenation operator  $\curvearrowright$  of TIC. Note that the property where two connected intervals share an overlapping time point is not allowed in TIC. The difference can be addressed by concatenating two general types of sets of intervals. This way covers all possible cases about the connections of any two intervals, captures the DC feature which ignores the predicate values at the interval endpoints. Hence, a DC model can be treated in TIC as a special case that a TIC model involves all types of intervals.

## 7. CONCLUSION

In conclusion, we presented a verification system for TIC based on the generic theorem prover PVS. TIC is highly expressive to support modeling of complex systems and (timing) requirements. The verification is usually nontrivial due to the analysis of continuous dynamics as well as the reasoning over arbitrary (infinite) time intervals. We have constructed the TIC semantics models in PVS so as to fully support the TIC modeling features. In the validation of all TIC reasoning rules, we have discovered two subtle flaws in two frequently used original rules. After fixing the flaws we further refined one of them to reduce the proof complexity to one quarter. A translator has been implemented in Java to automatically transform TIC models to PVS specifications.

The resultant system supports the verification of TIC to be carried out directly at the interval level by applying the reasoning rules and supplementary rules which capture domain specific features. We have developed 11 PVS proof strategies to facilitate the usage of our system. In addition, these strategies ease the users from understanding detailed encoding of TIC in PVS. Low level proof goals are discharged automatically by the PVS powerful prover, especially the decision procedures on sets and arithmetic on real numbers. We have successfully applied our approach to validate hybrid control systems with timing related safety requirements. The support of mathematical induction in PVS is useful to deal with infinite systems. We showed that our approach could be modified to support other real-time system notations, for instance DC.

Currently our approach is limited to fully automated verify TIC models in general. This is the price to be paid by the highly expressive power of TIC. The main challenge is to instantiate appropriate values (for example, intervals or time points) to eliminate quantified formulas in PVS. From the experiments, we find that there are heuristics which can elevate the automation grade (for example, when assigning a time point in an interval, usually the proof can be completed successfully in one of three following ways of the assignments: two endpoints or the middle point of the interval). We are in the process of developing more intelligent proof strategies to implement these heuristics so as to support the mechanized proving of TIC at a higher level. Moreover, embedding our approach into SAL [8] is possible to improve the verification, in particular the infinite model checking capability of SAL providing a way to handle infinite systems.

## 8. ACKNOWLEDGEMENTS

This research work is supported by MOE Tier2 research project ‘‘Rigorous Design Methods and Tools for Intelligent Autonomous Multi-Agent Systems (R-252-000-201-112)’’. We thank Yuzhang Feng and Ian Hayes for their insightful discussion. We appreciate Anders. P. Ravn, Chaochen Zhou and Jeremy Dawson for the help on related work.

## 9. REFERENCES

- [1] R. W. Butler. Formalization of the Integral Calculus in the PVS Theorem Prover. Technical report, NASA Langley Research Center, Hampton, Virginia, October 2004.
- [2] A. Cerone. Axiomatisation of an Interval Calculus for Theorem Proving. *Electronic Notes Theoretical Computer Science*, 42, 2001.
- [3] G. Chakravorty and P. K. Pandya. Digitizing Interval Duration Logic. In *Computer Aided Verification*, pages 167–179, 2003.
- [4] C. Chen, J. S. Dong, and J. Sun. Verification System for TIC. <http://www.comp.nus.edu.sg/~chenchun/TIC2PVS>, 2007.
- [5] B. H. Cheng and J. M. Atlee. Research Directions in Requirements Engineering. *Future of Software Engineering*, pages 285–303, 2007.
- [6] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
- [7] J. E. Dawson and R. Goré. Machine-checking the Timed Interval Calculus. In *Australian Joint Conference on Artificial Intelligence*, pages 95–106, 2002.
- [8] B. Dutertre and M. Sorea. Timed Systems in SAL. Technical report, SRI International, 2004.
- [9] C. J. Fidge, I. J. Hayes, and B. P. Mahony. Defining Differentiation and Integration in Z. In *Formal Methods and Software Engineering*, pages 64–73. 1998.
- [10] C. J. Fidge, I. J. Hayes, A. P. Martin, and A. Wabenhörst. A Set-Theoretic Model for Real-Time Specification and Reasoning. In *Mathematics of Program Construction*, pages 188–206, 1998.
- [11] S. T. Heilmann. *Proof Support for Duration Calculus*. PhD thesis, Department of Information Technology, Technical University of Denmark, 1999.

