4-2010

A verification system for interval-based specification languages

Chunqing CHEN

Jin Song DONG

Jun SUN

Andrew P. MARTIN

A Verification System for Interval-Based Specification Languages

CHUNQING CHEN, JIN SONG DONG, and JUN SUN
National University of Singapore
and
ANDREW MARTIN
University of Oxford

Interval-based specification languages have been used to formally model and rigorously reason about real-time computing systems. This usually involves logical reasoning and mathematical computation with respect to continuous or discrete time. When these systems are complex, analyzing their models by hand becomes error-prone and difficult. In this article, we develop a verification system to facilitate the formal analysis of interval-based specification languages with machine-assisted proof support. The verification system is developed using a generic theorem prover, Prototype Verification System (PVS). Our system elaborately encodes a highly expressive set-based notation, Timed Interval Calculus (TIC), and can rigorously carry out the verification of TIC models at an interval level. We validated all TIC reasoning rules and discovered subtle flaws in the original rules. We also apply TIC to model Duration Calculus (DC), which is a popular interval-based specification language, and thus expand the capacity of the verification system. We can check the correctness of DC axioms, and execute DC proofs in a manner similar to the corresponding pencil-and-paper DC arguments.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods; validation*

General Terms: Verification

Additional Key Words and Phrases: Formal specification languages, real-time systems, theorem proving

This article is a revised and extended version of a paper presented at the 30th International Conference on Software Engineering (ICSE'08) [Chen et al. 2008].

This work has been supported in part by ARC Approved Projects under the project "Rigorous Design Methods and Tools for Intelligent Autonomous Multi-Agent Systems".

Authors' addresses: C. Chen, Department of Computer Science, National University of Singapore, Computing 1, 13 Computing Drive, Singapore 117414, Republic of Singapore; email: chenchun@comp. nus.edu.sg.

DOI 10.1145/1734229.1734232 http://doi.acm.org/10.1145/1734229.1734232

ACM Reference Format:
Chen, C., Dong, J. S., Sun, J., and Martin, A. 2010. A verification system for interval-based specification languages. ACM Trans. Softw. Engin. Method. 19, 4, Article 13 (April 2010), 36 pages.
DOI = 10.1145/1734229.1734232 http://doi.acm.org/10.1145/1734229.1734232

1. INTRODUCTION

Real-time computing systems usually interact with the physical environment, and they often involve mathematical functions of time. With their increasing usage in safety-critical situations, it is necessary and important to rigorously validate the design of these systems associated with properties of the environment against requirements at an early stage [Cheng and Atlee 2007]. Consequently, it is desirable for their formal models to capture various behaviors, such as those described by discrete logics of computerized controllers and continuous dynamics of the environment [Henzinger and Sifakis 2006]. Moreover, it is crucial for the modeling language to possess powerful verification capabilities to verify whether the models satisfy requirements.

Formal models of real-time systems can be divided into two broad groups [Alur and Henzinger 1991]: those based on time points and those based on time intervals. Point-based specification languages express system behavior over time points, and they are convenient for describing event occurrences. Interval-based specification languages are typically used to express behavior over a period of time points, for instance, using integration. The latter can be regarded as more appropriate and concise than the former since constraints on intervals frequently occur in real-time systems [Mattolini and Nesi 2001], especially in the control engineering domain.

Two prominent interval-based specification languages are Timed Interval Calculus (TIC) [Fidge et al. 1998b] and Duration Calculus (DC) [Zhou and Hansen 2004]. Although both languages offer similar operators and capabilities, their bases are different. TIC is based on set-theory and reuses Z [Woodcock and Davies 1996] mathematical and schema notations. TIC models system behavior by constraining intervals during which enclosed predicates hold everywhere. DC is based on interval temporal logic [Moszkowski 1986], and it represents behavior by constraining state durations by accumulating the Boolean-valued states over closed intervals. Furthermore, TIC supports explicit references to interval endpoints, which can specify properties over special intervals with particular endpoints.

When real-time computing systems are complex, 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 thus necessary and important to develop a verification system to make proofs easier. Nevertheless, the analysis of these systems usually involves mathematical reasoning and induction mechanisms for dealing with arbitrary infinite intervals and continuous time domain. These characteristics are not well supported by model checking [Clarke et al. 1994], which usually applies a discrete abstraction for infinite state spaces. The abstraction could decrease the accuracy of analysis in continuous dynamics [Muñoz et al.

2003]. In contrast, theorem proving [Rushby 2000] can handle infinite state spaces directly and support expressive specifications.

Instead of building a theorem prover from scratch, we choose one of the powerful generic theorem provers, Prototype Verification System (PVS) [Owre et al. 1992], 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 techniques at low levels, such as the arithmetic of real numbers and sets. Users can directly control proof development at a high level, for example by selecting proper user-defined proof strategies. A recently developed NASA PVS library [Butler 2004] has formalized and validated the elementary calculus including integration and differentiation. The library has been successfully applied to verify a practical aircraft traffic control system [Muñoz et al. 2006]. These strengths of PVS are useful for achieving our goal of developing the mechanical proof support for interval-based specification languages.

In this article, we first present a way to systematically develop a verification system for TIC based upon PVS. We faithfully encode the TIC semantics using the PVS specification language. A tool is also implemented to support the automatic translation from TIC models to PVS specifications and graphical editing of TIC models. We further define a collection of supplementary reasoning rules and proof strategies to simplify the reasoning process. In addition, these proof strategies assist users by hiding the detailed encoding of TIC.

Using the verification system, we can rigorously analyze TIC models at the interval level by using the validated supplementary reasoning rules. Proofs at low levels, such as propositional logic and real numbers, can be automatically discharged by the PVS prover. The system was applied in our published work [Chen et al. 2007] to help discover semantic incompleteness and a bug in Simulink [The MathWorks 2008], which is a graphical toolkit for modeling and simulating dynamic systems. As illustrated in this article, we identify two subtle flaws in the original TIC reasoning rules, using our rigorous validation.

We further extend our verification system to support other interval-based specification languages, particularly DC. We formalize the DC constructs using TIC. Based on the encoding, we check the correctness of the DC axioms and reasoning rules in our system. Proofs of DC models can thus be rigorously carried out in a manner similar to the corresponding pencil-and-paper DC arguments. We apply the resulting system to a common DC case study, and an incorrect step in its original proof is discovered.

This article is based on our preliminary paper [Chen et al. 2008]. Going beyond the previous paper, we generalize the verification system to support another popular real-time specification language (DC) besides the expressive notation, TIC. In addition, the presentation of the way to construct TIC semantics in PVS has been improved significantly. We also provide our full experimental study that has previously only been sketched and a detailed explanation of the verification undertaken.

The remainder of this article is organized as follows. Related work is reviewed in the next subsection. Section 2 introduces the characteristics of TIC, DC, and PVS. Section 3 illustrates the development of the verification system for TIC, and shows the feasibility and benefits of our approach using an experimental study. Section 4 demonstrates the work on enhancing the verification system to support DC, together with the application of a DC case study. Conclusions and future work are provided in Section 5.

1.1 Related Works

We are aware of two other approaches to supporting TIC by exploiting theorem provers. Dawson and Goré [2002] applied Isabelle/HOL [Nipkow et al. 2002] to formalize and check the correctness of TIC reasoning rules. They focused on the encoding of TIC reasoning rules. Their encoding of the TIC semantics was incomplete; the construction (such as operators used in arithmetics and inequalities) of TIC predicates and expressions that make up TIC models was not modeled. It is hence difficult to support the TIC verification in general as the interpretation of TIC models is essential. Cerone [2001] implemented the axiomatization of TIC in the theorem prover Ergo. Cerone defined extensive axioms of the time domain, whereas we use the theory of real numbers provided with PVS. Cerone allowed a concatenation to be formed by two *both-open* intervals, and that is different from the original definition [Fidge et al. 1998b], which requires two concatenated intervals to meet exactly with no gap. Moreover, Cerone's work dealt with only five reasoning rules. In contrast, we have constructed complete TIC semantics systematically in PVS, and validated all reasoning rules. One subtle flaw has been discovered for the first time. Furthermore, our verification system supports advanced mathematical analysis such as integral calculus, which is not handled by the previous works.

Some researchers have investigated the machine-assistant proof for DC. Heilmann [1999] constructed a proof assistant for DC based on Isabelle [Paulson 1994]. The encoding of DC in Heilmann's approach was syntactic; the DC syntax and proof rules were introduced as entities of the Isabelle logic. One advantage of this encoding is that users can carry out DC proofs without having considerable knowledge of the Isabelle logic. On the other hand, Skakkebæk and Shankar [1994] implemented a proof checker by encoding the DC semantics within the PVS higher-order logic. The semantic encoding gives an advantage in utilizing the decision procedures as an integral part of PVS. They also defined a set of PVS strategies to enable users to work directly with the syntax and proof rules of DC and not their encoding in PVS. Although DC is undecidable for continuous time in general, it is mostly decidable for discrete time. Chakravorty and Pandya [2003] developed a tool to check the validity of a subclass of discrete-time DC. However, in that work, the duration operator (which is the key construct of DC) is not semantically encoded, and its properties are assumed as axioms. In our approach, we encode the duration operator based on the latest NASA PVS library, and we can hence directly validate those properties regarding DC durations in our verification system.

There exists some work on developing tools for various interval-based logics. Mattolini and Nesi [2001] presented temporal-interval logic with compositional operators (TILCO) with formal proof support from Isabelle/HOL. In TILCO, the time is discrete and the temporal domain is the set of integers; the minimum time interval corresponds to one time unit. On the other hand, the time domain of both TIC and DC is continuous-time. Moser et al. [1996] described a set of tools for real-time graphical interval logic (RTGIL) to assist specifying and verifying time-bounded properties of concurrent real-time systems. Intervals in RTGIL are derived from sequences of states and transitions that form their end-points. An interval is graphically depicted by a *left-closed and right-open* line segment. TIC and DC differ from RTGIL in that they treat intervals as primitive semantic objects, and they are well-suited for modeling and reasoning about accumulative behavior. The operator $\int$, for instance, can be used to specify the duration of a fragment of a computation during which a predicate holds.

Recently, the interval concept has been used by Mok et al. [2002] to capture the uncertainty in the exact times of event occurrences when monitoring timing constraints. In their model, a time stamp of an event consists of a pair of time values: the start and the end times. They assume that the maximum length of a time stamp is bounded and known to a monitoring system in advance. Yu et al. [2006] have extended the work to support the analysis of timing constraint violations caused by transient failure models with exponential distribution. In both of these, system models are highly abstract, as functional requirements are not their main concern. On the other hand, TIC can specify both functional and timing requirements. With the support of mathematical analysis in TIC and in PVS, we can reason about functional requirements using our verification system.

An alternative approach for modeling real-time systems is based on the automata theory. Hybrid automata [Alur et al. 1992] are used to model embedded systems with continuous variables, whose value may change at various rates. Arbitrary linear constraints are allowed for invariance conditions and triggering conditions. Hytech [Henzinger et al. 1997] is a symbolic model checker for linear hybrid automata, a subclass of hybrid automata that can be analyzed automatically by computing with polyhedral state sets. Timed automata [Alur and Dill 1990] are a special subclass of hybrid automata in which all continuous variables increase their values at a uniform rate and only upper-bound and lower-bound inequalities of clocks are allowed. Uppaal [Larsen et al. 1997] is a tool for specifying, simulating, and verifying real-time systems modeled in timed automata.

