Towards a Duration Calculus Proof Assistant in PVS *

From the Third International School and Symposium on "Formal Techniques in Real Time and Fault Tolerant Systems," 19–23 September 1994, Lubeck, Germany.
Springer-Verlag Lecture Notes in Computer Science Vol. 863, pp. 660–679.

Jens Ulrik Skakkebæk[1] and Natarajan Shankar[2]

[1] Department of Computer Science, Technical University of Denmark, Bldg. 344, DK-2800 Lyngby, Denmark. E-mail: jus@id.dtu.dk
[2] Computer Science Laboratory, SRI International, Menlo Park CA 94025, USA. E-mail: shankar@csl.sri.com

Abstract. The Duration Calculus (DC) is an interval temporal logic for reasoning about real-time systems. This paper describes a tool for constructing DC specifications and checking DC proofs. The *proof assistant* is implemented by encoding the semantics of DC within the higher-order logic of a general-purpose specification and verification environment called PVS. We develop a Gentzen style sequent proof system for DC which we show to be sound with respect to the semantic encoding. We exploit the similarity between the sequent calculus of PVS and the sequent calculus of DC to obtain a DC proof system where the proofs are carried out in PVS at the semantic level, but appear as proofs in the DC proof system. The resulting system, called PC/DC, retains the specification and verification capabilities of PVS within the framework of the Duration Calculus. We present an example of a DC proof whose PC/DC presentation closely follows the lines of the corresponding pencil-and-paper DC arguments. Our approach can be applied to specification logics other than DC.

1 Introduction

The Duration Calculus (DC) [24] was introduced to reason about safety properties of real-time systems. It is an extension of a dense-time version of Moszkowski's discrete-time Interval Temporal Logic (ITL) [12, 17]. DC was the first real-time formalism to introduce the concept of an integrating (duration) operator, which is convenient for reasoning about *intermittent* system behaviors. Examples of applications can be found in [14, 19, 20]. Since DC could be employed in safety-critical applications and has hitherto lacked any effective mechanical support, we have developed a specification front-end and a proof checker for DC.

* The work of the first author was partially funded by the Danish Research Academy, SRI International, and ESPRIT BRA 7071: Provably Correct Systems (ProCoS). The second author was partially funded by NSF Grant CCR-930044 and by the US Naval Research Laboratory under contract N00015-92-2177.

This mechanization has been carried out within the framework of the PVS specification and verification environment [18]. In this paper, we present the details of this mechanization and argue that our approach can be applied to specification logics other than DC.

As an illustration of an intermittent behaviour, consider a traffic light which alternates between *red* and *green* via *yellow*. The traffic light is located at the intersection of a highway with a small road, pointing along the highway. We would like to give the highway traffic priority over the traffic on the small road. That is, viewed in a sufficiently large time interval, say 100 seconds, the traffic light should be *green* for a fraction, say $\frac{1}{8}$, longer than it is *red*. Assuming that *red* and *green* are functions from dense time (say, the reals) to $\{0, 1\}$, we can specify this using the Riemann integral[3]

$$e - b \geq 100 \Rightarrow \frac{9}{8} \int_b^e red(t)dt \leq \int_b^e green(t)dt$$

in some observation interval from b to e. In DC, the same requirement is specified:

$$\textstyle\int 1 \geq 100 \Rightarrow \frac{9}{8} \displaystyle\int red \leq \int green$$

where we lift the Riemann integral by implicitly assuming an observation interval $[b, e]$, while noting that $\int_b^e 1 = e - b$.

DC is intended for the development of embedded, safety critical systems, where an abstract specification is developed in a series of refinements leading to concrete specifications of controllers, sensors, and actuators. Each new refinement must be proven to imply the previous step. In the traffic light example, for instance, a concrete implementation could be *red* for 10 seconds, *yellow* for 2.5 seconds (i.e., 5 seconds in total for each cycle), and *green* for 15 seconds. This implementation intuitively satisfies the above specification, since even in the worst-case scenario, within 100 seconds we would have 3 cycles of 30 seconds each and 10 extra seconds of *red*, giving a total of 40 seconds of *red* and 45 seconds of *green*. These values exactly satisfy the specification since $\frac{9}{8}40 = 45$. DC is used to carry out such arguments in a more formal setting.

As illustrated above, refinement proof obligations often involve arithmetic arguments on durations and (in)equalities. Presently, these proof obligations are discharged by pencil-and-paper reasoning. However, in the development of real-life, possibly safety critical systems it is hard to ensure correctness of each step in the reasoning and impossible to keep track of all the proof details.

It is clear that mechanized assistance is needed in order to achieve rigor and precision in the specification and verification of complex safety-critical systems. A mechanized *proof assistant* should support the writing of DC specifications in a convenient and conventional syntax. For instance, we would like to write the above specification as:

```
dur(1) >= 100 => 9/8*dur(red) <= dur(green)
```

[3] Assuming that *red* and *green* only have isolated points of discontinuity.

More importantly, mechanization should make it possible to construct proofs at the same conceptual level as informal mathematical thought. As a typical DC proof step, consider proving that if $\frac{9}{8}\int red \leq \int green$ holds on two adjacent sub-intervals, it also holds on the full interval. In DC, this is written:

$$\begin{array}{ll} (\frac{9}{8}\int red \leq \int green);(\frac{9}{8}\int red \leq \int green) & \{\mathit{Dur\text{-}Preserve}\} \\ \Rightarrow \frac{9}{8}\int red \leq \int green & \end{array}$$

where ; is the *chopping* operator that divides an interval into two contiguous sub-intervals, and *Dur-Preserve* is a general lemma with the precondition that the binary relation (here: $p(x,y) \mathrel{\widehat{=}} \frac{9}{8}x \leq y$) is *additive* in its arguments.[4] In a typical informal argument, such preconditions are "obviously" satisfied and the specifier often refrains from writing them explicitly. However, using these "obvious" steps often introduces errors in the reasoning. We would like the proof assistant to generate and, preferably, prove such preconditions automatically.

