A Duration Calculus Proof Checker: Using PVS as a Semantic Framework*

Jens U. Skakkebæk[†]& N. Shankar[‡]

Abstract

The Duration Calculus (DC) is an interval temporal logic for reasoning about real-time systems. We present a Gentzen-style sequent calculus for DC and a tool for constructing DC specifications and checking DC proofs. This tool is implemented by encoding the semantics of DC within a general-purpose specification and verification environment called PVS. In encoding DC, we have used PVS as a semantic logical framework, that is, we have employed the higher-order logic of PVS to define the semantics of DC and we have formally derived the DC inference rules using this semantic definition. The DC inference rules can then be used without any knowledge of the semantic encoding. The inference capabilities of PVS such as rewriting, decision procedures, and user-defined proof strategies can be applied within DC proofs. The DC sequent calculus, the semantic encoding of DC in PVS, and the DC proof checking tool are illustrated using the example of a gas burner.

*The work of the first author was partially sponsored 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.

[†]Visiting SRI International Research Fellow. On leave from Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark.

[‡]SRI International Computer Science Laboratory, Menlo Park CA 94025 USA.

Contents

List of Figures

1 Introduction

The Duration Calculus (DC) [5] 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) [17, 29]. DC was the first real-time formalism to introduce the concept of an integrating (duration) operator for reasoning about *intermittent* system behaviors. Examples of applications can be found in [21, 32, 34]. Since DC has been employed in safety-critical applications and has hitherto lacked any effective mechanical support, we have developed a specification front-end, a sequent calculus, and a proof checker for DC. This mechanization has been carried out within the framework of the PVS specification and verification environment [30]. An important point regarding our mechanization is that it retains all of the features of the PVS specification language and proof checker while staying within the notational framework of the Duration Calculus. This report illustrates the use of PVS as a semantic logical framework with the example of the embedding of the duration calculus within PVS. We present the details of this mechanization and argue that our approach can be applied to specification logics other than DC.

In the rest of this introduction, we briefly introduce the problem of mechanizing the Duration Calculus, and consider various alternative approaches to this task.

Mechanizing the Duration Calculus

To motivate and illustrate our goals for a mechanization of DC, we 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, facing the highway traffic. 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 (the reals) to $\{0,1\}$, we can specify this using the Riemann integral[1]:

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

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

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

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

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

specifications of controllers, sensors, and actuators. Each new refinement must be proven to implement the previous specification. 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 type in the above specification as, say:

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

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 subintervals, it also holds on the full interval. In DC this is written:

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

where '$;$' is the *chopping* operator that divides an interval into two contiguous subintervals, and *Dur-Preserve* is a general lemma with the precondition that the binary predicate (here: $p(x,y) \mathrel{\widehat{=}} \frac{9}{8}x \leq y$) is monotone in its arguments.[2] 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.

For instance, in our proof assistant prototype, called PC/DC, the above proof obligation (in a corresponding sequent formulation) looks as follows:

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

```
{-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 (number `1`) below. The proof tool command **`dur-preserve`** trivially completes the main proof as expected. However, the tool requires the user to prove monotonicity of the predicate (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 complete automatically.

PC/DC is obtained by carefully encoding the semantics of DC in the typed logic of PVS so that the mechanized support (typechecker, arithmetic and equality decision procedures, rewriting, etc.) available for the PVS logic can be exploited for constructing DC proofs. The typical disadvantage of such semantic encodings is that a user is required to understand the details of the encoding and often has to work directly in the encoding logic with the encoded form of the encoded logic. We now discuss various alternative approaches for mechanizing DC and indicate where our approach fits among these alternatives.

Mechanizing Special-Purpose Logics

There are several ways to mechanizing special-purpose logics such as DC. The most obvious approach would be to directly implement a tool that mechanizes DC. The effort involved in developing the parsing, typechecking and deductive components of such a tool would be considerable. The other approach would be to use an existing tool and to somehow embed DC within this tool so that effort involved in constructing the embedding retains the best features of both DC and the tool being used.

There are two ways of adapting an existing tool to construct a DC proof checker. One approach is to use the existing tool as a subroutine to a newly constructed DC proof checker. This subroutine is invoked to perform the arithmetic and logical reasoning steps. The DC proof steps are performed by the DC proof checker itself. This approach is used in the TLA [26] proof checker where the Larch Prover (LP) [11] is used as a back-end theorem prover [8]. The advantages of this approach are that the front-end is customized to the logic of interest, and several powerful back-end systems can be employed as subroutines. One disadvantage of this approach is that the exchange of information between one proof checking system and another can be very delicate. The quality of feedback one gets from such a subroutine, particularly when it fails, may not always be comprehensible to a user.[3]

The other approach to adapting an existing tool is by means of a logical embedding of a source logic, which in this case is DC, within the base logic used by the tool. There are two approaches to the embedding of one logic, the source logic, within another, the base logic. In a *syntactic embedding*, the base logic is used as a metalogic in which the source logic is defined by introducing the syntax and proof rules as first-class entities in the base logic. In a *semantic embedding*, the semantics of the source logic is described using the base logic and the desired proof rules are derived by means of lemmas in the base logic.

Syntactic logical frameworks provide a convenient way of defining new source logics by directly supporting the syntax of such a logic. To ensure correctness of an embedding, proofs of adequacy (soundness) should be carried out. For incomplete logics or logics not known to be complete, new axioms might have to be added at a later stage and proofs of adequacy extended and checked. For effective mechanization, the base logics are typically quite weak and do not support meta-level reasoning. So, for instance, it is not possible to derive any new inference rules from the primitive inference rules of the logic. While syntactic logical frameworks do make it easy to define new logics and provide computatonal mechanisms for formal proof construction, they provide little help in achieving the degree of mechanization that is given, for instance, by decision procedures. The various syntactic logical frameworks in use include Isabelle [31], λ-Prolog [28], and the LF [22].[4] It would be an interesting exercise to formalize DC using a syntactic logical framework and compare it to our formalization here.

Semantic encodings are commonly used for reasoning in temporal logics, modal logics and program formalisms (e.g., Unity [3], Hoare triples [23]). Ensuring correctness of the encoding is done once and for all by proving that the encoding is faithful to the semantics stated on paper. All axioms and proof rules are derived from the semantical description. For incomplete logics and logics of unknown completeness, in case new proof rules are needed, they can simply be derived from the semantic encoding.[5] Separate proofs of adequacy are not needed. Also, meta-level reasoning is permitted since new proof rules can be shown to be sound by semantic means. Another advantage is that a significant fraction of the capabilities of the underlying base

[3] PVS in fact uses various arithmetic and equality decision procedures in exactly this manner. Both PVS and the decision procedures are required to maintain consistent contexts and the decision procedures operate as black boxes that attempt to simplify given formulas to *true* or *false* in a given context.

[4] Although general-purpose theorem provers and proof checkers (such as PVS) could, in principle, be used in a syntactic embedding of some of the logics, they are not tailored specifically to this task and do not immediately provide the desired functionality of a syntactic framework.

[5] DC is not known to be complete. Only relative completeness has been shown [19]. This is also the case for a later extension for reasoning about hybrid systems. This is one reason why we favor a semantic framework as the basis of a DC proof checker.

logic and proof checking system can be exploited to support the source logic. One disadvantage of semantic logical frameworks is that they are only applicable when the semantics of the source logic can be correctly defined within the base logic. Since many typical logics have Tarskian semantics that can be defined within a sufficiently expressive classical logic, this is not a significant obstacle. However there are various *substructural* logics such as linear [12] and relevance logic [7] where the Tarskian semantics get to be quite cumbersome. The other disadvantage of the semantic approach is that quite a bit of work is needed to provide the syntactic convenience available in syntactic logical frameworks. Often, the 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.

We have chosen the approach of semantically encoding the proof rules of DC within an existing proof checking framework. This approach has been used previously and is nicely described by Gordon [15] where he presents an encoding of Hoare Logic [24] within the HOL system [14]. Goldschlag [13] 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 [16]. Chou [6] 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 embedding of DC in PVS.

Encoding DC in PVS

In the work presented here, we try to address problems of complex and detailed encodings by permitting the user to work directly with the syntax and proof rules of DC and not their 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. This report gives a detailed description of the encoding and an example.