2. BACKGROUND

In this section, we briefly present the background information of the notations and tools that are involved in this article, namely Timed Interval Calculus (TIC) [Fidge et al. 1998b], Duration Calculus (DC) [Zhou and Hansen 2004], and Prototype Verification System (PVS) [Owre et al. 1992]. Readers who are interested in knowing more may refer to the respective references. Table I lists the special symbols used in this article, with their informal meanings.

Table I. Symbols and their Informal Descriptions

Symbol	Explanation	Symbol	Explanation
$\mathbb{T}$	time domain	$(\ )$	a set of *both-open* intervals
$\mathbb{I}$	all non-empty intervals	$[\ )$	a set of *left-closed*, *right-open* intervals
α	starting point of an interval	$(\ ]$	a set of *left-open*, *right-closed* intervals
ω	ending point of an interval	$[\ ]$	a set of *both-closed* intervals
δ	length of an interval	$[\!(\ )\!]$	a union of the four interval brackets
$\curvearrowright$	connect two sets of intervals	$\frown$	chop two Duration Calculus formulas

2.1 Timed Interval Calculus

TIC is set-theory based and reuses the well-known formal notation, Z [Woodcock and Davies 1996] mathematical and schema notations. It uses total function of continuous time to represent system dynamics [Mahony and Hayes 1992], and defines *interval brackets* to concisely model system behavior in terms of intervals [Fidge et al. 1998b]. Interval endpoints can be explicitly accessed, and hence TIC can model behavior over special intervals with particular endpoints.

The time domain, $\mathbb{T}$, is denoted by nonnegative real numbers. An interval is a continuous range of time points, and intervals are classified into four basic types based on the inclusion/exclusion of endpoints. For example, *both-closed* intervals are defined in the following Z axiomatic style, where $\mathbb{P}$ is the power-set constructor and $\mathbb{R}$ denotes real numbers. Three other types of intervals, namely, *both-open*, *left-open and right-closed*, and *left-closed and right-open* are defined similarly.

$$
\begin{array}{|l}
[\_ \ldots \_] : \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{P}\mathbb{T} \\
\hline
\forall x, y : \mathbb{R} \bullet [x \ldots y] = \{z : \mathbb{T} \mid x \leq z \leq y\}
\end{array}
$$

There are three primitive types of elements to construct TIC models.

—*Constants*. A constant is independent of time points and intervals. For example, a maximum temperature that is a real number can be declared as a constant $MaxTmp : \mathbb{R}$.

—*Timed traces*. Timed traces model the dynamic (continuous or discrete) variables of systems. A timed trace is a total function from time domain to the type of the variable. For example, temperature in a room is represented by a timed trace $Tmp : \mathbb{T} \rightarrow \mathbb{R}$.

—*Interval operators*. Distinct from the timed traces, an interval operator is a function from intervals to the type of the variable. There are three predefined interval operators in TIC, namely, α, ω, and δ, which have the same type, $\mathbb{I} \rightarrow \mathbb{T}$, where the symbol $\mathbb{I}$ denotes all nonempty intervals. These operators respectively return the starting point, ending point, and length of an interval.

A key construction of TIC is interval brackets. A pair of interval brackets associated with a *predicate* returns all intervals during which the predicate is true everywhere. An enclosed predicate is usually expressed in the first-order logic, and all references to the time domain and intervals are elided in the predicate. For example, a *TIC expression*, $[Tmp(\alpha) \leq Tmp]$, represents a set of *both-closed* intervals, and in each interval the value of *Tmp* at each time point is not less than the value at the starting point. This is shown in the following equivalent expression (a set comprehension), where the domain of *Tmp* is time, and the domain of α is intervals. Without using the interval brackets, $⁅\ ⁆$, we need to explicitly associate timed traces and interval operators with their corresponding time points and intervals.

$$
\begin{aligned}
&⁅\mathit{Tmp}(\alpha) \leq \mathit{tmp}⁆\\
&\quad = \{x, y : \mathbb{T} \mid (\forall t : [x \ldots y] \bullet \mathit{Tmp}(\alpha([x \ldots y])) \leq \mathit{Tmp}(t)) \bullet [x \ldots y]\}
\end{aligned}
$$

Set operators such as $\cup$ and $\cap$ are applied to compose TIC expressions. To capture the sequential behavior over intervals, TIC defines an operator $\frown$ to concatenate two sets of intervals end-to-end, namely, no gap and no overlap.

$$
\begin{array}{|l}
\_ \frown \_ : \mathbb{P}\mathbb{I} \times \mathbb{P}\mathbb{I} ⇸ \mathbb{P}\mathbb{I}\\
\hline
\forall X, Y : \mathbb{P}\mathbb{I} \bullet X \frown Y =\\
\qquad \{z : \mathbb{I} \mid \exists x : X;\ y : Y \bullet z = x \cup y \wedge (\forall t1 : x;\ t2 : y \bullet t1 < t2)\}
\end{array}
$$

By specifying relationships among TIC expressions, we can model system properties and requirements at the interval level. For example, the following *TIC predicate* as a subset relationship specifies a periodic behavior that a detector should store the temperature *Tmp_in* every k time units, where $\mathbb{N}$ denotes natural numbers.

$$
⁅\exists i : \mathbb{N} \bullet \alpha = i * k \wedge \omega = (i + 1) * k⦆ \subseteq ⁅\mathit{store} = \mathit{Tmp\_in}(\alpha)⦆
$$

In this TIC predicate, the TIC expression at the left side of $\subseteq$ decomposes the time domain into a sequence of *left-closed and right-open* intervals (by $⁅\ ⦆$), and each interval lasts k time units; the TIC expression at the right side depicts the periodic update of the stored temperature.

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 relationships of these variables in its *predicate part*. The following schema represents the previous detector, where the symbol ⇴ (defined in Fidge et al. [1998a]) indicates that *Tmp_in* is a continuous function over the time domain.

$$
\begin{array}{|ll}
\multicolumn{2}{l}{\mathit{Detector}}\\
\hline
\mathit{Tmp\_in} : \mathbb{T} ⇴ \mathbb{R};\ \mathit{store} : \mathbb{T} \rightarrow \mathbb{R} & \text{[Declaration]}\\
\hline
⁅\exists i : \mathbb{N} \bullet \alpha = i * k \wedge \omega = (i + 1) * k⦆ \subseteq ⁅\mathit{store} = \mathit{Tmp\_in}(\alpha)⦆ & \text{[Predicate]}\\
\hline
\end{array}
$$

TIC contains a set of primitive rules about the properties of sets of intervals. These rules are used to carry out TIC verification at the interval level. For example, the following rule states that for any nonpointer interval (namely, $\delta > 0$) in which a predicate holds, the interval can be decomposed into two concatenated subintervals and the predicate is still true in each subinterval.

If α, ω, and δ do not appear free in a predicate P, then we have
$⟦⦅P \wedge \delta > 0⦆⟧ = ⟦⦅P⦆⟧ \frown ⟦⦅P⦆⟧$.

In the above specification, the interval brackets $⟦⦅\ ⦆⟧$ denote a union of four basic types of interval brackets: $⟦⦅P⦆⟧ == ⦅P⦆ \cup ⦅P⁆ \cup ⁅P⦆ \cup ⁅P⁆$. This operator is used when predicates are independent of interval endpoints. Moreover, the rule in the previous example is valid provided the time domain is continuous.

Using TIC, we can specify important requirements such as *safety* and *bounded liveness* requirements, and prove that system designs imply requirements by deduction. A proof is usually decomposed into several subproofs, and each subproof concentrates on a simple requirement of a subsystem. Each deductive step in a proof is reached by rigorously applying a hypothesis (as an axiom), a TIC reasoning rule, a mathematical law, or a proved requirement from a subproof.

2.2 Duration Calculus

DC is a logic-based approach to formal design of real-time systems. The basic calculus of DC [Zhou et al. 1991] and its extensions, including Mean Value Calculus [Zhou and Li 1994] and Extended Duration Calculus [Zhou et al. 1993], are founded on the interval temporal logic and integral calculus. We consider the basic DC in this article. It axiomatizes state durations for the Boolean state model, namely, integrals of Boolean-valued functions. Other extensions of the basic DC are introduced by adding extra axioms that formalize the extended models and also their interrelations with the Boolean state model.

In the basic calculus of DC (abbreviated as DC henceforth), *state variables* are the basic type to model system states. A state variable P is a function from time to Boolean values $\{0, 1\}$, namely, $P : \mathbb{T} \to \{0, 1\}$. Furthermore, DC assumes that state variables hold *finite variability*, which stipulates that a state variable can only change its value finitely many times in any bounded interval. This assumption ensures that state variables are integrable in every interval.

State expressions are formed by applying propositional logic operators over state variables, following the abstract syntax: $S ::= 0 \mid 1 \mid P \mid \neg S_1 \mid S_1 \wedge S_2$, where S, S_1, and S_2 are state expressions. Semantically, a state expression returns a value of 0 or 1 at a time point. For example, two state variables, *Gas* and *Flame*, are introduced in a gas burner system to characterize the flowing and burning of gas. $Gas(t) = 1$ means that gas is flowing and $Flame(t) = 1$ means that flame is burning. Hence, $Gas \wedge \neg Flame$ is the state expression specifying the leaking of gas, and it is interpreted with respect to a given time point t in the following way, where $(\neg Flame)(t) = 1 - Flame(t)$.

$$(Gas \wedge \neg Flame)(t) = \begin{cases} 1 & \text{if } Gas(t) = 1 \text{ and } (\neg Flame)(t) = 1 \\ 0 & \text{otherwise} \end{cases}$$

Temporal variables in DC, which are real-valued functions of intervals, can have a structure $\int S$ to denote the *duration* of a state expression S over a closed time interval $[b, e]$, where $b \leq e$. The duration is the accumulated presence time of S in the interval, namely, $(\int S)([b, e]) = \int_b^e S(t)dt$. Another predefined temporal variable in DC is ℓ, which denotes the interval length, namely, $\ell([b, e]) = e - b$.

DC *terms* are built upon temporal variables or constants using mathematical operators. DC *formulas* are composed by constraining DC terms or subformulas. Besides the conventional predicate logic operators such as the disjunction $\vee$

and the universal quantifier $\forall$, DC also adopts the chop operator $\frown$. A formula $\phi \frown \psi$, where ϕ and ψ are formulas, is satisfied by an interval if and only if the interval can be chopped into two adjacent *both-closed* subintervals such that the first subinterval satisfies ϕ and the second satisfies ψ. Based on the chop operator, two commonly used operators over subintervals, $\Diamond$ (eventually) and $\Box$ (always), are defined as follows: $\Diamond\phi == (\mathrm{true} \frown \phi) \frown \mathrm{true}$ and $\Box\phi == \neg \Diamond(\neg \phi)$. For example, an interval $[b, e]$ satisfies $\Diamond\phi$ provided there exist c and d such that $b \leq c \leq d \leq e$ and the interval $[c, d]$ satisfies ϕ.

A formula is *valid* in DC if and only if it holds in all intervals. For instance, a design property of a gas burner is that any leak represented by a state variable *Leak* should not last longer than one time unit. This design property can be represented by the formula, $\Box(\llbracket Leak \rrbracket \Rightarrow \ell \leq 1)$, where $\llbracket Leak \rrbracket$ is an abbreviation of the formula $\int Leak = \ell \wedge \ell > 0$. Note that $\Box$ indicates that the design property holds in any interval.

Properties of state durations are declared as axioms in DC. These axioms are important for deriving DC reasoning rules in DC proofs. Taking the axiom **DCA5** from Zhou and Hansen [2004] as an example, the axiom as shown below captures the relationship between the duration length (where x and y are nonnegative real numbers) of a state expression and the chop operator.