This paper presents the first steps towards a DC proof assistant meeting the above requirements. Rather than building a tool from scratch, we have chosen to implement this proof assistant using an existing general purpose theorem prover, the Prototype Verification System (PVS) [18]. PVS integrates arithmetic and logical decision procedures which are quite useful for mechanizing DC. Such decision procedures are essential for conducting high-level mathematical arguments since they alleviate the user of trivial proof obligations. For instance, in our proof assistant prototype, called PC/DC, the above proof obligation (in a corresponding sequent formulation) looks as follows:

```
{-1}   (9 / 8 * dur(red) <= dur(green));(9 / 8 * dur(red) <= dur(green))
     |-------
{1}   9 / 8 * dur(red) <= dur(green)
```

In the above sequent, there is one formula (numbered `-1`) above the sequent turnstile `|-------` and one formula (numbered `1`) below. The proof tool command `dur-preserve` trivially completes the main proof as expected but requires the user to prove additivity of the relation (using Skolem constants like `r1!1`)

```
[-1]   9 / 8 * r1!1 <= s1!1
[-2]   9 / 8 * r2!1 <= s2!1
  |-------
[1]   9 / 8 * r1!1 + 9 / 8 * r2!1 <= s1!1 + s2!1
```

which the decision procedures automatically complete.

PC/DC is obtained by carefully encoding the semantics of DC in the typed logic of PVS so that the mechanized support available for the PVS logic can be exploited for constructing DC proofs. The typical disadvantage of semantic encodings is that a user is required to understand the details of the encoding and often has to work directly with the encoded form of the encoded logic.

In the work presented here, we try to address this disadvantage by permitting the user to work directly with the syntax and proof rules of DC and not their

[4] That is: $p(x_1,y_1) \wedge p(x_2,y_2) \Rightarrow p(x_1+x_2, y_1+y_2)$.

encoding in PVS. We first develop a sequent calculus for DC. We then exploit the similarity between the sequent calculus of PVS and that of DC so that the underlying PVS proof is made to appear as a DC proof. As a result, the semantic encoding of DC in PVS is not explicitly visible to a user. That is, proofs constructed in PC/DC closely follow the lines of the corresponding pencil-and-paper DC arguments.

The basic idea of the encoding approach is to use PVS to define the semantics of DC terms and formulas with respect to explicit time intervals. Once the semantics has been encoded, we can prove the DC proof rules as lemmas in PVS. We then develop proof strategies in PVS so that these lemmas are applied within the framework of a sequent calculus for DC. At this point, the time intervals are still explicit and the encoding is visible. We have therefore developed a parser/unparser for the DC subset of PVS so that specifications and proofs can be developed in the syntax of DC. A more detailed description of the encoding and the examples is found in [22].

In general, there are a variety of logics for reasoning about safety-critical systems including other flavours of modal, Hoare, and temporal logics for which it would be interesting to supply mechanized proof assistance. The techniques presented here most likely generalize to also cover these logics. In the same style as for DC, these logics can be encoded so as to preserve the original syntax and proof theory but without sacrificing the mechanization available in PVS. Several systems other than PVS would also be useful for this purpose, but we have chosen PVS because it offers a convenient combination of an expressive specification notation and effective mechanization.

In Section 2 we use the encoding of DC in PVS to present the syntax and semantics of DC. A Gentzen style proof system for DC is presented in Section 3. The PC/DC specification in Section 4 of the classical gas burner [24] together with a sample PC/DC proof illustrate the use of the proof assistant. The approach to hiding the encoding is described in Section 5. Related work is discussed in Section 6.

2 The Duration Calculus in PVS

In this section, we define the syntax of DC and the PC/DC input language, and the DC semantics in the form that it is encoded in PVS. The background information needed to follow the PVS encoding is provided when needed.

The primitive DC concept of a *state* is essentially that of a binary signal over time. The *duration* operator of DC returns the accumulated duration for which a given state has been *high* over the implicit interval. DC *terms* are constructed by applying arithmetic operations to durations. Atomic DC *formulas* are constructed by applying arithmetic relations to DC terms. DC *formulas* can be constructed using the usual propositional connectives and quantification. There is a chopping operator ' ; ' so that $A;B$ holds of an interval if A holds of an initial sub-interval, and B holds of the remainder of the interval. In Section 2.2, the encoding of the DC semantics in PVS is obtained in a straightforward manner by making the time parameter explicit for DC states and making the interval explicit in the case of DC formulas. In this paper, the duration operator is not defined semantically since that would involve formalizing the Riemann integral in PVS. It would be interesting to formalize the duration operator this way, but for the time being, it is defined by means of axioms.

2.1 DC and PC/DC Syntax

The DC syntax consists of *states*, *terms*, and *formulas*. Intuitively, the meaning of a state is a function from time to $\{0, 1\}$. Terms and formulas are functions from intervals to the reals and the Booleans, respectively.

Let S denote a member of the syntactic category of Boolean *atomic states*. A *state* s is defined recursively:

$$s \ ::= S \mid \neg s \mid s_1 \vee s_2$$

where $\neg$ and $\vee$ are propositional negation and disjunction. The corresponding PC/DC syntax is:

$$s \ ::= S \mid \texttt{not} \ \ s \mid s_1 \ \texttt{\textbackslash/} \ s_2$$

As usual, we can define the remaining propositional operators $\wedge$, $\vee$, and $\Leftrightarrow$ (`/\`, `=>`, and `<=>` in PC/DC) in the standard way. In PC/DC, an atomic state S is introduced as `S: State`.

Let f denote a function symbol and let x denote a *variable*. A *term* t is defined as:

$$t \ ::= x \mid \int s \mid f(t_1, \ldots, t_n)$$

where $\int s$ (`dur` s in PC/DC) is a *duration*. Intuitively, a duration is an interval measure, i.e., a function from intervals to the non-negative reals. Although f may be any function of the right type, we most often use the classical arithmetic operators: $+$, $-$, $*$, and $/$ (with their obvious interpretations).

Let p be a *relation symbol*. A *formula* A is then[5]

$$A \ ::= p(t_1, \ldots, t_n) \mid \neg A \mid A_1 \vee A_2 \mid A_1 ; A_2 \mid (\forall x : A)$$

and in PC/DC:

$$A \ ::= p(t_1, \ldots, t_n) \mid \texttt{not} \ \ A \mid A_1 \ \texttt{\textbackslash/} \ A_2 \mid A_1 \ \texttt{;} \ A_2 \mid (\texttt{FORALL} \ x \ \texttt{:} \ A)$$

The '$;$' symbol denotes the *chopping* operator which is an interval operator in DC. Intuitively, $A;B$ holds of an interval if it can be chopped into two contiguous intervals where A holds of the first interval and B holds of the second. The $\neg$ and $\vee$ operators are the usual propositional connectives. As before, we can define the remaining propositional operators $\wedge$, $\Rightarrow$, and $\Leftrightarrow$ (`/\`, `=>`, and `<=>` in PC/DC) in the standard way. We assume $\neg$ to bind stronger than $;$ which again binds stronger than $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$.