- [12] T. A. Henzinger and J. Sifakis. The Embedded Systems Design Challenge. In *Formal Methods*, pages 1–15, 2006.
- [13] B. P. Mahony and I. J. Hayes. A Case-study in Timed Refinement: A Mine Pump. *IEEE Transactions on Software Engineering*, 18(9):817–826, 1992.
- [14] A. K. Mok, C.-G. Lee, H. Woo, and P. Konana. The Monitoring of Timing Constraints on Time Intervals. In *Real-Time Systems Symposium*, page 191–200. 2002.
- [15] C. Muñoz, V. Carreño, and G. Dowek. Formal Analysis of the Operational Concept for the Small Aircraft Transportation System. In *Rigorous Development of Complex Fault-Tolerant Systems*, pages 306–325, 2006.
- [16] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to Timed Graphs and Hybrid Systems. In *Real-Time: Theory in Practice*, pages 549–572. Springer-Verlag, 1992.
- [17] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.
- [18] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A Prototype Verification system. In *Automated Deduction*, pages 748–752, 1992.
- [19] J. M. Rushby. Theorem proving for verification. In *Modeling and Verification of Parallel Processes*, pages 39–57. Springer, 2000.
- [20] J. U. Skakkebak and N. Shankar. Towards a Duration Calculus Proof Assistant in PVS. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 660–679. 1994.
- [21] A. Tiwari, N. Shankar, and J. M. Rushby. Invisible Formal Methods for Embedded Control Systems. *Proceedings of the IEEE*, 91(1):29–39, 2003.
- [22] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof*. Prentice Hall International, 1996.
- [23] Y. Yu, S. Ren, and O. Frieder. Prediction of Timing Constraint Violation for Real-time Embedded Systems with known Transient Hardware Failure Distribution Model. In *Real-Time Systems Symposium*, pages 454–466. 2006.
- [24] C. C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. Springer Verlag, 2004.
- [25] C. C. Zhou and X. S. Li. A mean value calculus of durations. In *A practical mind: essays in honour of C. A. R. Hoare*, pages 431–451. Prentice-Hall International, 1994.
- [26] C. C. Zhou, A. P. Ravn, and M. R. Hansen. An Extended Duration Calculus for Hybrid Real-time Systems. In *Hybrid Systems*, pages 36–59. Springer-Verlag, 1993.

## APPENDIX

### A. THE CONTROL SYSTEM IN TIC

The TIC models of *plant*, *controller* and the whole system.

<div style="border-bottom: 1px solid black; margin-bottom: 5px;"> <p style="text-align: center; margin: 0;"><i>Plant</i></p> <p style="margin: 0;"><math>tmp\_out : \mathbb{T} \Rightarrow \mathbb{R}; heater\_in : \mathbb{T} \rightarrow \{0, 1\}</math></p> <p style="margin: 0;"><math>\llbracket heater\_in = 0 \rrbracket \subseteq</math></p> <p style="margin: 0;"><math>\llbracket tmp\_out(\omega) = tmp\_out(\alpha) - (1/10) * \int_{\alpha}^{\omega} tmp\_out \rrbracket</math></p> <p style="margin: 0;"><math>\llbracket heater\_in = 1 \rrbracket \subseteq</math></p> <p style="margin: 0;"><math>\llbracket tmp\_out(\omega) = tmp\_out(\alpha) + 6 * \delta - (1/10) * \int_{\alpha}^{\omega} tmp\_out \rrbracket</math></p> </div>
<div style="border-bottom: 1px solid black; margin-bottom: 5px;"> <p style="text-align: center; margin: 0;"><i>Controller</i></p> <p style="margin: 0;"><math>tmp\_in : \mathbb{T} \Rightarrow \mathbb{R}; heater\_out : \mathbb{T} \rightarrow \{0, 1\}</math></p> <p style="margin: 0;"><math>\llbracket tmp\_in \leq 20 \rrbracket \subseteq \llbracket heater\_out = 1 \rrbracket</math></p> <p style="margin: 0;"><math>\llbracket tmp\_in \geq 40 \rrbracket \subseteq \llbracket heater\_out = 0 \rrbracket</math></p> </div>
<div style="border-bottom: 1px solid black; margin-bottom: 5px;"> <p style="text-align: center; margin: 0;"><i>System</i></p> <p style="margin: 0;"><math>con : Controller; pla : Plant</math></p> <p style="margin: 0;"><math>\mathbb{I} = \llbracket con.tmp\_in = pla.tmp\_out \rrbracket \wedge \mathbb{I} = \llbracket pla.heater\_in = con.heater\_out \rrbracket</math></p> <p style="margin: 0;"><math>\llbracket \alpha = 0 \rrbracket \subseteq \llbracket pla.tmp\_out(\alpha) = 30 \wedge con.heater\_out(\alpha) = 0 \rrbracket</math></p> </div>