DCA5 $(\int S = x) \frown (\int S = y) \Rightarrow \int S = x + y.$

As we will show in Section 4.2, the DC axioms are declared as lemmas and they can be formally validated using our verification system.

Although DC and TIC possess similar capabilities, their bases are different. TIC is based on set theory, while DC is based on interval temporal logic. Furthermore, TIC supports explicit references of interval endpoints, which can specify properties over special intervals with particular endpoints.

2.3 Prototype Verification System

PVS is an integrated environment for formal specification and formal verification. It builds on over 25 years of experience at SRI in developing and using tools to support formal methods. 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.

New types can be defined from the built-in types using type constructors such as *predicate subtypes* and *record types*. A predicate subtype denotes a subset of individuals in a type satisfying a given predicate. For example, nonzero real numbers are written as `{x: real | x /= 0}`. Note that types in PVS are modeled as *sets*. Record types are of the form `[# a1:t1, ..., an:tn #]`, where `a1` is a *record accessor* and `t1` is the associated type.

Overloading is supported in PVS. In particular, functions can have the same name as long as they have different argument types. Specifications in PVS are built from *theories*, which usually contain type declarations, functions, 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 in which all leaves are trivially true. Each node in a proof tree is a *proof goal*, which is a sequent consisting 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 prover provides a collection of primitive proof commands 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, simplification, rewriting, and applying decision procedures. Users can introduce more powerful proof strategies that 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. These theories cover much of the mathematics needed to support specification and verification in PVS. Recently the NASA PVS library has formalized the definitions of *limits*, *derivatives*, *continuity*, and *integration*. The library has also validated a number of properties of these definitions and hence supports the rigorous analysis of continuous dynamics.

3. A VERIFICATION SYSTEM FOR TIC

We describe the development of a verification system for TIC built upon PVS in this section. The TIC semantics is faithfully encoded and all TIC reasoning rules are validated in PVS. A translator is implemented to automatically translate TIC models to PVS specifications. A collection of supplementary rules and proof strategies are defined to ease the verification process. An experimental study is provided at the end to show the feasibility and effectiveness of the verification system.

3.1 Encoding TIC Semantics in PVS

The encoding of TIC semantics forms a foundation from which we formalize the TIC reasoning rules and carry out the verification of TIC models. An important requirement is that the resulting PVS specifications should be concise in a way close to the structure of the TIC models, so any diagnostic information obtained at the level of PVS can be easily reflected back to the level of TIC. The PVS theories of the TIC semantics are formed in a bottom-up manner, and each subsection below corresponds to a PVS theory. Simple theories are hence used to compose complex ones (the complete PVS theories are available online [Chen 2008]). To avoid the problem of subgoal explosion, which often occurs in reasoning procedures, we model TIC constructs, especially the interval brackets and concatenation operator, in a hierarchical manner. Moreover, the flexible style of type declaration in PVS reduces the size of the PVS specifications.

3.1.1 *Time and Interval Domains.* The time domain is represented by the PVS built-in type `nnreal` as a set of nonnegative real numbers.

```
Time: TYPE = nnreal;
```

An interval is modeled as a tuple and its type is GenInterVal as shown in the following: the first element (invt) indicates the interval type (for example, CO indicating that the interval is *left-closed and right-open*); the second element is also a tuple, which consists of the starting point (stp) and the ending points (etp).

```
Interval_Type: TYPE = {OO, OC, CO, CC};
GenInterval: TYPE = [invt: Interval_Type, {stp, etp: Time | stp <= etp}];
```

The following type II denotes all nonempty intervals, and the constraints of interval endpoints with respect to interval types are captured. For example, the predicate i‘1 = CC and i‘2‘1 <= i‘2‘2 specifies that the ending point can be equal to the starting point if the interval is *both-closed* (indicated by CC), where the apostrophe ‘ is the PVS projection operator to refer to components in a tuple. By using the predicate subtype technique in PVS, specific interval types are easily constructed based on II. For instance, COInteral, which represents *left-closed and right-open* intervals restricts the interval type to be CO.

```
II: TYPE = {i: GenInterval | (i‘1 = CC and i‘2‘1 <= i‘2‘2)
                          or (i‘1 /= CC and i‘2‘1 < i‘2‘2)};
COInterval: TYPE = {i: II | i‘1 = CO};
```

3.1.2 *Timed Traces and Interval Operators.* A timed trace (Trace) is a function from time to the real numbers. We further model *discrete* timed traces (BTrace) whose ranges consist of two values, 0 and 1.

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

Interval operators are functions of intervals. They are independent from the inclusion/exclusion of interval endpoints. That is to say, we only need to define their functionalities with respect to II without respectively listing those of specific interval types (for example, COInterval). The following PVS specifications correspond to three predefined TIC interval operators, namely, α, ω, and δ.

```
ALPHA(i: II): Time = i‘2‘1;  OMEGA(i: II): Time = i‘2‘2;
DELTA(i: II): Time = OMEGA(i) - ALPHA(i);
```

3.1.3 *Expressions and Predicates.* As a modeling feature of TIC, the references to the time domain and interval domain are elided in the expressions and predicates that are enclosed in a pair of interval brackets. However, it is necessary for these references to be explicitly shown for the correct interpretation of the expressions and predicates. We declare expressions (TExp) and predicates (TPred) to be functions in PVS where time and intervals compose the domain.

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

Primitive elements of TIC form expressions and in turn, predicates. An element is a constant, a timed trace, or an interval operator. By the overloading mechanism of PVS, the following function `LIFT` performs different functionalities according to the type of its first argument.[1] `LIFT` returns the value at a time point, `t`, for a timed trace while evaluating an interval operator with respect to an interval, `i`.

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

When interpreting an expression of TIC, we pass its parameters denoting the time domain and interval domain to its constituent expressions. This propagation repeats until all constituent expressions are primitive elements. For instance, a (prefix) subtraction of TIC is interpreted in the following in PVS, where the pair `(t, i)` is passed to the component expressions `el` and `er`. A similar approach is used to handle predicates (a disjunction of TIC is provided as an example).

```
-(el, er)(t, i): real = el(t, i) - er(t, i);      % el, er: TExp
or(pl, pr)(t, i): bool = pl(t, i) OR pr(t, i);    % pl, pr: TPred
```

We remark that TIC supports elementary calculus including integration and differentiation. The calculus is handled in our system with the formal definitions from the NASA PVS library. For example, the expression $\int_{\alpha(i)}^{\omega(i)} tr$ is represented by the following PVS function `TICIntegral`, which invokes function `Integral`, which is defined in the NASA PVS library.

```
TICIntegral(tr)(t, i): real = Integral(ALPHA(i), OMEGA(i), tr)
```

3.1.4 *TIC Expressions.* A TIC expression denotes a set of intervals. The basic structure of TIC expressions is a pair of interval brackets that enclose a predicate. Common set operators can be applied to make up complex TIC expressions. Here we demonstrate how to encode the TIC expressions with interval brackets and the special set operator of TIC, namely the concatenation operator. Other types of TIC expressions can be constructed by the built-in functions in the PVS set theory.

A pair of interval brackets enclosing a predicate represents a set of intervals, and in each interval the predicate holds everywhere, namely, at all time points of the interval. In the following PVS specifications, the function `t_in_i` detects whether a time point is within an interval according to the interval type. Note that there are four basic types of interval brackets. Based on `t_in_i`, we define the function `Everywhere?` to check if a predicate holds in an interval. TIC expressions containing general interval brackets ⟦- -⟧ are thus modeled

[1] Characters following the symbol '%' are comments in PVS.

by the function AllS. In addition, TIC expressions of basic types of interval brackets are easily specified by applying the predicate subtype mechanism (for example, the function COS for [ )).

```
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 two connected intervals must meet exactly, that is, with no overlap and no gap. There are thus eight correct ways of concatenating from four basic types of intervals. Instead of modeling each one individually, we represent all eight cases together by the following function concat. The function takes two sets of intervals as parameters (namely, iisl and iisr), which may contain any type of interval, and each interval in the returned set is composed by two adjacent intervals respectively from two parameters.

```
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 = OC)))
 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):
    ConcatType(i1, i2, i) AND member(i1, iisl) AND member(i2, iisr) AND
    OMEGA(i1) = ALPHA(i2) AND ALPHA(i1) = ALPHA(i) AND OMEGA(i2) = OMEGA(i)};
```

In these PVS specifications, function ConcatType constrains the types of concatenated intervals. The constraints cover all eight cases. Being different from other constraints in concat (for example, OMEGA(i1) = ALPHA(i2) indicates that a concatenated interval's ending point is equal to the starting point of the other), the application of ConcatType in concat encapsulates the predicates of interval types at a lower level. That is to say, we create a hierarchical structure. This structure is useful to avoid the problem of subgoal explosion, which is often encountered during reasoning procedures in PVS. That is, the PVS prover automatically splits a proof goal into a number of subgoals at a proof step, although the split is unnecessary at that step since there are many repetitive proof commands used to discharge those subgoals. For instance, if we directly specify eight constraints of interval types in concat, the prover would automatically split one proof goal into eight subgoals when expanding the concatenation definition in PVS, although these subgoals can be proved by applying many repetitive proof commands.

So far, we have carefully formalized the TIC constructs in PVS, while the way of handling TIC schemas and TIC predicates will be presented in Section 3.3.1. During the encoding, the overloading mechanism has assisted us to define the function LIFT with different functionalities, and the higher-order logic of the PVS specification language has facilitated the interpretation of expressions and predicates of TIC in a bottom-up manner. These PVS theories of the TIC semantics form a base from which to validate the TIC reasoning rules and support mechanical verification of TIC models, as we will show in the following sections.

3.2 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. Guaranteeing their correctness is thus necessary and important. In this section we first describe the challenge of validation. Next we demonstrate the flaws discovered from our rigorous reasoning process and provide remedies.

Checking TIC reasoning rules is not trivial. Though some of these rules are automatically proved by the PVS prover, others require complicated analysis covering all types of intervals and various types of predicates (for example, whether a predicate is dependent on interval operators). Taking the rule introduced in Section 2.1 as an example, its PVS specification is represented based on the encoding in the previous section, where function `No_Term?` returns true when a predicate `pl` is independent with interval operators.

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

To validate this rule, we need to consider the concatenation of two sets of all types of intervals. Therefore there are eight cases. In the reasoning process, human interactions are helpful to increase the efficiency. Our simplified proof goal in the following aims to show that there exist two concatenated intervals, which form an interval `x!1` and satisfy the hypotheses depicted by three antecedents (prefixed by negative integers). For instance, the antecedent at `[-1]` restricts the type of `x!1` to be *left-closed and right-open*. To prove the goal, we select the middle point of `x!1` as the connecting point, namely, `(ALPHA(x!1) + OMEGA(x!1))/ 2`, and then instantiate the requested intervals by applying our defined proof strategy `assignconct`.