[5] Note, that we overload the propositional symbols: They are used both to combine states and formulas.

Often used relations are the classical comparison operators $<$, $>$, $\leq$, and $\geq$, which are written `<`, `>`, `<=`, and `>=` in PC/DC. Useful shorthand operators are:

ℓ	$\mathrel{\widehat{=}} \int 1$	Length
$\lceil s \rceil$	$\mathrel{\widehat{=}} \int s = \ell \wedge \ell > 0$	s holds almost everywhere on a non-point interval
$\lceil\rceil$	$\mathrel{\widehat{=}} \ell = 0$	Point interval

which in PC/DC are written `len`, `'s'`, and `point`.

The modal operators $\Diamond$ and $\Box$ (`<>` and `[]`) can easily be defined in terms of the (associative) chop operator: $\Diamond A \mathrel{\widehat{=}} \mathit{true} \,;\, A \,;\, \mathit{true}$ expresses that there exists a sub-interval in which A holds, and $\Box A \mathrel{\widehat{=}} \neg\Diamond\neg A$ expresses that A must hold for all sub-intervals.

2.2 Semantics

We present the semantics of DC by its encoding in PVS.

Domains. The declarations of DC type domains use the built-in PVS domains: the reals (written `real`), the integers (`integer`), the natural numbers (`nat`), and the Booleans (`bool`). They also use the PVS notion of a predicate subtype which is a type definition construct similar to the comprehension principle of set theory.

DC formulas are evaluated on a dense time domain; we have chosen the reals $\mathbb{R}$. The time domain is declared as a type `Time` in PVS:

```
Time : TYPE = real
```

DC specifications typically also make use of the non-negative and non-zero subdomains of `Time`. These are specified by means of the predicate subtype construction in PVS as:

```
NonNegTime : TYPE = {x:Time|x>=0} CONTAINING 1
NonZeroTime : TYPE = {x:Time|x/=0} CONTAINING 1
```

where the `CONTAINING` construct is used to demonstrate that this subtype is nonempty[6].

DC also employs the domain of finite intervals over `Time`:

```
Interval: TYPE = {(b,e):[Time,Time]|b<=e} CONTAINING (1,1)
```

States. Semantically, a state represents a function from time to the set $\{0,1\}$. This is defined in PVS (using the Booleans instead of $\{0,1\}$) as:

```
State : TYPE = [Time -> bool]
```

We require that for any atomic state, the associated function must be *finitely variable*, i.e., have only a finite number of discontinuity points in any interval. This assumption is encoded in PVS and is necessary in the later induction rules.

[6] 1 is provided as an example of an element of the type. If the `CONTAINING` construct is omitted, PVS requires the user to show non-emptiness of the type.

Functions in PVS are typically declared in the form:

operator(*formals*): *type* = *definition*

The meaning of the propositional operators and the truth constants of DC can then be given in terms of the corresponding propositional operators in PVS. Let `s`, `s1`, and `s2` be states and let `r` range over `Time`. Also assume the built-in PVS Boolean conjunction, disjunction and negation: `AND`, `OR`, and `NOT`, and let `tt` and `ff` be the constant true and constant false functions from `Time` to `bool`:

```
/\ (s1,s2)(r) : bool = s1(r) AND s2(r)
\/ (s1,s2)(r) : bool = s1(r) OR s2(r)
not (s)(r) : bool = NOT s(r)

=>(s1,s2) : State = (not s1 \/ s2)
<=>(s1,s2) : State = (s1 => s2) /\ (s2 => s1)

tt(r) : bool = true
ff(r) : bool = false
```

Note that the operators `/\`, `\/`, `=>`, and `<=>` can be used as infix operators.

Terms. A DC term is a function from intervals to reals:

```
ITime : TYPE = [Interval -> Time]
```

In the classical presentation of DC, a *duration* of a state s, $\int s$, evaluated on an interval $[b, e]$ is the integral[7] of the step function f_s associated with s: $\int_b^e f_s(t)dt$. Instead of formalising the integral within PVS, we declare a duration as an uninterpreted function:

```
dur(s) : INonNegTime
```

where `s` is a variable of type `State` and `INonNegTime` denotes the type of functions from `Interval` to the non-negative numbers. We axiomatize the behaviour of `dur` using a sequent formulation of the Hilbert style axioms (see below). This axiomatization of the duration operator has previously been shown to be complete [13].

Let `t1` and `t2` be variables of type `ITime`, `nt2` range over functions from `Interval` to `NonZeroTime` (denoted `NZITime`), and let `i` be of type `Interval`. The usual *arithmetic* operators are defined as:

```
*(t1,t2)(i): Time = t1(i) * t2(i)
/(t1,nt2)(i): Time = t1(i) / nt2(i)
+(t1,t2)(i): Time = t1(i) + t2(i)
-(t1,t2)(i): Time = t1(i) - t2(i)
```

where the encoded DC operators on the left overload the PVS operators on the right.

[7] We use the Riemann integral which is the familiar notion of integration and applies to functions that are continuous over the interval except for a finite number of discontinuity points.

In the traditional presentations of DC, a *valuation* $\mathcal{V}$ associates a real value with each variable x. In PVS, we refrain from introducing the syntactic category of variables. Instead we use the variables of PVS which together with the quantifiers of PVS are unparsed to look as DC variables and quantifiers.

Often we use numerical constants in terms, e.g., $\frac{9}{8}$ in the term $\frac{9}{8}\int red$. However, such constants are really constant *functions* of intervals. For instance, the interval function $\frac{9}{8}$ is the function that for every interval returns the real value $\frac{9}{8}$. We would like to exploit the decision procedures for **real** on terms of type `ITime`. For this, we introduce a *coercion* function

```
n(r:Time)(i:Interval) : Time = r
```

as a mapping that lifts reals to constant functions of intervals. The encoded version of the above term is therefore `n(9/8) * dur(red)`, although we strip away the coercion function in the output from PC/DC. At present, in PC/DC real constants must explicitly be coerced as indicated[8].

Formulas. Formulas are functions from `Interval` to `bool`:

```
IForm : TYPE = [Interval -> bool]
```

The comparison operators are defined as expected:

```
>(t1,t2)(i): bool = t1(i) > t2(i)
<(t1,t2)(i): bool = t1(i) < t2(i)
>=(t1,t2)(i): bool = t1(i) >= t2(i)
<=(t1,t2)(i): bool = t1(i) <= t2(i)

=(At1,At2)(i) : bool = (At1(i) = At2(i))
```

where `t1` and `t2` are terms of type `ITime`, `nt2` is of type `NZITime`, and `i` is of type `Interval`. `At1` and `At2` are functions of type `[Interval -> t]` and `t` is any type. The operators are overloaded to operate on arguments of type `NZITime` as well.