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 can easily be generalized 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. There are, of course, systems other than PVS that would also be useful for this purpose but we have chosen PVS because it offers a convenient combination of an expressive specification notation combined with effective mechanization.

The following presentation of PC/DC the PVS-based proof checker for DC, uses a simple gas burner as the running example. Section 2 introduces the syntax, semantics, and proof theory of the Duration Calculus. Section 3 briefly introduces PVS. Section 4 describes the implementation in PVS of the proof checker PC/DC based on an embedding of a sequent calculus for DC within PVS. The basic technique that is used here to encode the Duration Calculus within PVS can easily be applied to other modal and program logics. Section 5 illustrates the use of PC/DC. The DC proof system is provided in Appendix A. A more detailed description of the encoding is found in Appendix B. Appendix C contains a detailed description of the gas burner example. The PC/DC user guide [33] provides a detailed description of the PC/DC user interface.

2 The Duration Calculus

In classical logic, all reasoning is relative to a single universe that fixes the meanings of all the propositional atoms and the function and predicate symbols. A sentence is *valid* in such a logic if it holds true in every universe. Such logics are only capable of dealing with static universe in which sentences have fixed truth values. They are not well suited for handling dynamic states of affairs. Modal logics introduce the notion of possible worlds so that meanings of sentences are interpreted relative to these possible worlds. A propositional atom can be true in one world and not in the other. The Kripke model for a modal logic [18] consists of a set of worlds, the interpretation for each propositional atom, function, or predicate symbol relative to each world, and a binary accessibility relation between worlds. The modal statement $\Box A$ holds in a given world of a given model if and only if A is true in every accessible world. A *valid* modal statement is one that is true in every world of every model. In temporal logics, the worlds represent states and the accessibility relation corresponds to the temporal relationship between states: a world v is accessible from world u if v follows u in time. In an interval temporal logic, the worlds correspond to time intervals rather than states. A world (interval) v is accessible from a world (interval) u if v is a subinterval of u.[6]

[6]Note that there are interval temporal logics with slightly different notions of accessibility. Venema [35] uses a *ternary* accessibility relation between intervals.

The Duration Calculus (DC) is an extension of Moszkowski's interval temporal logic [29]. A *state* in DC is essentially a time-varying Boolean signal. The duration operator, inspired by the definite integral operator of calculus, computes the accumulated length of time that a state has been true within an interval. Equalities and inequalities involving the duration operator can be used to place upper and lower bounds on accumulated time. The duration operator can be used to state properties of real-time systems. Examples of assertions that can be stated using interval temporal logic combined with the duration operator include:

1. *Gas is leaking for at most 5% of time over a sufficiently large interval;*
2. *The interval between two successive gas leaks is at least 30 seconds;*
3. *In any subinterval, almost everywhere over the subinterval, no two processes are simultaneously in their critical sections;*
4. *Over an interval of length x, two clocks drift apart by at most $2\rho x$.*

The following explanation of DC syntax and semantics follows closely the presentation found in [20]. The semantics of DC will be embedded in PVS so that it is possible to construct DC proofs using DC syntax.

2.1 Syntax

We now describe the syntax of DC starting with state expressions which are analogous to signals. The duration operator applied to state expressions measures the duration (over an interval) for which the state expression as a signal has been high. These durations can be combined with various arithmetic operators to get DC terms which in turn can be combined with various arithmetic relations to get the atomic formulas of DC.

A *state expression* is either a *constant* 0 or 1, a *state* X, a negation $(\neg P)$, or a disjunction $(P \vee Q)$, where P and Q are states. Semantically, a state expression represents a function from time (ranging over the reals $\mathbf{R}$) to the set $\{0, 1\}$.

A *term* is either a *duration* $\int P$ for a given state P, a *global variable* x, or a function application $f_i^n(r_1, \ldots, r_n)$, where $r_1, \ldots, r_n$ are terms and f_i^n is an n-ary function letter. Semantically, given an assignment of values to variables, a term is a function from an interval $[b, e]$ to the reals $\mathbf{R}$.

A *duration formula* is either one of the constants *true* or *false*, an *atomic duration formula* $A_i^n(r_1, \ldots, r_n)$, a negation $(\neg\mathcal{D}_1)$, a disjunction $(\mathcal{D}_1 \vee \mathcal{D}_2)$, a chop $(\mathcal{D}_1 \,;\, \mathcal{D}_2)$, or a universal quantification $(\forall x.\ \mathcal{D}_1)$, where x is a global variable, $\mathcal{D}_1$ and $\mathcal{D}_2$ are duration formulas, A_i^n is an n-ary predicate letter and $r_1, \ldots, r_n$ are terms. Note that the Boolean connectives $\neg$ and $\vee$ on duration formulas are semantically different from those on states.

We shall use $\wedge, \Rightarrow$, and $\Leftrightarrow$ to abbreviate the equivalent expressions involving $\vee$ and $\neg$ for both states and duration formulas, and we freely use infix notation for functions (such as $+$) and predicates (such as $=$). Furthermore, we use the following abbreviations:

$$
\begin{array}{lcll}
\ell & \widehat{=} & \textstyle\int 1 & \\
\lceil P \rceil & \widehat{=} & (\textstyle\int P = \textstyle\int 1) \wedge \neg(\textstyle\int 1 = \textstyle\int 0) & \\
\lceil\,\rceil & \widehat{=} & \neg\lceil 1 \rceil & \\
\Diamond\mathcal{D} & \widehat{=} & \mathit{true}\,;\mathcal{D}\,;\mathit{true} & \text{reads "for some subinterval"} \\
\Box\mathcal{D} & \widehat{=} & \neg\Diamond(\neg\mathcal{D}) & \text{reads "for every subinterval"}
\end{array}
$$

Here ℓ returns the length of the current interval. Thus the formula $\lceil P \rceil$ means that the state P is 1 almost everywhere in a given interval, and $\lceil\,\rceil$ is true only for point intervals. For a duration formula $\mathcal{D}$, $\Diamond\mathcal{D}$ asserts that $\mathcal{D}$ holds of some subinterval of the current interval, and $\Box\mathcal{D}$ asserts $\mathcal{D}$ to hold of every subinterval of the current interval.

In the following we assume $\neg$ to bind stronger than $;$, $\Diamond$, and $\Box$ which again bind stronger than $\wedge$ and $\vee$ followed by $\Rightarrow$ and $\Leftrightarrow$.

2.2 Semantics

We now define an interpretation of the Duration Calculus notation by recursion on the syntax given above. The PVS definition of the semantics of DC as described below, is given in Section 4.