```
[-1]  TypeOf(x!1) = CO
[-2] No_Term?(pl1!1)
[-3]  AllS(pl1!1 AND LIFT(DELTA) > LIFT(0))(x!1)
|-------
[1]   concat(AllS(pl1!1), AllS(pl1!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)))")
```

The PVS prover always checks the correctness of assignments, so we are required to show that these user-specified intervals satisfy the concatenation definition indicated by the function `concat`. Doing so can thus prevent mistakes from users such as assigning two concatenated intervals with the *both-open* interval type.

During our rigorous validation of all TIC reasoning rules, two subtle flaws in the original reasoning rules have been discovered. We present these problematic rules with counterexamples, followed by their corresponding solutions, which have been validated in PVS.

- The *True and False* rule is frequently used to reason about safety requirements. The original rule states that a predicate P is true in all intervals if and only if its negation is true nowhere. That is, $\llparenthesis P \rrparenthesis = \mathbb{I} \Leftrightarrow \llparenthesis \neg\, P \rrparenthesis = \varnothing$. However, the implication, $\llparenthesis \neg\, P \rrparenthesis = \varnothing \Rightarrow \llparenthesis P \rrparenthesis = \mathbb{I}$, does not hold in certain circumstances.

  For example, let x be a timed trace having the value 1 from time points 5 to 7 and the value 0 elsewhere. It is obvious to see that the predicate $\neg\, P == x = 1 \wedge \delta = 3$ fails everywhere, although its negation $P == x \neq 1 \vee \delta \neq 3$ is false in some intervals such as the interval $[5 \ldots 8]$.

  To solve the problem, a stronger hypothesis is needed. The predicate within interval brackets should be independent of interval characteristics: the starting point, ending point and length of an interval. The modified rule is expressed in PVS as a lemma named `Emp_to_All`, where sets `emptyset` and `fullset` denote respectively the empty set and the set of all intervals.

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

- The *Concatenation Duration* rule is useful to deal with proofs involving concatenation. Using the rule, a set of intervals can be decomposed into two concatenated sets of intervals with specified interval lengths. So given a predicate P where interval operators do not occur, if we have $r, s : \mathbb{T}$ and $r > 0 \vee s > 0$, then we can deduce $\llparenthesis P \wedge \delta = r + s \rrparenthesis = \llparenthesis P \wedge \delta = r \rrparenthesis \frown \llparenthesis P \wedge \delta = s \rrparenthesis$.

  However this equality in terms of sets of intervals does not always hold. For example, if $r = 0$, then any interval of $\llparenthesis P \wedge \delta = r \rrparenthesis$ must be *both-closed* according to the interval definition. However, it is possible that $\llparenthesis P \wedge \delta = r + s \rrparenthesis$ contains intervals that are *left-open*, hence type conflict occurs. The conflict can be removed by a stronger assumption, namely, $r > 0 \wedge s > 0$.

We remark that this is the first time that the first flaw has been discovered (while the second flaw has also been observed by Dawson and Goré [2002]). These discoveries demonstrate the benefits of exploiting a theorem prover for rigorous verification.

Based on the lemma `Emp_to_All`, we further derive a new rule, `EmpCC_to_All` to reduce the proof complexity. When applying `Emp_to_All`, we have to show that the proof goal can be discharged with respect to four basic interval types, although usually each subproof follows a similar reasoning process. In contrast, the new rule allows us to focus on just one type of interval: *both-closed* intervals (as indicated by `CCS`).

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

Currently, we have validated all TIC reasoning rules in PVS. These rules can thus be applied as lemmas in PVS to verify TIC models. Two flaws have been discovered and fixed.

3.3 Translating TIC Models to PVS Specifications

Based on the encoding of TIC semantics in PVS (refer to Section 3.1), we present in this section the translation from TIC models to PVS specifications, with an adaptive temperature control system as an illustration. The translated PVS specifications closely follow the original TIC models, so the diagnostic information obtained at the level of PVS can be reflected back to the level of TIC.

3.3.1 *Representing TIC Models in PVS.* Using TIC, system properties and requirements are specified at the interval level. TIC schemas model the properties, and TIC predicates model the requirements. In the following, we explain the way to represent them in PVS.

—TIC schemas are used to structure and compose models; collating pieces of information, encapsulating them and naming them for reuse. Each schema represents a composite type, which is made up of a set of bindings; each binding relates a declared variable with its restrictive values. This modeling feature enables schemas to be used as types, to support the methodology of component-based design.

Each schema is represented by a PVS *record* type. We construct a set of records in PVS, where schema declarations are denoted by record accessors associated with corresponding types and schema predicates are used to constrain relationships over the record accessors. Moreover, implicit properties indicated by certain kinds of functions in TIC, such as continuity and integrability, are captured by additional constraints in PVS to further restrict the record type.

—A TIC predicate specifies a requirement of a system or some components, and is constructed based on the TIC schemas of the relevant system or components. Each TIC predicate is represented by a PVS theorem formula constructed based on the PVS specifications that represent corresponding TIC schemas.

These relationships between TIC models and PVS specifications can be informally illustrated in the following, where `temp` in the construction for schemas is an auxiliary variable to access record accessors in the generated `Predicate` specification.

schema_name
declaration
predicate

```
SchemaName: TYPE = {temp:
 [# Declaration #] |
 Predicate }
```

requirement_name == *predicate*

```
RequirementName : THEOREM
 Predicate
```

To automatically translate TIC models to PVS specifications, we develop an automated process which consists of three steps: *scanning*, *parsing*, and *translating*. A scanner splits TIC models into a sequence of meaningful tokens such as TIC interval brackets, mathematical operators and so on. A parser constructs a set of abstract syntax trees (ASTs) for TIC models based on those tokens. Each AST represents a TIC model, and the leaves of an AST denote the primitive elements of a TIC model: constants, timed traces, and interval operators. A translator traverses an AST in a top-down order to produce corresponding PVS specifications.

We have developed a tool using Java to implement the translation algorithm. Besides supporting the automatic translation, the tool also provides functionality to facilitate the use of TIC including graphical editing as well as syntax and type checking. The tool is available online [Chen 2008].

3.3.2 *A Temperature Control System.* A temperature control system [Nicollin et al. 1992; Tiwari et al. 2003] is a hybrid application that controls the temperature by turning a heater on and off. The system involves discrete logic and continuous dynamics, and is designed to fulfill important requirements such as safety requirements. We present the system properties and requirements in TIC and their translated PVS specifications. The system will be applied to demonstrate the advantages of our verification system, in the next section.

The control system is composed by two subsystems: a *plant* represents the physical environment where the temperature changes continuously following different integral equations based on the heater status; and a *controller* that turns the heater on and off according to the temperature of the plant.

In the *plant* subsystem, there are two conditions.

—When the heater (denoted by *heater_in*) is off, namely, $heater\_in = 0$, in an interval, say $[b, e]$, the temperature (by *tmp_out*) decreases and follows the integral equation: $tmp\_out(e) = tmp\_out(b) - 0.1 * \int_b^e tmp\_out$.

—When the heater is on in an interval $[b, e]$, the temperature increases and follows the integral equation: $tmp\_out(e) = tmp\_out(b) + 6 * (e - b) - 0.1 * \int_b^e tmp\_out$.

The TIC schema of this subsystem and the corresponding PVS record type are given in the following, where the PVS variables (for example, `Trace`) and functions (for instance, `LIFT`) for encoding TIC have been defined in Section 3.1. Other special symbols include the PVS projection function (`), which acts like the Z selector operator "." to select a component/attribute from a schema, and the *function composition* operator o in PVS. Note that the temperature is declared as a continuous function (indicated by ⇴); this characteristic is also captured by the function `continuous` from the NASA PVS library.

Plant

$tmp\_out : \mathbb{T} ⇴ \mathbb{R};\ heater\_in : \mathbb{T} \rightarrow \{0, 1\}$

$$[\!(heater\_in = 0)\!] \subseteq [\!(tmp\_out(\omega) = tmp\_out(\alpha) - (1/10) * \int tmp\_out)\!]$$
$$[\!(heater\_in = 1)\!] \subseteq [\!(tmp\_out(\omega) = tmp\_out(\alpha) + 6 * \delta - (1/10) * \int tmp\_out)\!]$$

```
Plant: TYPE = { temp: [# tmp_out: Trace, heater_in: BTrace #] |
    subset?(AllS(LIFT(temp‘heater_in) = LIFT(0)),
            AllS((LIFT(temp‘tmp_out) o LIFT(OMEGA)) =
                 (LIFT(temp‘tmp_out) o LIFT(ALPHA)) -
                    LIFT(1/10) * TICIntegral(temp‘tmp_out))) AND
    subset?(AllS(LIFT(temp‘heater_in) = LIFT(1)),
            AllS((LIFT(temp‘tmp_out) o LIFT(OMEGA)) =
                 (LIFT(temp‘tmp_out) o LIFT(ALPHA)) + LIFT(6) * LIFT(DELTA) -
                    LIFT(1/10) * TICIntegral(temp‘tmp_out))) AND
    continuous(temp‘tmp_out)};
```

In the *controller* subsystem, the heater must be on, namely, $heater\_out = 1$, when the temperature tmp_in is not higher than the minimum value 20; and the heater must be off when the temperature is not lower than the maximum value 40.

Controller

$$tmp\_in : \mathbb{T} \mathrel{\circ\!\!\!\rightarrow} \mathbb{R};\ heater\_out : \mathbb{T} \to \{0, 1\}$$

$$\llbracket\!(tmp\_in \le 20)\!\rrbracket \subseteq \llbracket\!(heater\_out = 1)\!\rrbracket \wedge \llbracket\!(tmp\_in \ge 40)\!\rrbracket \subseteq \llbracket\!(heater\_out = 0)\!\rrbracket$$

```
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)};
```

We specify the connections between these two subsystems and the initial situation of the control system in the following TIC schema. The heater is off and the temperature is 30 at time point 0.

System

$$con : Controller;\ pla : Plant$$

$$\mathbb{I} = \llbracket\!(con.tmp\_in = pla.tmp\_out)\!\rrbracket \wedge \mathbb{I} = \llbracket\!(pla.heater\_in = con.heater\_out)\!\rrbracket$$
$$\llbracket\!(\alpha = 0)\!\rrbracket \subseteq \llbracket\!(pla.tmp\_out(\alpha) = 30 \wedge con.heater\_out(\alpha) = 0)\!\rrbracket$$

```
System: TYPE = { temp: [# con: Controller, pla: Plant #] |
    fullset = AllS(LIFT(temp‘con‘tmp_in) = LIFT(temp‘pla‘tmp_out)) AND
    fullset = AllS(LIFT(temp‘pla‘heater_in) = LIFT(temp‘con‘heater_out)) AND
    subset?(AllS(LIFT(ALPHA) = LIFT(0)),
            AllS((LIFT(temp‘pla‘tmp_out) o LIFT(ALPHA)) = LIFT(30) AND
                 (LIFT(temp‘con‘heater_out) o LIFT(ALPHA)) = LIFT(0)))};
```

Based on these specifications of system properties, we can specify a safety requirement that the temperature is always within a valid range from the value 20 to the value 40 in all nonempty intervals.

$$Safety == \forall s : System \bullet \mathbb{I} = \llbracket\!(s.pla.tmp\_out \le 40 \wedge s.pla.tmp\_out \ge 20)\!\rrbracket$$

Using TIC, we can specify important timing requirements. As shown in the following, the requirement *Length* says that given an interval starting from the time point 0 (namely, $\alpha = 0$), 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 remark that this requirement is not supported in the original [Nicollin et al. 1992; Tiwari et al. 2003] as they lack the capacity to model continuous behavior.

$$Length == \forall\, s : System \bullet \llparenthesis \alpha = 0 \rrparenthesis \subseteq \llparenthesis \int s.con.heater\_out \leq \frac{3}{4} * \delta \rrparenthesis$$

In this section, we have presented a way to translate TIC schemas and TIC predicates to PVS specifications with the illustration of a hybrid application. We remark that representing TIC schemas as PVS record types supports the modeling technique of Z (namely, using schemas as types to handle large scale systems). For example, the record accessor `con` in the record `System` denotes the *controller* subsystem. This approach is different from that of Gravell and Pratten [1998], who discussed issues on embedding Z into both PVS and HOL [Gordon and Melham 1993]. They interpreted Z schemas as Boolean functions of record types and it is thus difficult to handle the case where schemas are declared as types. Moreover, a means to handle schemas was missing in the work of Stringer-Calvert et al. [1997], who applied PVS to prove Z refinements for a compiler development; their work focused on supporting partial functions of Z in PVS.

3.4 Machine-Assisted Proof Support for TIC

Verification of TIC models is nontrivial, since real-time computing systems usually contain continuous dynamics and requirements often involve arbitrary intervals and continuous time domain. We demonstrate in this section how our verification system facilitates verification with a high level of automation. We begin by defining a collection of supplementary rules of TIC and proof strategies to simplify the reasoning process. Next we present a general proof procedure to systematically analyze TIC models. In the end, we apply them to the temperature control system to show the advantages of the verification system.

3.4.1 *Supplementary Rules and Proof Strategies.* The TIC reasoning rules validated in Section 3.2 capture primitive properties of sets of intervals. They are inadequate to support domain-specific characteristics. 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 deduce that i can be decomposed into three connected intervals, where the values of *tr* are larger than *TH* everywhere in the last subinterval and are equal to *TH* in the middle subinterval. This property is expressed by the following PVS lemma, where the function `continuous` expresses the property that a timed trace `tr` is continuous over the time domain.