Let `i` be a variable of type `Interval` and let `A` and `B` be variables of type `IForm`. The meaning of a formula is:

```
/\(A,B)(i): bool = A(i) AND B(i)
\/(A,B)(i): bool = A(i) OR B(i)
not (A)(i): bool = NOT A(i)

^(A,B)(i) : bool = (EXISTS (m:Time) : proj_1(i)<=m AND m<=proj_2(i) AND
                                      A(proj_1(i),m) AND B(m,proj_2(i)))

fall(L:[t -> IForm])(i:Interval) : bool = (FORALL (r: t): L(r)(i))
```

where `proj_1(b,e)` returns `b`, and `proj_2(b,e)` returns `e` for an interval `(b,e)`. The operator `^` is the encoded form of the chopping operator and `fall` is the encoded form of the universal quantifier. `L` is a function of the type `t` indicated by the quantified variable.

[8] Version 2.0 of PVS relieves the user of this by introducing *implicit* type coercion.

Validity. Given an assignment of values to variables and functions to states, a formula `A` constructed using the above operators is *satisfied* by an interval `i`, iff `A(i)` is true. A formula `A` is *satisfiable* iff there exists an interval `i` such that `i` satisfies `A` for some assignment of values to variables and functions to states. A formula `A` is *valid* iff for any interval `i` and any assignment of values to variables and functions to states, `A(i)` is true.

3 A Sequent Style Proof System for DC

The PC/DC tool is based on the Gentzen style proof system [7] for DC shown below. The proof system is very similar to the basic system presented in [6], which was developed in parallel with our work. All of the proof rules, except the axioms for durations, are proved in PVS using the semantic encoding.

Let A, B, C, and D range over Duration Calculus formulas. A formula or a term is *duration free* if it does not contain any symbols of the form $\int s$ for any state s. A *chop free* formula is a formula not containing the ; operator (and hence, neither the $\Diamond$ nor the $\Box$ operator).

A sequent in the DC proof system is an ordered pair (Γ, Δ) of sets of formulas and is written $\Gamma \vdash \Delta$. Here the formulas in Γ are called the *antecedent* formulas, and those in Δ are called the *consequent* formulas. We write $\Gamma \vdash$ for the sequent $(\Gamma, \emptyset)$ and $\vdash \Delta$ to represent the sequent $(\emptyset, \Delta)$. Also, Γ, A and A, Δ denote the sets $\Gamma \cup \{A\}$ and $\Delta \cup \{A\}$ respectively.

Given an assignment of values to variables and functions to states, a sequent $\Gamma \vdash \Delta$ is *true at i* iff i satisfies B for some $B \in \Delta$ whenever i satisfies A for all $A \in \Gamma$. A sequent $\Gamma \vdash \Delta$ is *valid* (written $\models \Gamma \vdash \Delta$) iff $\Gamma \vdash \Delta$ is true at i for any interval i and any assignment of values to variables and functions to states. This is encoded in PVS as:

```
seq(A,B) : bool = (FORALL (i:Interval): (A(i) IMPLIES B(i)))
```

where `A` and `B` ranges over `IForm` and `seq` is the encoded form of $\vdash$.

The DC proof system is given by the *inference rules* shown below. An *inference rule* is of the form:

$$\frac{\Gamma_1 \vdash \Delta_1 \ \ldots \ \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}\ \textbf{name}$$

where the (zero or more) upper sequents are called *premises* and the lower sequent is called the *conclusion*. In PVS, an inference rule with two premises is encoded using the predicate `rule`:

```
rule(seq(Gamma1,Delta1),seq(Gamma2,Delta2),
   %------------------------------------
                 seq(Gamma,Delta))
```

where the dotted line preceded by `%` is a simple comment. Given Boolean variables `X`, `Y`, and `Z`, the definition of `rule` is:

```
rule(X,Y,Z): bool = X AND Y IMPLIES Z
```

An *axiom* is an inference rule with an empty set of premises.

A *proof* is a tree of sequents whose leaves are axioms and where each sequent follows from its offspring sequent(s) by one of the inference rules. A formula A is *provable* in the proof system below iff there exists a proof tree with $\vdash A$ as the root sequent.

We employ the inference rules shown below. The *basic* inference rules axiomatize the calculus. The *derived* rules are defined for practical convenience and can be proven using the basic inference rules. Note that unless otherwise mentioned, all the rules have in PVS been proved correct with respect to the semantic definitions.

3.1 Basic Inference Rules

In this presentation we omit the structural rules, the propositional rules, the cut rule, and the quantifier rules since these are conventional with any classical first-order, predicate sequent calculus. We only present some rules that are particular to the Duration Calculus.

Equality. The rules for equality shown below also apply to equivalences between formulas. Let therefore an *expression* denote either a term or a formula.

An expression e is said to be *free* in a formula A if there are no free variable occurrences in e that appear bound in A. Let $\Gamma[e]$ (respectively, $\Delta[e]$) denote a sequence of formulas with one or more *free* occurrences of an expression e in the formulas of Γ (resp. Δ).

The following rules define top-level replacement:

$$\frac{\Gamma[b]\,,\, a=b \,\vdash\, \Delta}{\Gamma[a]\,,\, a=b \,\vdash\, \Delta}\ \textbf{Repl}\vdash \qquad\qquad \frac{\Gamma\,,\, a=b \,\vdash\, \Delta[b]}{\Gamma\,,\, a=b \,\vdash\, \Delta[a]}\ \vdash\textbf{Repl}$$

where either a and b are duration free or all instances of e in $\Gamma[e]$ (resp. $\Delta[e]$) only occur in chop free subformulas.