For $b, e \in \mathbf{R}$ and $b \leq e$, the *time interval* $[b, e]$ is the subset $\{t \in \mathbf{R} \mid b \leq t \leq e\}$ of real numbers. *Intv* denotes the domain of intervals $\{[b, e] \mid b \leq e \wedge b, e \in \mathbf{R}\}$. An *interpretation* $\mathcal{I}$ is a function which assigns a total function of type $\mathbf{R} \to \{0, 1\}$ to each state, a total function of type $\mathbf{R}^n \to \mathbf{R}$ to each n-ary function letter f_i^n, and a total function of type $\mathbf{R}^n \to \{tt, ff\}$ to each n-ary predicate letter A_i^n. We require that for any state X, $\mathcal{I}(X)$ must be *finitely variable*, i.e., it have only a finite number of discontinuity points in any interval.

The meaning of a state expression P in interpretation $\mathcal{I}$ is a function:

$$\mathcal{I}[\![P]\!] \in (\mathit{Time} \to \{0, 1\})$$

defined inductively on the structure of state expressions:

$$
\begin{array}{lcl}
\mathcal{I}[\![0]\!](t) & = & 0 \\
\mathcal{I}[\![1]\!](t) & = & 1 \\
\mathcal{I}[\![X]\!](t) & = & \mathcal{I}(X)(t) \\
\mathcal{I}[\![(\neg P)]\!](t) & = & 1 - \mathcal{I}[\![P]\!](t) \\
\mathcal{I}[\![(P \vee Q)]\!](t) & = & \begin{cases} 0 & \text{if } \mathcal{I}[\![P]\!](t) = 0 \text{ and } \mathcal{I}[\![Q]\!](t) = 0 \\ 1 & \text{otherwise} \end{cases}
\end{array}
$$

A *valuation* is a pair $(\mathcal{V}, [b, e])$ where $\mathcal{V}$ associates a real value with each variable x and $[b, e]$ is an interval. Let $\mathcal{V}al$ be the set of all valuations. Two valuations $(\mathcal{V}, [b, e])$ and $(\mathcal{V}', [b, e])$ are *x-equivalent* if they agree on the value of every variable other than x. Terms and duration formulas are interpreted over time intervals relative to a given assignment of values to variables. The meaning of a *term* r in an interpretation $\mathcal{I}$ is therefore a function:

$$\mathcal{I}[\![r]\!] \in (\mathcal{V}al \to \mathbf{R})$$

defined inductively on the structure of terms as:

$$\begin{array}{lcl} \mathcal{I}[\![\textstyle\int P]\!](\mathcal{V}, [b, e]) & = & \int_b^e \mathcal{I}[\![P]\!](t)\,dt \\ \mathcal{I}[\![x]\!](\mathcal{V}, [b, e]) & = & \mathcal{V}(x) \\ \mathcal{I}[\![f_i^n(r_1, \ldots, r_n)]\!](\mathcal{V}, [b, e]) & = & \mathcal{I}(f_i^n)(\mathcal{I}[\![r_1]\!](\mathcal{V}, [b, e]), \ldots, \mathcal{I}[\![r_n]\!](\mathcal{V}, [b, e])) \end{array}$$

In the above definition, the expression $\int_b^e \mathcal{I}[\![P]\!](t)\,dt$ employs the Riemann integral[7] to compute the duration of a state over an interval. In the following we use the classic operators (i.e., $+$, $-$, $*$, $/$) of real arithmetic with their usual meanings assigned by the interpretation.

The meaning of a *formula* $\mathcal{D}$ in an interpretation $\mathcal{I}$ is a function:

$$\mathcal{I}[\![\mathcal{D}]\!] \in (\mathcal{V}al \to \{tt, ff\})$$

defined inductively on the structure of formulas below.

We use the abbreviations:

$$\begin{array}{l} \mathcal{I}, \mathcal{V}, [b, e] \models \mathcal{D} \;\widehat{=}\; \mathcal{I}[\![\mathcal{D}]\!](\mathcal{V}, [b, e]) = tt \\ \mathcal{I}, \mathcal{V}, [b, e] \not\models \mathcal{D} \;\widehat{=}\; \mathcal{I}[\![\mathcal{D}]\!](\mathcal{V}, [b, e]) = ff \end{array}$$

in the inductive definition:

$$\begin{array}{lll} \mathcal{I}, \mathcal{V}, [b, e] \models true & & \\ \mathcal{I}, \mathcal{V}, [b, e] \models A_i^n(r_1, \ldots, r_n) & \text{iff} & \mathcal{I}(A_i^n)(\mathcal{I}[\![r_1]\!](\mathcal{V}, [b, e]), \ldots, \mathcal{I}[\![r_n]\!](\mathcal{V}, [b, e])) = tt \\ \mathcal{I}, \mathcal{V}, [b, e] \models \neg\mathcal{D} & \text{iff} & \mathcal{I}, \mathcal{V}, [b, e] \not\models \mathcal{D} \\ \mathcal{I}, \mathcal{V}, [b, e] \models (\mathcal{D}_1 \vee \mathcal{D}_2) & \text{iff} & \mathcal{I}, \mathcal{V}, [b, e] \models \mathcal{D}_1 \text{ or } \mathcal{I}, \mathcal{V}, [b, e] \models \mathcal{D}_2 \\ \mathcal{I}, \mathcal{V}, [b, e] \models (\mathcal{D}_1\,;\,\mathcal{D}_2) & \text{iff} & \mathcal{I}, \mathcal{V}, [b, m] \models \mathcal{D}_1 \text{ and } \mathcal{I}, \mathcal{V}, [m, e] \models \mathcal{D}_2, \\ & & \text{for some } m \in Time \text{ where } m \in [b, e] \\ \mathcal{I}, \mathcal{V}, [b, e] \models \forall x.\, \mathcal{D} & \text{iff} & \mathcal{I}, \mathcal{V}', [b, e] \models \mathcal{D}, \\ & & \text{for all } (\mathcal{V}', [b, e]) \text{ } x\text{-equivalent with } (\mathcal{V}, [b, e]) \end{array}$$

An interpretation $\mathcal{I}$ *is a model of* a duration formula $\mathcal{D}$ (or $\mathcal{D}$ is *true* in $\mathcal{I}$), written $\mathcal{I} \models \mathcal{D}$, iff $\mathcal{I}, \mathcal{V}, [b, e] \models \mathcal{D}$ for all valuations $(\mathcal{V}, [b, e])$.

$\mathcal{D}$ is *valid* (written $\models \mathcal{D}$) iff $\mathcal{I} \models \mathcal{D}$ for every interpretation $\mathcal{I}$.

A formula $\mathcal{D}$ is called *satisfiable* iff there exists an interpretation $\mathcal{I}$ and a valuation $(\mathcal{V}, [b, e])$ in $\mathcal{I}$ such that $\mathcal{I}, \mathcal{V}, [b, e] \models \mathcal{D}$. It is easy to see that $\mathcal{D}$ is satisfiable if and only if $\not\models \neg\mathcal{D}$.

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

2.3 The DC Sequent

The PC/DC tool is based on the Gentzen style proof system [10] for DC shown below. The proof system is very similar to the basic system presented in [9], which was developed in parallel with our work. The proof rules described below are actually proved in PVS using the semantic embedding of DC within PVS.

Let A, B, C and D be Duration Calculus formulas. A formula or a term is *duration free* if it does not contain any symbols of the form $\int P$ for any state P. 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.

A sequent $\Gamma \vdash \Delta$ is *true* in the interpretation $\mathcal{I}$ (written $\mathcal{I} \models \Gamma \vdash \Delta$) iff for any valuation $(\mathcal{V}, [b, e])$ in $\mathcal{I}$, if $\mathcal{I}, \mathcal{V}, [b, e] \models A$ for all $A \in \Gamma$, then for some $B \in \Delta$, $\mathcal{I}, \mathcal{V}, [b, e] \models B$.

$\Gamma \vdash \Delta$ is *valid* (written $\models \Gamma \vdash \Delta$) iff $\mathcal{I} \models \Gamma \vdash \Delta$ for all interpretations $\mathcal{I}$.

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*. 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.

The DC inference rules combine the usual structural and propositional rules with special rules for chop operator and for the duration operator. The *basic* inference rules axiomatize the calculus. The *derived* rules are defined for practical convenience and can be proven using the basic inference rules. The system is shown in detail in Appendix A.

As an example of an inference rule special to DC, consider the rule of *monotonicity*:

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

Often identical formulas on adjacent subintervals can be shown to hold on the full interval. In fact, all expressions preserving the relationship between durations on intervals and which hold on any adjacent subintervals also hold on the full interval. Let $\otimes$ be a relation such that:

$$r_1 \otimes s_1 \wedge r_2 \otimes s_2 \Rightarrow (r_1 + r_2) \otimes (s_1 + s_2)$$

for any $r_1, s_1, r_2, s_2 \in \mathbf{R}$ (e.g. $\otimes\ is\ <$). Then:

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

2.4 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 semantically true DC conclusions when the premises to the rules are semantically true. We will not repeat the proofs here. Interested users can obtain the relevant proofs from the authors.