```
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)))));
```

Note that this domain-specific property is not captured by any existing TIC reasoning rule. It is derived from the classic intermediate value theorem of continuous functions. Its correctness has been validated in PVS, and we can hence apply `mid_ivl_exi` when analyzing continuous dynamics.

To make the reasoning process in PVS more automated and shield users from the detailed TIC encoding, we have developed several PVS proof strategies. Each strategy combines repetitive proof commands that are frequently used in practice. These strategies mainly cope with quantified PVS formulas, since the PVS prover possesses powerful capabilities (such as automatic deduction and simplification) on reasoning about primitive formulas that are represented in the propositional logic. According to the quantifier type, these strategies are classified into two groups. One eliminates the universal quantifier by Skolemization, and the other removes the existential quantifier by proper instantiation. In addition, they usually automatically expand PVS functions that encode the TIC semantics, so detailed encoding of TIC can hence be hidden from users. We present the strategy, `AssignInvlnTime`, which offers a flexible way to assign an interval and a time point to a user-specified formula.

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

Using this strategy, we can either instantiate explicit values of an interval (denoted by `ivl` at line 1) and a time point (by `pt`), or let the PVS prover automatically fix the values by using the PVS proof command `inst?` (at lines 5 and 7). This strategy handles all interval types, by repeatedly applying the basic PVS proof strategy `else`[2] from line 2 to line 4 to expand proper PVS functions that encode interval brackets. Note that at line 6 `AssignInvlnTime` automatically expands the function `Everywhere?` (defined in Section 3.1.4). In other words, the strategy hides the detailed encoding of TIC, namely, `Everywhere?`, from users.

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

3.4.2 *A General Proof Procedure.* In general, to verify TIC models is to prove that the logical implication relationships among TIC models are valid, to check whether the TIC schemas representing system properties imply the TIC predicates denoting requirements. A proof is a deduction that starts from hypotheses (namely, TIC schemas) and proceeds in a forward manner. Each deductive step involves applying a TIC reasoning rule or a mathematical law

[2] The `else` proof strategy executes the first step, and if that does nothing, then the second step is executed.

to a hypothesis. Usually a proof can be decomposed into several subproofs over subsystems.

Our primary aim of developing the verification system is to systematically carry out the verification with a high degree of automation. Moreover, the reasoning process in the system is intended to closely follow the manual TIC arguments. Based on our experiments, we describe a general proof procedure with respect to a proof goal that is a PVS sequent (introduced in Section 2.3), which is initially expressed in terms of intervals. The proof procedure is effective and efficient in practice to accomplish our aims. The main objective of the procedure is to eliminate quantified formulas in the sequent by assigning appropriate values to intervals and time points. Following that, the PVS prover can directly manipulate the sequent and automatically discharge most of the tedious proof, for instance reasoning about linear arithmetic and sets.

(1) *Introduce new antecedents representing system properties to the sequent*. At the beginning of a proof, the consequent of the sequent is a PVS theorem that denotes a requirement. The consequent only contains the names of PVS records that model the properties of a system or components. These records can be inserted as new antecedents to the sequent by applying the PVS proof command `typepred` to the relevant record names.

(2) *Use TIC reasoning rules and supplementary rules*. The sequent is modified in two directions. (1) *Backward proof*: generating new consequents if some of the current consequents match the conclusion of a rule. (2) *Forward proof*: generating new antecedents if some of the current antecedents satisfy the hypotheses of a rule.

(3) *Exploit proved lemmas*. These lemmas are usually subgoals that capture requirements over some components/subsystems, which can be used to compose complex proofs.

(4) *Instantiate intervals and time points*. This step removes the quantifiers that indicate intervals and time points in the sequent. The values of intervals and time points can be manually assigned by users or automatically fixed by the PVS prover.

(5) *Adopt mathematical laws*. As TIC models often contain continuous dynamics modeled by the integral and differential calculus, we can choose dedicated mathematical laws from the NASA PVS library to support the analysis.

(6) Apply the PVS proof command `grind` as the last step to automatically discharge the proof goal.

In this procedure, the first three steps manipulate a proof goal at the interval level, and the last step reduces human effort on checking proofs at the low level by taking advantage of the powerful PVS reasoning capabilities. Furthermore, for Steps 4 and 5, human heuristics sometimes improve the efficiency, for example, assigning a proper interval value by hand instead of letting the PVS prover try out all possible instantiations. In the next section, we will demonstrate how the proposed proof procedure facilitates the analysis of our case study.

3.4.3 *Evaluation with the Temperature Control System.* We present the verification of two requirements specified in Section 3.3.2. The first illustrates the advantages of the general proof procedure, and the second demonstrates the ability to handle induction proofs in the verification system. At the end, we summarize the experimental result. The detailed specifications of the proved lemmas with their complete proofs can be found online [Chen 2008].

Proof of the Safety Requirement. Instead of checking the temperature in all intervals, we use the proof by contradiction mechanism to show that there is no interval during which the temperature is outside the valid range. According to the reasoning rule `EmpCC_to_All` defined in Section 3.2, we further reduce the proof complexity by only checking that no *both-closed* interval exists.

The proof is divided into two subproofs that check whether the temperature is greater (or lower) than the maximum (or minimum). Each subproof relies on a lemma that depicts the continuous behavior of the temperature. For example, the lemma *Decreasing* specifies here that the temperature at the ending point of a *both-closed* interval is not higher than that at the starting point of the interval provided the temperature is not lower than the value 40 in the interval.

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

This lemma can be systematically proved by the general proof procedure proposed in Section 3.4.2; the proof commands are listed 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") (ground)
 9:       (("1" (grind)) ("2" (grind))
10:        ("3" (use "cont_Integrable?")(grind))
11:        ("4" (skosimp) (grind))))
12:      ("2" (rewrite "Invariant_True_R")
13:        (expintervaltotime 1) (grind))))
14:    ("2" (rewrite "Invariant_True_R")(grind))))
```

—The properties of the whole system, `s!1`, and its subsystems, the controller `s!1‘con` and the plant `s!1‘pla`, are added as new antecedents to the proof goal by applying `typepred` to relevant names (at lines 3 and 5). For instance, `(typepred "s!1")` introduces the connections between `s!1‘con` and `s!1‘pla`.

—The proof strategies defined in Section 3.4.1 make the reasoning process more automated (at lines 2, 4, 6, and 13). The application `(assignsubset -1)` at line 4 for example directs the PVS prover to automatically instantiate an interval to the first antecedent of the sequent.

—As the temperature changes continuously, we exploit special lemmas from the NASA PVS library (at lines 7 and 10) to cope with advanced analysis.

The lemma `Integral_ge_0` (at line 7) represents the mathematical law that the integral of a function over a closed interval is nonnegative if this function is integrable and has only nonnegative values in the interval.

—At each branch of a subproof, invoking the proof command `grind` automatically discharges the proof (at lines 9, 10, 11, 13 and 14).

In this verification, all instantiations of intervals and time points are automatically accomplished. Nevertheless the assignment at line 8 is manual for the mathematical law of integral calculus, and this is acceptable, as human heuristics are helpful to efficiently choose proper arguments of the law. Note that the PVS prover always checks the correctness of user-specified values, so `OMEGA(x!1)` must be less than `ALPHA(x!1)` in the manual assignment.

Proof of the Length Requirement. Based on the assumption of *finite variability* [Zhou and Hansen 2004], we apply the *proof by induction* mechanism to show that the requirement holds in any interval. The finite variability ensures that a discrete-valued state in a bounded interval changes only finitely many times. In other words, we assume that an interval can be classified into one of the following groups with respect to a discrete timed trace: (1) the timed trace is constant during the interval; or (2) the interval can be decomposed into a sequence of connected subintervals where the trace has different values in adjacent subintervals. We formalize this assumption in PVS as presented in Section 4.1.1, and hence we can analyze behavior over arbitrary intervals by means of induction.

The *Length* requirement is concerned with intervals that start with time point 0, namely $\llparenthesis \alpha = 0 \rrparenthesis$. Since the heater status *heater_out* in the controller is a discrete timed trace; according to finite variability, each of these intervals can be divided into a sequence of concatenated subintervals, where the heater is off in the first subinterval (the initial condition specified by the schema *System*) and the heater is either off or on in the last subinterval. Moreover, the integration of the heater status is 0, namely, $\int heater\_out = 0$, when the heater is off, and we can hence focus on the analysis for a subset of the intervals with an additional constraint: the heater is on in the last subinterval. These special intervals can be constructed by the function *superCon* defined in the following. It takes three parameters: k is natural number acting as a counter, *base* is a set of intervals, and *unit* is a set of intervals as well. The function returns a set of intervals by repeatedly concatenating *unit* for k times to *base*.

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

These special intervals can hence be easily represented by an application of *superCon*, $superCon(k, base, unit)$, where the values of the parameters *base* and *unit* are shown in the following. *Base* denotes the first state transition of the heater, turning on from its initial off state; and *unit* denotes a sequential behavior of the heater, becoming on from off.

Table II. Experiment Results of the Verification of the Temperature Control System

Lemma Name	Steps	Time (*Seconds*)	Lemma Name	Steps	Time (*Seconds*)
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

$$
\begin{aligned}
base &= \lceil\!\lceil \alpha = 0 \land s.con.heater\_out = 0 \rceil\!\rceil \frown \lceil\!\lceil s.con.heater\_out = 1 \rceil\!\rceil \\
unit &= \lceil\!\lceil s.con.heater\_out = 0 \rceil\!\rceil \frown \lceil\!\lceil s.con.heater\_out = 1 \rceil\!\rceil
\end{aligned}
$$

For the sake of simplicity, we take the following lemma, *end_with_On*, as an example to demonstrate how our system supports inductive proofs. The lemma states that the heater is on at the end of all special intervals that are produced by applying the function *superCon*.

$$
\begin{aligned}
&end\_with\_On == \forall s : System \bullet \forall k : \mathbb{N} \bullet \\
&\qquad superCon(k, base, unit) \subseteq \lceil\!\lceil \mathrm{true} \rceil\!\rceil \frown \lceil\!\lceil s.con.heater\_out = 1 \rceil\!\rceil
\end{aligned}
$$

The proof script of the lemma including the proof commands invoked follows.

```
1:  ((skosimp)
2:   (induct "k")
3:   (("1" (expandsubset) (expand "superCon") (expandconcat -1)
4:     (assignconcat 1 "i1!1" "i2!1") (grind))
5:    ("2" (skosimp) (expandsubset) (expand "superCon")
6:     (expandconcat -2 1) (expandconcat -1)
7:     (assignconcat 1 "TwoTOneIvl(i1!1, i1!2)" "i2!2") (grind))))
```

Because *superCon* is defined recursively, induction is necessary. Using the PVS proof command `induct` at line 2, `(induct "k")`, directs the PVS prover to employ its induction scheme to the variable `k`. Two subproof goals are automatically generated: one denotes the base case, where $k = 0$ (at line 3); and the other corresponds to the induction case (at line 5).