Apart from top-level replacement, also unconditional substitution is permitted when $a = b$ is true on every sub-interval:

$$\frac{\Gamma[b]\,,\, \Box(a=b) \,\vdash\, \Delta}{\Gamma[a]\,,\, \Box(a=b) \,\vdash\, \Delta}\ \textbf{Subst}\vdash \qquad\qquad \frac{\Gamma\,,\, \Box(a=b) \,\vdash\, \Delta[b]}{\Gamma\,,\, \Box(a=b) \,\vdash\, \Delta[a]}\ \vdash\textbf{Subst}$$

Rules for Durations. The four axioms for $\int$ presented below are treated as axioms in the PVS encoding and are not derived from the semantics. They are a straightforward translation of the original Hilbert style DC axioms:

$$\frac{}{\vdash\ \int 0 = 0}\ \textbf{Ax1} \qquad\qquad \frac{}{\vdash\ \int s \geq 0}\ \textbf{Ax2}$$

$$\frac{}{\vdash\ \int s_1 + \int s_2 = \int(s_1 \wedge s_2) + \int(s_1 \vee s_2)}\ \textbf{Ax3}$$

$$\frac{}{\vdash\ (\int s = d_1);(\int s = d_2) \Leftrightarrow (\int s = d_1 + d_2)}\ \textbf{Ax4}$$

where $d_1, d_2 \geq 0$ and s, s_1, and s_2 are states.

Rules for Chop. Below we define a number of inference rules for proving properties of chopped formulas. The chop operator is monotone if from $\Box(A \Rightarrow C)$ and $\Box(B \Rightarrow D)$ we can show $A;B \Rightarrow C;D$. The chop operator is also associative, and distributes over disjunction:

$$\frac{\Gamma \vdash \Box(A \Rightarrow C)\,,\,\Delta \qquad \Gamma \vdash \Box(B \Rightarrow D)\,,\,\Delta}{\Gamma\,,\,A;B \vdash C;D\,,\,\Delta}\ \vdash;$$

$$\frac{}{\vdash (A;B);C \;\Leftrightarrow\; A;(B;C)}\ \textbf{assoc} \qquad\qquad \frac{}{\vdash (A \vee B);C \;\Leftrightarrow\; (A;C \vee B;C)}\ \vee;$$

The distributive rule has a symmetrical rule $(;\vee)$, which we omit here.

The following rules relate existential quantification with the chop operator and describe the propagation of duration free formulas in and out of the scope of the chop operator (again omitting the symmetrical rules):

$$\frac{}{\vdash (\exists x : A);B \Leftrightarrow (\exists x : A;B)}\ \exists; \qquad\qquad \frac{}{\vdash (A \wedge C);B \;\Leftrightarrow\; C \wedge A;B}\ \textbf{cf};$$

where x does not occur free in B and C is duration free.

The following rules assert that *false* is the zero element with respect to the chop operator and $\lceil\,\rceil$ the identity element:

$$\frac{}{false;A \vdash}\ false;\vdash \qquad\qquad \frac{}{\vdash A;\lceil\,\rceil \;\Leftrightarrow\; A}\ ;\lceil\,\rceil$$

As before, we leave out the symmetric rules.

Induction. The condition of finite variability of states allows for induction over behaviors of states: Given a state P, initially and finally in a non-point interval either $\lceil P \rceil$ or $\lceil \neg P \rceil$ will hold.

We exploit this property in a pair of induction principles. Informally, to induct in DC is to prove an invariant over the variation of a state. We do this by showing that (1) the invariant holds for a point interval, and (2) if the invariant holds in the current interval, it must also hold on an extension of the interval, for which either $\lceil\,\rceil$, $\lceil P \rceil$ or $\lceil \neg P \rceil$ holds.

Formally, let X be a formula letter describing a property that holds on the current interval and let $R(X)$ be an invariant, i.e., an induction formula in which X occurs. Pick a state P over which to induct. One induction rule is written (omitting the symmetrical rule):

$$\frac{\vdash R(\lceil\,\rceil) \qquad \Gamma \vdash \Box(\Box R(X) \Rightarrow R(X \vee \lceil P \rceil ; X \vee \lceil \neg P \rceil ; X))\,,\,\Delta}{\Gamma \vdash \Box R(true)\,,\,\Delta}\ \textbf{LInd}$$

3.2 Derived Rules

The rules shown below can be derived from the basic rules above. These rules are quite handy in doing proofs and have also been proved in PVS.

Rules for Modal Operators. The following inference rules are characteristic of S4 modal systems [23]. Let $[\Gamma]$ be the result of removing all those formulas in Γ that do not have $\Box$ as the outermost operator, and let $<\Delta>$ be the result of removing all those formulas in Δ that do not have $\Diamond$ as the outermost operator:

$$\frac{[\Gamma]\,,\, A \vdash <\Delta>}{\Gamma\,,\, \Diamond A \,\vdash\, \Delta}\;\Diamond\vdash \qquad\qquad \frac{\Gamma \,\vdash\, A\,,\, \Delta}{\Gamma \,\vdash\, \Diamond A\,,\, \Delta}\;\vdash\Diamond$$

$$\frac{\Gamma\,,\, A \,\vdash\, \Delta}{\Gamma\,,\, \Box A \,\vdash\, \Delta}\;\Box\vdash \qquad\qquad \frac{[\Gamma] \,\vdash\, A\,,\; <\Delta>}{\Gamma \,\vdash\, \Box A\,,\, \Delta}\;\vdash\Box$$

Duration Preservation. Often, identical formulas on adjacent sub-intervals can be shown to hold on the full interval. In fact, for any formula preserving the relationship between durations on intervals, if the formula holds on two adjacent sub-intervals, it also holds on the full interval.

Let $\otimes$ be an additive relation: $r_1 \otimes s_1 \wedge r_2 \otimes s_2 \Rightarrow (r_1 + r_2) \otimes (s_1 + s_2)$ for any $r_1, s_1, r_2, s_2 \in \mathbb{R}$ (e.g., $\otimes$ could be $<$). The inference rule is:

$$\frac{\Gamma \,\vdash\, (\int p \otimes \int q);(\int p \otimes \int q)\,,\, \Delta}{\Gamma \,\vdash\, (\int p \otimes \int q)\,,\, \Delta}\;\textbf{DurPres}$$

where we have lifted $\otimes$ from being a binary predicate on the reals to a DC predicate on functions from intervals to reals.

3.3 Soundness and Completeness

The soundness of the proof system with respect to the semantics has been shown using the semantic encoding in PVS. This is done by showing that all the axioms are valid and that all the inference rules yield valid DC conclusions when the premises to the rules are valid. We will not repeat the proofs here.

The above proof system for DC has not been shown complete with respect to the given semantics. That is, there might be semantically valid formulas that cannot be proved using the above proof system. Instead, validity of such formulas is shown directly in the semantic encoding. Relative completeness has been shown of the duration axioms extending an ITL Hilbert style proof system [13].

4 Example: Simple Gas Burner

We illustrate the use of PC/DC by the example of a simple gas burner [24]. Note that the original proof contains an error which was later corrected [25].