The above proof system for DC has not been shown complete with respect to the given semantics. That is, there might be semantically true formulas that cannot be proved using the above proof system. For this reason, the encoding of DC in PVS allows the user to show the validity of formulas directly in the semantic encoding.

A proof of relative completeness of a Hilbert style proof system for an extension of ITL with durations is found in [19].

3 The PVS Prover

PVS [30] is an environment for writing specifications and developing proofs. It was developed as a prototype for exploring new approaches to mechanized formal methods for computing. The primary goal of PVS is to combine an expressive specification language with a productive, interactive proof checker that has a reasonable amount of theorem proving power. PVS has been strongly influenced by the observation that theorem proving capabilities can be employed to enrich the type system of a typed logic and, conversely, that an enriched type system facilitates more expressive specifications and effective theorem proving.

3.1 The Specification Language

The PVS specification language builds on a classical typed higher-order logic. The base types consist of booleans, real numbers, rationals, integers, natural numbers, lists, and so forth. The primitive type constructors include those for forming function, record, and tuple types. PVS departs from simply typed logics by allowing *predicate subtypes*. A predicate subtype consists of exactly those elements of a given type satisfying a given predicate. Predicate subtypes are used to explicitly constrain the domains and ranges of operations in a specification. They also make it possible to define functions that are only partially defined, *e.g.*, division, as total functions on a specified subtype. The division operation can be defined only on nonzero denominators. Typechecking with predicate subtypes is obviously undecidable, since it is undecidable, for example, whether a given denominator expression for division is nonzero.[8] PVS contains a further enrichment to the type system in the form of *dependent* function, record, and tuple constructions where the type of one component of a compound value depends on the value of another component. Dependent types are both natural and useful in the context of a specification language. PVS specifications are packaged as *theories*. Theories can be parametric in types and/or individuals. Some form of type parametricity (or *polymorphism*) is absolutely essential in a typed language so that certain important concepts and results can be stated uniformly for all types. PVS also has a facility for automatically generating abstract datatype theories (containing recursion and induction schemes) for a class of abstract datatypes.

PVS also contains a number of utilities for browsing, maintaining, modifying, and presenting specifications and proofs.

3.2 The Proof Checker

The PVS proof checker is intended to serve as a productive medium for debugging specifications and for constructing readable proofs. The human verifier constructs proofs in PVS by repeatedly simplifying a conjecture into subgoals using inference rules, until no further subgoals remain. A proof goal in PVS is represented by a *sequent* since it succinctly captures most of the information relevant to a branch of a proof, and in this sense, sequents are suitable for both human and machine use. PVS differs from most proof checkers in providing primitive inference rules that are fairly powerful. Decision procedures for ground linear arithmetic and equality are frequently invoked from primitive inference rules. The primitive rules also perform

[8]PVS does have an effective typechecker which easily checks for type correctness relative to the simple types. It generates proof obligations corresponding to predicate subtypes. The typical proof obligations can be automatically discharged by the PVS decision procedures. The provability of such proof obligations is the only source of undecidability in the PVS type system so that none of the benefits of decidable typechecking are lost.

steps such as rewriting, beta-reduction, and Boolean simplification. More complicated proof *strategies* can be defined by combining simpler inference rules by means of an extremely simple strategy language. The typechecker is frequently invoked by the proof checker, for example, to check if an instantiating term respects the type of the instantiated variable. The resulting proof obligations generated by the typechecker are presented as subgoals. Proofs and partial proofs can be saved, edited, and rerun. It is possible to extend and modify specifications during a proof; the finished proof has to be rerun to ensure that such changes are benign.

4 Implementing PC/DC

We have already argued that classical first-order or higher-order logic is not suitable for capturing dynamic behavior. PVS supports a classical higher-order logic and hence may not be the best formalism in situations where the Duration Calculus is applicable. However, PVS does contain many features that are useful in a specification/verification environment. We are therefore interested retaining these useful features in the process of constructing a notational layer above PVS called PC/DC that is based on the Duration Calculus. We do this by defining the semantics of DC presented in Section 2 within PVS, deriving the inference rules of DC from this semantic definition, and masking the semantic encoding with DC syntax in both the specification language and the proof checker. We now describe the PVS encoding of the DC semantics.

Observe that the inference rules of DC include those of classical propositional logic. Also observe that the semantics of DC can be expressed in first-order logic (which of course includes propositional logic). In general, it is a characteristic of our approach that the proof system of the source logic must include the propositional proof system of the base logic as a subset. This allows us to mimic base logic proofs (in the semantic encoding) as source logic proofs.

For a formula A belonging to the axiomatized subset of DC formulas, showing validity of A is to prove that $\vdash A$.

The syntax and semantics of DC formulas can be described in PVS. To show that a given DC formula is valid, we fix an interpretation and a value assignment and show that the formula is true with respect to this valuation for *all* intervals. As an example, let $A \vee B$ and $A \Rightarrow B$ be the DC formulas representing the conjunction and implication, respectively, of two DC formulas A and B. Assume we want to prove $\vdash A \Rightarrow (A \vee B)$ in DC. In the DC proof system the proof could be:

$$\cfrac{\cfrac{\cfrac{}{A \vdash A, B}\ \mathbf{Ax}}{A \vdash (A \vee B)}\ \vdash\vee}{\vdash A \Rightarrow (A \vee B)}\ \vdash\Rightarrow$$

In a semantic framework, proving $\vdash \ A \Rightarrow (A \vee B)$ corresponds to showing the semantical soundness of the formula $(A \Rightarrow (A \vee B))(i)$ for all intervals i. Equivalently, let i_1 be a new constant denoting an arbitrary interval. Validity can be shown by proving that $(A \Rightarrow (A \vee B))(i_1)$ is true, which again is equivalent to showing that $A(i_1)$ *implies* $(A(i_1)$ *or* $B(i_1))$ holds, where *implication* and *or* is the semantic notion of implication and disjunction, respectively. In PVS, that $A \Rightarrow (A \vee B)$ is valid is written:

```
|-------
{1} (FORALL (i:Time): (A => (A \/ B))(i))
```

which is Skolemized and rewritten into the sequent:

```
|-------
{1} A(i!1) IMPLIES (A(i!1) OR B(i!1))
```

where `i!1` is the Skolem constant corresponding to i_1 above.

The above sequent contains only one formula below the turnstile `|-------`. In general, a sequent will contain a (possibly empty) list of *antecedent* formulas, say $A_1, \ldots, A_m$, above the turnstile, and another (possibly empty) list of *consequent* formulas, say $B_1, \ldots, B_n$, below the turnstile. The meaning of such a sequent is that the conjunction of the antecedent formulas $A_1 \wedge \ldots \wedge A_m$ implies the disjunction of the consequent formulas $B_1 \vee \ldots \vee B_n$.

Reasoning within PVS, the proof of the above sequent can be continued by flattening the formula:

```
{-1} A(i!1)
|-------
{1} A(i!1)
{2} B(i!1)
```

which is trivially true. Note how closely the PVS proof follows the DC proof. The only major difference is the interval index. The above PVS proof can be transformed into a DC proof by hiding the index and mapping the PVS propositional operators to DC equivalents. The resulting proof is:

```
|-------
{1} A => (A \/ B)
```

which again can be flattened:

```
{-1} A
|-------
{1} A
{2} B
```

where `=>` and `\/` are abstractions of `IMPLIES` and `OR`.

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 the PC/DC proof checking system to be described below, the semantic encoding of DC in PVS is not explicitly visible to a user. Proofs are constructed in the DC proof system as is done in a pencil-and-paper argument. The details of the semantic encoding are not relevant at this level since they are only needed to derive the validity of the axioms and proof rules.

This trick of using and hiding semantics can only be applied to logics that are based on classical propositional logic. Relevance and linear logic [12], for example, do not obey the structural rules of the sequent system and are therefore not easily semantically encoded in the manner described above.

The following sections are intended to give an overview of implementation of PC/DC. Appendix B provides a detailed presentation. The implementation consists of a series of PVS theories defining DC, a number of inference rules encoded as PVS lemmas, strategies which use these inference rules and a customized parser. The definitions and inference rules are the heart of the encoding; the strategies and parser abstract away the semantical details.

4.1 Definitions

The DC domains, operators and their semantics are defined in a series of PVS theories. These definitions are explained below; the full PVS specification is given in Appendix B.

Domains

The time domain `Time` is defined as the reals. The positive, non-negative, and non-zero subdomains are also easily defined as subtypes of `Time`. The `CONTAINING` construct is used to demonstrate that these subtypes are nonempty. States are functions from `Time` to `bool`. The type `Interval` of intervals is a tuple type of the form `[Time, Time]` where the second value of the tuple is no smaller than the first. Interval formulas (of type `IForm`) are functions from `Interval` to `bool`. Interval measures (type `ITime`) are functions from `Interval` to `Time`:

```
Time : TYPE = real
NZTime : TYPE = {x:Time|x/=0} CONTAINING 1
PosTime : TYPE = {x:Time|x>=0} CONTAINING 1
NZPosTime : TYPE = {x:PosTime|x/=0} CONTAINING 1