We remark that the proof of the inductive case takes into account two connections among three sets of intervals as indicated by expanding the function `concat` twice at line 6. If the proof is carried out by hand, we have to examine sixteen situations with respect to the interval types from three sets. In contrast, this tedious work is automatically completed in our verification system by assigning appropriate values to the intervals at line 7 (where the function `TwoTOneIvl` creates a new interval from two adjacent intervals).

Experimental Results. We summarize our experimental results in Table II, which lists the names of lemmas associated with the steps of proof commands entered from users and the execution time (in seconds) of the PVS prover. The experiments are conducted on the SunOS 5.10 platform with PVS version 3.2. In total, we have proved twelve lemmas, some of which have been explained early, namely, *Decreasing*, *Safety*, *end_with_On*, and *Length*. We sketch the rest in the following, while their specifications and the complete proof scripts are available online [Chen 2008].

—*Increasing* describes the situation that the temperature increases in an interval, with the temperature not greater than the value 20 throughout the interval.

—*Safe1* claims that the temperature is always lower than or equal to the maximum value 40, and *Safe2* that the temperature is always greater than or equal to 20.

—*On_Off_On* (*Off_On_Off*) checks the length bound of the interval (1) in which the heater is off (on), and (2) which is between two consecutive intervals during which the heater is on (off).

—The lemma *invariant* shows that the special intervals highlighted in the previous subsection (using the function *superCon*) satisfy the *Length* requirement.

—The lemma *super_and_BT* indicates that any interval that starts with time point 0 and can be decomposed into k parts of subintervals according to the finite variability, can also be formed by executing *superCon* for k times.

—*Length_Cover* shows that any interval starting with 0 can be constructed in one of the three ways based on *superCon*.

During the proofs, mathematical reasoning, in particular the theories of integral calculus, is frequently involved (for lemmas *Decreasing*, *Increasing*, *On_Off_On* and *Off_On_Off*), since the temperature changes continuously in the control system. Moreover the induction mechanism plays an important role in tackling the analysis over infinite intervals (for lemmas *end_with_On*, *invariant* and *super_and_BT*). Last but not least, our developed supplementary rules for domain-specific features can ease proofs. For instance, the supplementary rule `mid_ivl_exi` defined in Section 3.4.1 has been applied to verify lemmas *Safe1* and *Safe2*.

In this section, we have developed a verification system for TIC based on PVS. Based on the encoding of TIC semantics in PVS, all TIC reasoning rules have been formalized and validated, and two subtle flaws have been discovered. A translator has been implemented in Java to translate TIC models to PVS specifications. We have successfully applied the verification system to validate the temperature control system as a hybrid application against timing requirements.

4. SUPPORTING MORE INTERVAL-BASED SPECIFICATION LANGUAGES

Up to now, we have developed a verification system based on PVS to support TIC verification. With the expressive power of TIC, the verification system is generic and can be applied to handle other interval-based specification languages. As introduced in Section 2.2, DC [Zhou and Hansen 2004] is another popular real-time specification language based on interval temporal logic. We will describe in this section how to systematically extend the system to support DC.

First, DC constructs are carefully modeled in TIC. Next, DC axioms and reasoning rules are formalized based on the encoding of TIC and are in turn validated in our verification system. We hence apply our expanded system to a typical case study in DC, and identify an improper proof step in its original presentation.

4.1 Modeling DC Semantics in TIC

In this section, we demonstrate the way to model DC semantics including special DC constructs in TIC. We also discuss how to resolve the differences between TIC and DC. The demonstration follows a bottom-up path, from basic DC constructs to complex ones. Corresponding PVS specifications are provided when needed.

4.1.1 *State Variables.* State variables in DC are functions from time to Boolean values $\{0, 1\}$, and each state variable is integrable in all intervals. We represent each state variable as an integral function whose range consists of values 0 and 1, that is, $\mathbb{T} \xrightarrow{\Box} \{0, 1\}$, where the symbol $\xrightarrow{\Box}$ of TIC [Fidge et al. 1998a] indicates the integrability of the declared function. The type of state variables is modeled in the following PVS type declaration, where the function `Integrable?` indicates whether the declared timed trace is integrable in any interval and the type `BTrace`, defined in Section 3.1.2, denotes that the timed trace is Boolean-valued.

```
DCState: TYPE = {bt: BTrace | forall (a, b: Time): Integrable?(a, b, bt)};
```

An important property of state variables is finite variability. This property indicates that any non-empty interval can be decomposed into a finite sequence of subintervals where a state variable is constant in each subinterval, which enables the application of mathematical induction. We show the PVS specifications of the property followed by detailed explanations (a similar approach was adopted by Skakkebaek [1994]).

```
k: var posnat; dcs1: var DCState; i: var II;
Cnst(dcs1)(i): bool = ALPHA(i) < OMEGA(i) and exists (x: real):
   forall (t: Time): t < OMEGA(i) and t > ALPHA(i) => dcs1(t) = x;
fvl(k)(dcs1)(i): RECURSIVE bool = IF k = 1 THEN Cnst(dcs1)(i)
   ELSE exists (m: Time): m < OMEGA(i) and m > ALPHA(i) and
     Cnst(dcs1)((OO, (ALPHA(i), m))) and fvl(k - 1)(dcs1)((OO, (m, OMEGA(i))))
   ENDIF MEASURE k;
fvr(k)(dcs1)(i): RECURSIVE bool = IF k = 1 THEN Cnst(dcs1)(i)
   ELSE exists (m: Time): m < OMEGA(i) and m > ALPHA(i) and
     fvr(k - 1)(dcs1)((OO, (ALPHA(i), m))) and Cnst(dcs1)((OO, (m, OMEGA(i))))
   ENDIF MEASURE k;
fv(k)(dcs1)(i): bool = fvl(k)(dcs1)(i) and fvr(k)(dcs1)(i);
DCState_is_FV: AXIOM forall i: ALPHA(i) = OMEGA(i) or exists k: fv(k)(dcs1)(i);
```

—`Cnst` is a function to check if a state variable `dcs1` is equal to a constant `x` at all time points in a non-empty interval. Note that the values of a state variable at the endpoints of an interval are ignored in DC. That is to say, we only need to constrain the values over *both-open* intervals in TIC.

—The function `fvl` *recursively* decomposes an interval into a sequence of *both-open* subintervals where two successive subintervals share one endpoint.

At each step of the decomposition, an interval `i` as a parameter is divided to two *both-open* intervals, where a state variable `dcs1` is constant in the first interval `(OO, (ALPHA(i),m))` and `fvl` holds in the second interval `(OO, (m, OMEGA(i)))`. The variable `k` in the PVS *measure function*, `MEASURE k`, is used as a counter to measure and terminate the decomposition. In other words, `fvl` unfolds an interval from the left-hand end with respect to a state variable. Similarly, we define the function `fvr` to recursively unfold an interval from the right-hand end.

Both `fvl` and `fvr` play an important role in enabling the application of a powerful proof method that is proof by induction. For instance, we can guide the PVS prover to invoke an induction scheme to the counter `k` from `fvl` or `fvr` by the instruction `(induct "k")`. Note that the construction of the constituent subintervals can exclude the endpoints of intervals (indicated by the interval type `OO`) due to the irrelevance of these endpoints when interpreting DC state variables.

- `DCState_is_FV` is an axiom that assumes: given a state variable, we can deduce that for any interval, either the interval is a point interval, or there exists a `k`-partition for some `k` such that functions `fvl` and `fvr` hold with respect to the state variable and the interval.

4.1.2 *State Expressions.* In DC, state expressions are produced by applying propositional logic operators to state variables. Semantically, state expressions are also functions from time to Boolean values $\{0, 1\}$. In the following we describe the conversion of three primitive logical operators including the negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$), in state expressions.

Let $S1$ and $S2$ be state expressions, and t be a time point.

- $(\neg\, S1)(t) = 1 - S1(t)$;
- $(S1 \wedge S2)(t) = S1(t) * S2(t)$;
- $(S1 \vee S2)(t) = 1 - (1 - S1(t) * (1 - S2(t)) = S1(t) + S2(t) - S1(t) * S2(t)$, since in the propositional logic, we have $(S1 \vee S2)(t) \Leftrightarrow (\neg\, (\neg\, S1 \wedge \neg\, S2))(t)$.

4.1.3 *Temporal Variables.* As temporal variables in DC are functions of intervals, they can be straightforwardly modeled as the interval operators of TIC. We demonstrate here the way to handle two predefined temporal variables in DC. One is the symbol ℓ, which denotes the length of an interval; this functionality is the same as δ in TIC (the corresponding PVS specification is defined in Section 3.1.2). Another is the duration operator $\int S$ where S is a state expression; this operator is supported in TIC as well (its PVS specification is shown in Section 3.1.3).

4.1.4 *Formulas.* Formulas of DC are evaluated with respect to intervals only. In other words, their semantics is independent of the interval endpoints. Hence DC formulas are considered to be a subset of predicates of TIC, where these predicates usually rely on both time points and intervals. As shown in the following PVS specifications based on the encoding of TIC in PVS in Section 3.1, a DC formula is a special predicate in which the values are independent of interval types (by the constraint at the second row) and time points within an interval (at the third row).

```
DCFormula?(p: TPred): bool =                        % i1, i2: II; t1, t2: Time
   forall i1, i2: ALPHA(i1)= ALPHA(i2) and OMEGA(i1) = OMEGA(i2) =>
      forall t1, t2: t_in_i(t1, i1) and t_in_i(t2, i2) => p(t1, i1) = p(t2, i2);
DCFormula: TYPE = {p: TPred | DCFormula?(pred)};
```

The particular operator in DC is the chop operator ($\frown$) used to specify that two formulas respectively hold in two subintervals that overlap at one time point. Note that DC is concerned with closed intervals. The chop operator may look like the concatenation operator ($\curvearrowright$) in TIC, since both can model sequential behavior at the interval level. However they are different: first of all, $\frown$ links DC formulas, which are indeed predicates of TIC, while $\curvearrowright$ concatenates sets of intervals; furthermore, there is no overlap and no gap between two connected intervals in TIC. These differences indicate that we cannot simply replace $\frown$ by $\curvearrowright$. For example, a point interval i, $\alpha(i) = \omega(i)$), may satisfy the DC formula $\phi \frown \psi$, although it is impossible for a point interval to be the result of a concatenation operation in TIC as any point interval cannot be divided into two non-overlapped intervals. Therefore, we define a function to represent the chop operator in TIC, where $\mathbb{I} \to \mathbb{B}$ indicates the type of DC formulas and the symbol $\mathbb{B}$ (where $\mathbb{B} ::=$ true | false) for a Boolean type.