Consider a possibly leaking gas burner. A safety requirement for the system is that gas only leaks from the burner for a twentieth of the time when monitored for at least one minute.

Let *Leak* be a state assertion indicating the system being in a state where gas is leaking from the burner. The requirement can be expressed in DC as:

$$Req \mathrel{\widehat{=}} \ell \geq 60 \Rightarrow Safe$$

where $Safe \mathrel{\widehat{=}} 20\int Leak \leq \ell$.

Assume that a design of the control system assures (1) that gas is turned off at most one second after gas has started leaking, and (2) that gas is not turned on until 30 seconds have elapsed since the gas was last leaking. These decisions can be formulated as:

$$Des_1 \mathrel{\widehat{=}} \Box(\lceil Leak \rceil \Rightarrow \ell \leq 1)$$

$$Des_2 \mathrel{\widehat{=}} \Box((\Diamond\lceil Leak \rceil); (\Diamond\lceil \neg Leak \rceil); (\Diamond\lceil Leak \rceil) \Rightarrow \ell \geq 30)$$

The objective is to prove that the design satisfies the requirement:

$$Des_1 \wedge Des_2 \Rightarrow Req$$

This is shown by proving the invariant $Triple$ from the design:

$$Des_1 \wedge Des_2 \Rightarrow Triple$$

where $Triple \mathrel{\widehat{=}} T \vee (T; \lceil Leak \rceil) \vee (T; \lceil Leak \rceil; \lceil \neg Leak \rceil)$ and $T \mathrel{\widehat{=}} 30\int Leak \leq \ell$, and then showing that:

$$(Des_1 \wedge Triple \wedge \ell \geq 60) \Rightarrow Safe$$

The above lemmas and definitions are specified as follows in PC/DC:

```
dcgasburner: DCTHEORY
BEGIN
  IMPORTING dcp_thms

  %%% DEFINITIONS %%%

  Leak: State

  Safe: IForm = n(20)*dur(Leak) <= len
  Req: IForm = len >= n(60) => Safe

  Des_1: IForm = []('Leak' => len <= n(1))
  Des_2: IForm = []((<>'Leak');(<>'not Leak');(<>'Leak') => len >= n(30))

  T: IForm = n(30)*dur(Leak) <= len
  Triple: IForm = T \/ (T;'Leak') \/ (T;'Leak';'not Leak')

  %%% LEMMAS %%%

  lem1: LEMMA Des_1 /\ Des_2 => Triple
  lem2: LEMMA Des_1 /\ Triple /\ (len >= n(60)) => Safe

  Gasburner: LEMMA (Des_1 /\ Des_2 /\ (len >= n(60))) => Safe
END dcgasburner
```

As in PVS, axioms, theorems and lemmas are in PC/DC grouped together in *theories*. A theory is described by a name followed by the keyword **DCTHEORY** (instead of the PVS **THEORY**) and ends with the keyword **END** followed by the name. The **IMPORTING** clause in the figure refers to a previously defined theory **dcp_thms** of useful theorems. Next, a state **Leak** is declared. The declarations are followed by a number of shorthand definitions. These are in turn followed by the lemmas that are used in the proof. Note that they closely follow the lemmas found in the original specification. `Gasburner` is the final lemma which contains the proof obligation. The other lemmas are used in this proof. In general, only constants, variables and lemmas that are declared before a lemma can be used in the proof thereof.

The look and feel of the PC/DC prover is identical to the PVS prover. All system and prover commands from PVS can be still be used. Extra prover commands have been added to PVS to customize it to the needs of doing DC proofs. These commands include commands for applying most of the specific DC proof rules.

A lemma of the gas burner proof is to show that $\vdash T;T \Rightarrow T$. The proof of this corresponds very closely to the proof the traffic light example:

$$\begin{array}{ll} \quad T;T & \textit{defs} \\ \Rightarrow (30\int Leak \leq \int 1);(30\int Leak \leq \int 1) & \textit{Dur-Preserve} \\ \Rightarrow (30\int Leak \leq \int 1) & \textit{defs} \\ \Rightarrow T & \end{array}$$

As before, we refrain from writing that the binary predicate $30x \leq y$ is additive in its arguments. The PC/DC proof reflects the same level of reasoning (where a user has typed in the prover commands following the `Rule?` prompts):

```
Preserve :

  |-------
{1}   (T;T => T)

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
Preserve :

{-1}   T;T
  |-------
{1}   T

Rule? (apply (then* (expand "T") (expand "len")))
Applying
   (THEN* (EXPAND "T") (EXPAND "len")),
this simplifies to:
Preserve :
```

```
{-1}   (30 * dur(Leak) <= dur(tt));(30 * dur(Leak) <= dur(tt))
  |-------
{1}   30 * dur(Leak) <= dur(tt)

Rule? (dur-preserve)
Using duration preservation,
this simplifies to:
Preserve :

{-1}   30 * r1!1 <= s1!1
{-2}   30 * r2!1 <= s2!1
  |-------
{1}   30 * r1!1 + 30 * r2!1 <= s1!1 + s2!1
{2}   30 * dur(Leak) <= dur(tt)

Rule? (assert)
Invoking decision procedures,
Q.E.D.
```

After expanding the definitions, only one PC/DC command is needed to prove the lemma and the decision procedures take care of the "obvious" details.

The commands `flatten`, `assert` and `expand` are basic PVS prover commands. The `flatten` command breaks the implication and `assert` applies the built-in decision procedures. `expand` expands definitions into their bodies. In the example, `T` is expanded out into `30 * dur(Leak) <= len` and `len` is then again expanded out into its definition `dur(tt)`.

The commands `then*` and `apply` invoke PVS proof commands. The `then*` command takes a list of commands and applies them one at a time in the order of appearance. The `apply` command encapsulates the argument proof command making it an atomic proof step.

The `dur-preserve` command is a complex PC/DC proof step which applies the duration preservation proof rule. A significant amount of work is carried out by this command. First, the sequent is searched for a chopped antecedent formula of the form `p(x,y);p(x,y)` for a binary relation `p`. The search determines `p(x,y)` to be `30*x <= y`. Next, the *dur-preserve* proof rule is brought in as a lemma, instantiated with the definition of `p` and arguments `dur(Leak)` and `dur(tt)`, and simplified into two sub-goals. One sub-goal requires `p(x,y)` to satisfy the additivity requirement and the second goal continues the proof with `p(dur(Leak),dur(tt))` on the full interval. The latter goal is trivially satisfied, since the same formula now appears on both sides of the sequent. The former goal is finished by the decision procedures which are invoked by `assert`.

`dur-preserve` is an example of a proof command where underlying *strategies* alleviate the user from doing trivial computations. Such strategies are essential to lift the interaction with the proof checker to the same level of thoughts as the user would use when doing the proof on paper. The example also shows the benefit of using PVS with its tight integration of decision procedures as the semantic framework. A detailed description of the PC/DC commands can be found in [21].

5 Hiding the Encoding

This section shows how the details of the semantic encoding can be hidden in PC/DC proofs. Without the unparsing mechanism, the above proof obligation looks as follows:

```
  |-------
{1}  (FORALL (i: Interval): (T;T => T)(i))
```

which is Skolemized and rewritten into the sequent:

```
  |-------
{1}  (T;T)(i!1) IMPLIES T(i!1)
```

where `i!1` is the Skolem constant corresponding to `i` and `IMPLIES` is the PVS implication. Next, the implication is flattened

```
{-1}  (T;T)(i!1)
  |-------
{1}  T(i!1)
```

and the definitions expanded and rewritten:

```
{-1} ((n(30) * dur(Leak) <= dur(tt));(n(30) * dur(Leak) <= dur(tt)))(i!1)
  |-------
{1} 30 * dur(Leak)(i!1) <= dur(tt)(i!1)
```

Finally, the duration preservation rule is applied. Let `i!2` be a new interval Skolem constant obtained from this application. The additivity requirement is:

```
{-1}  30 * r1!1 <= s1!1
{-2}  30 * r2!1 <= s2!1
  |-------
{1}  30 * r1!1 + 30 * r2!1 <= s1!1 + s2!1
{2}  30 * dur(Leak)(i!2) <= dur(tt)(i!2)
```

which is trivially shown using the decision procedures of PVS. Note how the encoded propositional operators (like `=>`) and the encoded arithmetic and comparison operators are immediately rewritten into their PVS equivalents while `;` and `dur` remain. For the latter operators we instead apply the inference rules.

As illustrated above, the rules of first-order logic are used indirectly by rewriting top-level DC formulas to PVS Boolean formulas so that they can be handled by the proof commands of PVS. The rules specific to DC are encoded as indicated in Section 3. For instance, the duration preservation rule used in the previous examples is encoded in PVS as:

```
dur_preserve: LEMMA
     rule(seq(Gamma, p(dur(s1),dur(s2))^p(dur(s1),dur(s2)) + Delta),
          seq(Gamma, p(dur(s1),dur(s2)) + Delta))
```

where `s1` and `s2` are variables ranging over `State`, `+` is disjunction of formulas, `Gamma` and `Delta` (ranging over `IForm`) are intended to be a conjunct respectively a disjunct, and `p` is an additive predicate of the type `addpred` (in the following let `q` be of type `[real,real -> bool]` and `IReal` be the type `[Interval -> real]`):

```
addpred: TYPE = {p1: [IReal,IReal -> IForm] |
                          (EXISTS q: lifted(p1,q) AND additive(q))}
```

Note how the predicate subtyping of PVS is very useful to describe the meta-theoretical notion of lifted and additive predicates. The predicates `lifted` and `additive` are defined as (let `p1` be of type `[IReal,IReal -> IForm]`):

```
lifted(p1,q): bool = (FORALL (i:Interval),(ir1,is1:IReal):
                                 p1(ir1,is1)(i) = q(ir1(i),is1(i)))

additive(q): bool = (FORALL (r1,s1,r2,s2:real):
                       q(r1,s1) AND q(r2,s2) IMPLIES q(r1+r2,s1+s2))
```

The encoded rules (such as `dur_preserve`) are applied using PVS strategies which essentially are macros of primitive inferences. Strategies are used to form "intelligent" DC proof commands such as `dur-preserve`, i.e., proof commands that in a single step of the proof apply a chosen inference rule to the goal sequent in the appropriate manner. Strategies serve two purposes here. First, they allow users to apply the proof rules without requiring that they understand the semantical details. Second, they allow the user to apply inference rules without supplying much in the way of explicit guidance.

A strategy applies each encoded rule through a *search phase* and an *application phase*. The search phase extracts information from the goal sequent. The application phase suitably instantiates the conclusion of the inference rule, rewrites it into its semantical encoding in PVS, introduces a Skolem constant for the intervals in any new sequents introduced as premises by the inference rule, and hides any irrelevant formulas from the conclusion sequent.

As described previously, the proof command applying the duration preservation rule (1) identifies the first antecedent formula of the goal sequent that is a chopped formula with two identical subformulas, (2) searches for a consequent formula identical to the two subformulas, and (3) instantiates and applies the inference rule appropriately using the gathered information. Note that strategies do not in any way interfere with the prover internals and will therefore not jeopardize the soundness of the encoding or of the proofs.

A customized parser enables us to obtain a concrete syntax in PC/DC which closely resembles the usual DC syntax. For instance, parsing and unparsing permits the use of special operator symbols, which are not permitted in the PVS specification language (e.g., semi-colon). Furthermore, the DC unparser removes all occurrences of interval constants and quantified interval variables.

The PC/DC parser and unparser was generated using the ERGO parser/unparser generator [16] which was also used to generate the original PVS parser. In fact, we modified the original PVS syntax description and the interface between the parser and the PVS internals. The PVS Emacs interface was modified to distinguish between several parsers, such that the file extension of the theory file determines which parser/unparser to apply. The changes are generic such that new parsers for other formalisms can be included with very little work.

6 Related Work

Previously, the semantics of various logics have been encoded within existing theorem proving or proof checking systems. The approach is nicely described by Gordon [10] where he describes an encoding of Hoare Logic [15] within the HOL system [9]. Goldschlag [8] has similarly encoded the Unity logic [3] using the Boyer-Moore theorem prover [2]. Andersen [1] describes an encoding of Unity using the HOL system. An encoding of Timed Transition Systems and Real-Time Temporal Logic in HOL is described by Hale, Cardell-Oliver, and Herbert [11]. The advantage of these encodings is that a significant fraction of the capabilities of the underlying *base* logic and proof checking system can be exploited to support the encoded *source* logic. The typical disadvantage of the approach is that a user is required to understand the details of the encoding and often has to work directly in the base logic with the encoded form of the source logic. Chou [4] presents a method for lifting the HOL inference rules and tactics to operate at the predicate rather than the Boolean level. This form of lifting is also essential to our encoding of DC in PVS.