$$
\begin{array}{|l}
\_ \frown \_ : (\mathbb{I} \to \mathbb{B}) \times (\mathbb{I} \to \mathbb{B}) \to (\mathbb{I} \to \mathbb{B}) \\
\hline
\forall \phi, \psi : \mathbb{I} \to \mathbb{B};\ i : \mathbb{I} \bullet (\phi \frown \psi)(i) \Leftrightarrow \\
\qquad \exists\, i1, i2 : \mathbb{I} \bullet \alpha(i) = \alpha(i1) \wedge \omega(i) = \omega(i2) \wedge \omega(i1) = \alpha(i2) \wedge \phi(i1) \wedge \psi(i2)
\end{array}
$$

This definition takes two predicates of TIC as arguments, and returns a predicate of TIC that holds in an interval if and only if there are two special subintervals in the interval and the passed predicates are true respectively in the subintervals. Specifically, these subintervals share one endpoint, and their other endpoints are equal to the endpoints of the interval. Note that we only constrain the values of the interval endpoints without taking into account the types of these endpoints. The PVS specification of the chop operator is given in the following.

```
dcf1, dcf2: var DCFormula;
DCChop(dcf1, dcf2)(t, i): bool = exists (i1, i2: II):
    ALPHA(i) = ALPHA(i1) and OMEGA(i) = OMEGA(i2) and OMEGA(i1) = ALPHA(i2) and
    Everywhere?(dcf1, i1) and Everywhere?(dcf2, i2);
```

Based on the encoding of $\frown$, we specify two frequently used DC operators, $\diamond$ and $\Box$, in TIC, and in turn in PVS, following the DC syntax construction style introduced in Section 2.2. The result of each operation is a DC formula as well. The function `TTRUE` used in the following PVS specifications always returns true in PVS, namely, `TTRUE(t, i): bool = true`.

```
<>(dcf1): DCFormula = DCChop(DCChop(TTRUE, dcf1), TTRUE);
[](dcf1): DCFormula = not(<>(not(dcf1)));
```

We have presented so far the way to formalize DC constructs using TIC and PVS. The differences between DC and TIC have been discussed and solutions have been proposed, such as supporting the chop operator and the finite variability of state variables. This encoding serves as a foundation for performing rigorous validation of DC axioms and reasoning rules as well as proofs of DC models in the following sections.

4.2 Validating DC Axioms and Reasoning Rules

In DC, reasoning rules are derived from the axioms of state durations. Checking the correctness of these axioms and reasoning rules is important when developing machine-assisted proof support for DC. Existing works [Skakkebaek 1994; Heilmann 1999; Rasmussen 2002] directly assume the validity of the axioms. Our approach differs from them in that we represent the axioms based on the encoding from the previous section and rigorously reason about their correctness using our verification system. The DC reasoning rules are in turn validated as well.

DC axioms and reasoning rules are required to be true in all closed intervals. They are modeled as predicates in TIC and hence their correctness can be verified with respect to intervals. We remark that intervals considered in TIC are classified into four basic types based on the inclusion/exclusion of interval endpoints. That is to say, the intervals involved in the validation of each axiom and rule are not just closed intervals. Nevertheless this method of interpretation is acceptable because DC formulas are independent of the interval types.

Mathematical analysis, especially mathematical laws of integral calculus, is important in the validation. The axiom **DCA5** from Section 2.2 is an example; the axiom captures the association between state durations and the chop operator. We formalize the axiom in the following TIC predicate and PVS specifications, where the symbol S represents a state expression and variables x and y nonnegative real numbers. Note that we take into account all interval types (indicated by $\llbracket\!\!(\ )\!\!\rrbracket$ and `AllS`) in the formalizations of the axiom.

$$DCA5 == \mathbb{I} = \llbracket\!\!((\int S = x) \frown (\int S = y) \Rightarrow \int S = x + y)\!\!\rrbracket$$

```
S: var DCState; x, y: var nnreal;
DC_DCA5: LEMMA fullset =
    AllS(DCChop(TICIntegral(S) = LIFT(x), TICIntegral(S) = LIFT(y))
         => TICIntegral(S) = LIFT(x + y));
```

The mathematical analysis of integration is necessary to check $DCA5$. For instance, one proof sequent in the reasoning process is shown in the following: two intervals, `il!1` and `ir!1`, are automatically generated as Skolem constants from the interval `x!2` according to the chop definition, and the constraints over the endpoints of these intervals are expressed by the antecedents indexed by −3, −4, and −5; the first two antecedents (indexes are −1 and −2) say that the accumulations of the state expression `S!1` in `il!1` and `ir!1` are equal to two nonnegative real numbers, `x!1` and `y!1`, respectively.

```
  {-1} Integral(ALPHA(ir!1), OMEGA(ir!1), S!1) = y!1
  {-2} Integral(ALPHA(il!1), OMEGA(il!1), S!1) = x!1
  [-3] ALPHA(x!2) = ALPHA(il!1)
  [-4] OMEGA(x!2) = OMEGA(ir!1)
  [-5] OMEGA(il!1) = ALPHA(ir!1)
  |-------
  {1}  Integral(ALPHA(x!2), OMEGA(x!2), S1!1) = x!1 + y!1
```

To accomplish this proof goal, we need to apply the lemma `Integral_split`, which captures the additivity of integration on intervals, where its contents are explicitly displayed as an antecedent.

```
  Rule? (lemma "Integral_split")
  this simplifies to:
  {-1} FORALL(a, b, c: Time, f: [Time -> real]):
         Integrable?(a, b,f) AND Integrable?(b, c, f)
         => Integrable?(a, c, f) AND
            Integral(a, b, f) + Integral(b, c, f) = Integral(a, c, f)
 ...
```

Besides the mathematical analysis of continuous dynamics, induction is also frequently used for complex DC formulas. For example, the reasoning rule **DC15** from Zhou and Hansen [2004] states that if the duration of a state expression S is positive in an interval then the interval can be chopped into three subintervals: the duration of S is zero in the first interval and S is true almost everywhere in the middle interval. Note that $\llbracket S \rrbracket$ is short for $\int S = \ell \wedge \ell > 0$ in DC.

DC15 $\int S > 0 \Rightarrow (\int S = 0) \frown \llbracket S \rrbracket \frown \text{true}$

The validation of this rule is sketched in the following. The formal reasoning process, which takes more than fifty proof commands is available online [Chen 2008]. First, an interval in which the duration of S is positive ($\int S > 0$) can be divided into k subintervals; S is constant in each subinterval, based on the axiom `DCState_is_FV` (from Section 4.1.1). Next, we invoke an induction proof over k, which is a positive natural number according to `DCState_is_FV`.

—The base case is $k = 1$, which indicates that S is constant in the interval. As the range of a state expression is Boolean-valued, the constant value of S is either 0 or 1. Moreover, the hypothesis $\int S > 0$ constrains the constant value to only be 1. Therefore by forming the first interval and the last interval as point intervals, the consequence holds since the duration in a point interval is always 0 and the predicate true also holds in point intervals.

—The inductive case assumes that the consequence is true when $k = n$, where n is an arbitrary positive natural number, and we need to prove that the consequence holds when $k = n + 1$. We apply the finite variability formalized in Section 4.1.1, in particular, the function `fvr`, which unfolds an interval from the *right-hand end*. When we expand the application `fvr(n + 1)` for an interval, the interval is decomposed into two consecutive *both-open* subintervals. The first subinterval satisfies the consequence due to the inductive hypothesis; S is constant in the second subinterval, which follows the first subinterval. Because the predicate true holds in the second subinterval, it is easy to show that the second subinterval has little influence on the validity of the consequence.

Therefore, the reasoning rule is proved by induction.

Currently, we have formalized and checked all DC axioms and the reasoning rules that are used in our example studies, as illustrated in the following section.

4.3 Handling DC Proofs

So far, we have modeled the DC constructs in Section 4.1, and validated the DC axioms and reasoning rules in Section 4.2. Based on this work, we can handle DC proofs using our verification system in a manner that closely follows the manual DC arguments. In this section, we demonstrate the usability of our approach with a typical DC case study, a gas burner [Zhou and Hansen 2004].

A gas burner is a software-embedded system in a safety-critical context. Let *Leak* be a state variable modeling the critical behavior, namely, $Leak : \mathbb{T} \rightarrow \{0, 1\}$, where the value 1 means that gas is leaking and the value 0 means no leaking. There are two design properties for the gas burner system. The first design property (also mentioned in Section 2.2) is that any leak should last for not longer than 1 time unit. The second design property is that the interval length between two consecutive leaks must be at least 30 time units. Both properties are modeled here in DC.

$$Des1 == \Box(\llbracket Leak \rrbracket \Rightarrow \ell \leq 1)$$
$$Des2 == \Box(\llbracket Leak \rrbracket \frown \llbracket \neg\, Leak \rrbracket \frown \llbracket Leak \rrbracket \Rightarrow \ell \geq 30)$$

A real-time requirement is that the proportion of the leaking time in an interval is always less than or equal to one-twentieth of the interval, provided the interval lasts at least 60 time units. This requirement is expressed here in DC, based on *Leak*.

$$GbReq == \ell \geq 60 \Rightarrow 20 * \int Leak \leq \ell$$

We remark that the *GbReq* requirement here is different from the *Length* requirement of the temperature control system in Section 3.3.2. *Length* is concerned with the intervals that start from time point 0. However, the intervals considered by *GbReq* are restricted by their lengths.

These properties and the requirement are DC formulas. They are modeled by the following PVS specifications, where the function pq denotes the abbreviation $\llbracket\ \rrbracket$ and other PVS symbols for DC constructs are defined in Section 4.1.

```
Leak: DCState;
Des1: DCFormula = [](pq(Leak) => LIFT(DELTA) < LIFT(1));
Des2: DCFormula = [](DCChop(DCChop(pq(Leak), pq(not(Leak))), pq(Leak))
                       => LIFT(DELTA) >= LIFT(30));
GbReq: DCFormula = LIFT(DELTA) >= LIFT(60)
                     => LIFT(20) * TICIntegral(Leak) <= LIFT(DELTA);
```

To prove the correctness of the design properties, we need to show that the formula *Des1* $\wedge$ *Des2* $\Rightarrow$ *GbReq* is *valid*. That is, the formula is true in all intervals. Hence, our proof goal is $\mathbb{I} = \llbracket Des1 \wedge Des2 \Rightarrow GbReq \rrbracket$, which is equivalently converted to the following goal $\llbracket Des1 \rrbracket \cap \llbracket Des2 \rrbracket \subseteq \llbracket GbReq \rrbracket$. That is to say, an interval in which both design properties hold is also the interval in which the requirement is true. This proof goal is expressed here by a PVS theorem named `ProofGoal`.

```
ProofGoal: theorem subset?(intersection(AllS(Des1), AllS(Des2)), AllS(GbReq));
```

Using our verification system, the proof process for this goal can strictly comply with the original process [Zhou and Hansen 2004] in terms of the order of applying DC reasoning rules and lemmas. These rules and lemmas are also validated. In the following, we informally describe important steps in the process associated with a simplified proof script.

```
("1"... (lemma "Math_PL1") ...
 (("1" ... (lemma "Lemma3_5") ...
   (("1" ... (lemma "Lemma3_6") ...
     (("1" ... (lemma "DC_DC8") ...
```

(1) Lemma `Math_PL1` captures the property that any interval, in which two design properties hold, can be partitioned into a sequence of n parts of intervals of size 30 time units followed by an interval whose size does not exceed 30 time units, where n is a natural number.