An alternative approach is to build a separate user interface. For instance, in the TLA proof checker [5], a customized user interface uses an encoding in the Larch Prover as a back-end inference engine.

7 Discussion

We have implemented a mechanized proof assistant prototype by carefully encoding the semantics of DC in the logic of PVS. We have developed a Gentzen style proof system which has been proven sound with respect to the semantic encoding. Next, we exploit the similarity between the sequent calculus of PVS and the sequent calculus of DC to obtain a DC proof system where the proofs are actually being carried out in PVS at the semantic level, but appear as proofs in the DC proof system. In fact, we are able to maintain the syntax and proof theory of DC without sacrificing the mechanization available in PVS.

The duration operator has not been semantically encoded; instead we have introduced axioms to describe its properties. Initial experiments indicate that it will not be difficult to formalize the Riemann integral and show the validity of the encoded axioms.

The approach illustrated here can be generalized to any formalism with a simple semantics. For instance, modal logics, temporal logic, Unity and TLA can be encoded in this way. More importantly, such encodings can be used as the basis for *combining* different formalisms in order to reason about system behavior in a convenient manner. As an example, Unity could be combined with Duration Calculus (DC) to reason about systems where designs are represented in Unity and real-time reasoning is performed in DC. These formalisms are unified by their semantic encodings, i.e., rules translating formulas from one formalism to another can be proven correct using the semantic encodings.

PC/DC provides the user with a lot of reasoning power but it also requires a certain discipline on the part of the user to use the system so as to preserve the appearance of DC formulas. The system does not prevent the user from explicitly expanding the semantic encoding, nor from introducing intervals in the proof.

So far, we have only checked small, well-known examples using the tool: the simple gas burner, a railway crossing, and a water level monitor. As expected, the decision procedures turned out to be useful. Next, the tool must be submitted to the scrutiny of bigger examples and different users to obtain a more robust system. Initial experiences indicate that the simple process of writing the specification in a rigorously typed environment ferrets out a number of errors in the specification.

Using PVS as a semantic framework has exercised it in a useful manner. It has illustrated how the higher-order logic of PVS and the tight integration of decision procedures can be used to encode and reason about modal formalisms. This work has inspired several modifications to PVS: incorporating new parsers within the same tool, user specified type coercion, and the application of decision procedures to indexed terms.

Acknowledgements

We are indebted to Sam Owre for his help during the construction of the prototype tool, particularly the PC/DC parser. We are also grateful for the valuable comments from Paul Curzon, Michael R. Hansen, Jozef Hooman, Sreeranga Rajan, Anders P. Ravn, Hans Rischel, Friedrich von Henke, and the anonymous referees.

References

1. F. Andersen. *A Theorem Prover for UNITY in Higher Order Logic.* PhD thesis, Technical University of Denmark, Lyngby, Denmark, 1992. Available as ID-TR: 1992-106.
2. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, NY, 1979.
3. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.
4. Ching-Tsun Chou. A sequent formulation of a logic of predicates in HOL. In L.J.M. Claesen and M.J.C. Gordon, editors, *Higher Order Logic Theorem Proving and Its Applications, 5th International Workshop*, number A-20 in IFIP Transactions, pages 71–80. North-Holland, 1992.
5. U. Engberg, P. Grønning, and L. Lamport. Mechanical verfication of concurrent systems with TLA. In *Computer Aided Verification 1992*, volume 663 of *Lecture Notes in Computer Science*, pages 44–55. Springer-Verlag, 1992.
6. M. Engel, A.P. Ravn, and H. Rischel. Timed-Z specification language for real-time systems. ProCoS II, ESPRIT BRA 7071 ID/DTH ME 1/1, Department of Computer Science, Technical University of Denmark, September 1993.
7. J.H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* Harper and Row, New York, 1986.
8. D. Goldschlag. Mechanizing unity. In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods.* North-Holland, 1990.

9. M.J.C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, Boston, MA, 1988.
10. M.J.C. Gordon. Mechanizing programming logics in Higher-Order Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Theorem Proving*, pages 387–439, New York, NY, 1989. Springer-Verlag.
11. R. Hale, R. Cardell-Oliver, and J. Herbert. An embedding of timed transition systems in HOL. *Formal Methods in System Design*, 3(1/2):151–174, August 1993.
12. J. Halpern, B. Moszkowski, and Z. Manna. A hardware semantics based on temporal intervals. In J. Díaz, editor, *10th ICALP. Automata, Languages and Programming*, volume 154 of *Lecture Notes in Computer Science*, pages 278–291. Springer-Verlag, 1983.
13. M. R. Hansen and Zhou Chaochen. Semantics and completeness of Duration Calculus. In W.-P. de Roever J. W. de Bakker, C. Huizing and G. Rozenberg, editors, *Real-Time: Theory in Practice, REX Workshop*, volume 600 of *Lecture Notes in Computer Science*, pages 209–225. Springer-Verlag, 1992.
14. M. R. Hansen, Zhou Chaochen, and J. Staunstrup. A real-time duration semantics for circuits. In *TAU 1992 ACM/SIGDA Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, March 1992.
15. C. A. R. Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
16. P. Lee, F. Pfenning, J. Reynolds, G. Rollins, and D. Scott. Research on semantically based program-design environments: The ERGO project in 1988. Technical Report CMU-CS-88-118, Department of Computer Science, Carnegie Mellon University, 1988.
17. B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Transactions on Computers*, 18(2):10–19, 1985.
18. S. Owre, N. Shankar, and J.M. Rushby. *User Guide for the PVS Specification and Verification System, Language, and Proof Checker (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA, February 1993. Three volumes.
19. A.P. Ravn, H. Rischel, and K. M. Hansen. Specifying and verifying requirements of real-time systems. *IEEE Trans. Software Engineering*, 19(1):41–55, Jan. 1993.
20. J.U. Skakkebæk. Development of provably correct systems. Master's thesis, Department of Computer Science, Technical University of Denmark, 1991.
21. J.U. Skakkebæk, S. Owre, and N. Shankar. A user guide for the Duration Calculus proof checker, 1994. In Manuscript.
22. J.U. Skakkebæk and N. Shankar. A Duration Calculus proof checker: Using PVS as a semantic framework. Technical Report SRI-CSL-93-10, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA, December 1993.
23. L. Wallen. *Automated Deduction in Nonclassical Logics*. MIT Press, 1990.
24. Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, December 1991.
25. Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations, 1992. Revised June 3, 1992.

This article was processed using the LaTeX macro package with LLNCS style