(2) From both design properties we can deduce that the duration of leaking does not exceed 1 time unit in any interval that lasts not longer than 30 time units. This result is modeled by the lemma `Lemma3_5`.

(3) Based on `Lemma3_5`, we can deduce that gas can be leaking for n time units during the first n intervals of size 30 (produced at the first step). This is specified by the lemma `Lemma3_6`.

(4) From `Lemma3_5`, we conclude that the leaking duration is less than 1 time unit in the interval that follows the first n intervals; we can thus apply DC reasoning rule **DC8** from Zhou and Hansen [2004] to accumulate the leaking time of the first n intervals and this interval.

After these steps, the proof can be automatically accomplished.

In the rigorous verification performed in our verification system, we have identified an improper proof step in the original proof [Zhou and Hansen 2004]. A proof obligation before applying **DC8** is $(\int Leak \leq n) \frown (\int Leak \leq 1) \Rightarrow (\int Leak \leq n+1)$, and the original proof adopts the axiom **DCA5** (mentioned in Section 2.2); however, it is obvious to see that **DCA5** cannot resolve the obligation. Instead, the appropriate reasoning rule is **DC8**, which easily discharges the proof obligation.

DC8 $(\int S \leq x) \frown (\int S \leq y) \Rightarrow \int S \leq x + y$

The gas burner is a typical DC case study to which our extended verification system has been applied in this section. We have presented the corresponding PVS specifications of the system design and real-time requirement. The DC reasoning rules and lemmas used in analyzing the gas burner have been formalized and checked. Thus the reasoning process in the verification system follows a manner similar to the manual DC arguments. We have also shown our discovery of the incorrect proof step in the original proof.

In this section, we have generalized the verification system to support other interval-based specification languages, in particular, DC. This enhancement has been accomplished by applying TIC to model DC. We have carefully encoded DC constructs in TIC, and further formalized and validated the DC axioms as well as reasoning rules. The resulting system enables users to carry out DC proofs along the lines of the hand proof without knowing the detailed encoding. Furthermore, the rigorous verification capability of the system elevates the confidence level of DC proofs, for example the improper proof step identified in the original manual DC arguments of the gas burner.

5. CONCLUSION

We developed a verification system to support formal interval-based specification languages, TIC and DC, which model real-time computing systems. The verification of these systems is difficult due to the analysis of continuous dynamics as well as the reasoning over arbitrary (infinite) time intervals. Our verification system supports mathematical analysis including integration, and is capable of handling infinite systems by induction.

The verification system is built upon PVS and encodes the TIC semantics and reasoning rules. It is generic enough to support DC by representing the DC constructs in TIC as well as the DC axioms and reasoning rules. The verification of TIC in our verification system is carried out directly at the interval level by applying the reasoning rules and supplementary rules that capture domain-specific features. Low level proof goals, for example the decision procedures on sets and linear arithmetic on real numbers, can be automatically discharged by the PVS powerful prover. Furthermore, we have developed 11 PVS proof strategies to save the users from needing to understand the detailed encoding of TIC in PVS. Likewise, DC proofs constructed in the verification system can closely follow the corresponding manual DC reasoning procedure.

The rigorous reasoning feature of the verification system can elevate the level of confidence in system designs. From our experimental study, two subtle flaws of TIC original reasoning rules (which are often used to verify safety requirements) have been discovered, and an improper proof step in the original manual DC arguments of a gas burner has also been identified.

In general, our approach to verification of TIC models is not fully automated. This is the price to be paid for 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 that can elevate the amount of automation (for example, when instantiating a time point in an interval during a proof, usually the proof can be successfully accomplished in one of three following ways in the instantiation: either one of the interval 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 mechanized proofs of TIC at a higher level. Moreover, another of our goals is implementing a translator to automatically transform DC models into TIC models.

ACKNOWLEDGMENTS

We thank Yuzhang Feng and Ian Hayes for their insightful discussion. We appreciate Anders P. Ravn and Chaochen Zhou for their help on Duration Calculus.

REFERENCES

Alur, R., Courcoubetis, C., Henzinger, T. A., and Ho, P.-H. 1992. Hybrid Automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*. Springer, 209–229.

Alur, R. and Dill, D. L. 1990. Automata for modeling real-time systems. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*. Springer, 322–335.

Alur, R. and Henzinger, T. A. 1991. Logics and models of real time: A survey. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*. Springer-Verlag, 74–106.

Butler, R. W. 2004. Formalization of the integral calculus in the PVS theorem prover. Tech. rep., NASA Langley Research Center, VA.

Cerone, A. 2001. Axiomatisation of an interval calculus for theorem proving. In *Electronic Notes in Theoretical Computer Science 42*.

Chakravorty, G. and Pandya, P. K. 2003. Digitizing interval duration logic. In *Proceedings of the 15th International Conference on Computer-Aided Verification*. Springer, 167–179.

Chen, C. 2008. A Verification System for interval-based specification languages. `http://www.comp.nus.edu.sg/~chenchun/verifysys`.

Chen, C., Dong, J. S., and Sun, J. 2007. Machine-assisted proof support for validation beyond Simulink. In *Proceedings of the 9th International Conference on Formal Engineering Methods*. Springer, 96–115.

Chen, C., Dong, J. S., and Sun, J. 2008. A verification system for timed interval calculus. In *Proceedings of the 30th International Conference on Software Engineering*. ACM, 271–280.

Cheng, B. H. C. and Atlee, J. M. 2007. Research directions in requirements engineering. In *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society, 285–303.

Clarke, E. M., Grumberg, O., and Long, D. E. 1994. Model checking and abstraction. *ACM Trans. Prog. Lang. Syst. 16*, 5, 1512–1542.

Dawson, J. E. and Goré, R. 2002. Machine-checking the timed interval calculus. In *Proceedings of the 15th Australian Joint Conference on Artificial Intelligence*. Springer-Verlag, 95–106.

Fidge, C. J., Hayes, I. J., and Mahony, B. P. 1998a. Defining differentiation and integration in Z. In *Proceedings of the 2nd International Conference on Formal Engineering Methods*. IEEE Computer Society, 64–73.

Fidge, C. J., Hayes, I. J., Martin, A. P., and Wabenhorst, A. 1998b. A set-theoretic model for real-time specification and reasoning. In *Proceedings of the 4th International Conference on Mathematics of Program Construction*. Springer, 188–206.

Gordon, M. J. C. and Melham, T. F., Eds. 1993. *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press.

Gravell, A. M. and Pratten, C. H. 1998. Embedding a formal notation: Experiences of automating the embedding of Z in the higher order logics of PVS and HOL. In *Proceedings of 11th International Conference on Theorem Proving in Higher Order Logics (supplementary proceedings)*. Springer, 73–84.

HEILMANN, S. T. 1999. Proof support for duration calculus. Ph.D. thesis, Department of Information Technology, Technical University of Denmark.

HENZINGER, T. A., HO, P.-H., AND WONG-TOI, H. 1997. HYTECH: A model checker for hybrid systems. In *Proceedings of the 9th International Conference on Computer Aided Verification*. Springer, 460–463.

HENZINGER, T. A. AND SIFAKIS, J. 2006. The embedded systems design challenge. In *Proceedings of the 14th International Symposium on Formal Methods*. Springer, 1–15.

LARSEN, K. G., PETTERSSON, P., AND YI, W. 1997. UPPAAL in a nutshell. *Int. J. Softw. Tools Tech. Transf. 1*, 1-2, 134–152.

MAHONY, B. P. AND HAYES, I. J. 1992. A case-study in timed refinement: A mine pump. *IEEE Trans. Softw. Engin. 18*, 9, 817–826.

MATTOLINI, R. AND NESI, P. 2001. An interval logic for real-time system specification. *IEEE Trans. Softw. Engin. 27*, 3, 208–227.

MOK, A. K., LEE, C.-G., WOO, H., AND KONANA, P. 2002. The monitoring of timing constraints on time intervals. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*. IEEE Computer Society, 191–200.

MOSER, L. E., MELLIAR-SMITH, P. M., RAMAKRISHNA, Y. S., KUTTY, G., AND DILLON, L. K. 1996. The real-time graphical interval logic toolset. In *Proceedings of the 8th International Conference on Computer Aided Verification*. 446–449.

MOSZKOWSKI, B. 1986. *Executing Temporal Logic Programs*. Cambridge University Press.

MUÑOZ, C., CARREÑO, V., AND DOWEK, G. 2006. Formal analysis of the operational concept for the Small Aircraft Transportation System. In *Rigorous Development of Complex Fault-Tolerant Systems*. Springer, 306–325.

MUÑOZ, C., CARREÑO, V., DOWEK, G., AND BUTLER, R. W. 2003. Formal verification of conflict detection algorithms. *Int. J. Softw. Tools Tech. Transf. 4*, 3, 371–380.

NICOLLIN, X., SIFAKIS, J., AND YOVINE, S. 1992. From ATP to timed graphs and hybrid systems. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*. Springer-Verlag, 549–572.

NIPKOW, T., PAULSON, L. C., AND WENZEL, M. 2002. Isabelle/HOL—A proof assistant for higher-order logic. Lecture Notes in Computer Science, vol. 2283. Springer.

OWRE, S., RUSHBY, J. M., AND SHANKAR, N. 1992. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction*. Springer, 748–752.

PAULSON, L. C. 1994. Isabelle: A generic theorem prover. Lecture Notes in Computer Science, vol. 828. Springer.

RASMUSSEN, T. M. 2002. Interval logic—proof theory and theorem proving. Ph.D. thesis, Technical University of Denmark.

RUSHBY, J. M. 2000. Theorem proving for verification. In *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*. Springer, 39–57.

SKAKKEBAEK, J. U. 1994. A verification assistant for a real-time logic. Ph.D. thesis, Department of Computer Science, Technical University of Denmark.

SKAKKEBÆK, J. U. AND SHANKAR, N. 1994. Towards a duration calculus proof assistant in PVS. In *Proceedings of the 3rd International Symposium Organized on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer, 660–679.

STRINGER-CALVERT, D. W. J., STEPNEY, S., AND WAND, I. 1997. Using PVS to prove a Z refinement: A case study. In *Proceedings of the 4th International Symposium of Formal Methods Europe*. Springer, 573–588.

THE MATHWORKS 2008. *Simulink® 7—Using Simulink*. The MathWorks.

TIWARI, A., SHANKAR, N., AND RUSHBY, J. M. 2003. Invisible formal methods for embedded control systems. *Proc. IEEE 91*, 1, 29–39.

WOODCOCK, J. AND DAVIES, J. 1996. *Using Z: Specification, Refinement and Proof*. Prentice-Hall, Inc., Upper Saddle River, NJ.

YU, Y., REN, S., AND FRIEDER, O. 2006. Prediction of timing constraint violation for real-time embedded systems with known transient hardware failure distribution model. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*. IEEE Computer Society, 454–466.

ZHOU, C. AND HANSEN, M. R. 2004. *Duration Calculus: A Formal Approach to Real-Time Systems*. Springer-Verlag.

ZHOU, C., HOARE, C. A. R., AND RAVN, A. P. 1991. A calculus of durations. *Inform. Proc. Lett. 40*, 5, 269–276.

ZHOU, C. AND LI, X. 1994. A mean value calculus of durations. In *A Classical Mind: Essays in Honour of C. A. R. Hoare*. Prentice-Hall International Ltd., 431–451.

ZHOU, C., RAVN, A. P., AND HANSEN, M. R. 1993. An extended duration calculus for hybrid real-time systems. In *Hybrid Systems*. Springer, 36–59.

Received October 2008; revised February 2009; accepted February 2009