State : TYPE = [Time->bool];

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

IForm : TYPE = [Interval -> bool];
ITime : TYPE = [Interval -> Time];
NZITime : TYPE = [Interval -> NZTime];
IPosTime : TYPE = [Interval -> PosTime];
NZIPosTime : TYPE = [Interval -> NZPosTime];
```

The semi-colons in the above specification are used as separators between declarations in PVS. They are only needed when there is some syntactic ambiguity concerning the end of a declaration. All the types allowed in PVS (e.g., records, sets, lists) are also available in PC/DC.

Operators

Each DC operator is declared as a PVS function. In PVS, a function declaration has the form:

```
<name>(<args>) : <type> = <body>
```

`body` of an operator declaration defines the semantical meaning of the operator.

Let `r` be a variable of type `Time` . The state constants `tt` and `ff` are respectively defined as the everywhere true and everywhere false functions on the time domain.

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

State assertions can be combined into composite state assertions using propositional operators. Let `p` and `q` be variables of type `State`. Then the conjunction, disjunction, and negation operators on states are defined as shown below:

```
/\ (p,q)(r) : bool = p(r) AND q(r);
\/ (p,q)(r) : bool = p(r) OR q(r);
not (p)(r) : bool = NOT p(r);
```

A DC *formula* has the type `IForm`. Operators on DC formulas are declared as functions of DC formulas returning a DC formula.

Let `i` be a variable of type `Interval` and let `A` and `B` be variables of type `IForm`. The conjunction, disjunction, and negation operators on DC formulas are defined as:

```
/\(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);
```

where the operators have been declared in their Curried form. That is, the operator takes two DC formula arguments and returns a function which takes an `Interval` and returns a `bool`.

Arithmetic and *comparison* operators are declared in a similar fashion. Let `t1` and `t2` be variable of type `ITime` and `nt2` of type `NZITime`:

```
*(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);

>(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);
```

The operators are overloaded to operate on arguments of type `NZITime` as well.

We want to be able to use *equality* on terms of any type and therefore declare it in a parametric PVS theory:

```
dcp_equality [t : TYPE]: THEORY

  BEGIN
    ...
    A,B : VAR [Interval -> t]

    =(A,B)(i) : bool = (A(i) = B(i));
    ...
  END dcp_equality
```

The theory takes a parameter `t` ranging over types. The parameter `t` may be used as any other type within the theory, both in the declaration of the variables `A` and `B`, and in the declaration of `=`.

The chop operator of DC is defined semantically as shown below. If the interval `i` is a 2-tuple `(b,e)`, then `proj_1(i)` returns `b`, and `proj_2(i)` returns `e`. The modal operators can easily be defined in terms of the chop operator:

```
^(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)))

<>(A) : IForm = true^A^true
[](A) : IForm = not <>not A
```

Note that we use `^` as the encoded form of the chop operator since semicolon is a reserved operator in PVS.

Duration measures are given as uninterpreted functions and length is defined as in Section 2.

```
dur(p) : IPosTime
len : IPosTime = dur(tt)
```

where `p` is a variable of type `State`.

Remark: Note the difference between the declarations of `^` and `dur`. The `^` operator is an interpreted function, whereas `dur` is an uninterpreted function. Since `^` does not have a complete axiomatisation, its semantical meaning is explicitly defined. The `dur` operator, on the other hand, has a complete axiomatisation so it is possible to axiomatize its meaning. There are pragmatic reasons for axiomatizing the `dur` operator and proving the soundness would require a formalization of the Riemann integral which may require a significant amount of work.

The definitions of $\lceil p \rceil$ ("almost everywhere p") and $\lceil\ \rceil$ ("point interval") are obtained from those of the DC operators in the expected manner. Let `pr(p)` represent $\lceil p \rceil$ and `empty` represent the point interval $\lceil\rceil$:

```
pr(p) : IForm = ((dur(p) = len) /\ (len > n(0)));
empty : IForm = (len = n(0));
```

Quantifiers

PC/DC allows quantifiers over any domain. This effect is obtained by including the quantifier definitions in a parametric theory with the type of the bound variables as the parameter. Let `fall` and `ex` represent universal and existential quantifiers. Then:

```
dcp_quant  [ t: TYPE ]: THEORY

  BEGIN
  ...
  fall(A:[t -> IForm])(i:Interval) : bool = (FORALL (r: t): A(r)(i));
  ex(A:[t -> IForm])(i: Interval) : bool = (EXISTS (r:t) : A(r)(i));
  ...
  END dcp_quant
```

4.2 Rewrite Rules

Apart from automatically expanding definitions of operators, a number of rewrite rules are used in the transformation of formulas into their semantical definitions. These rewrite rules are often used within the strategies defining the DC proof rules and are typically not visible to the user. Examples of such rewrite rules are:

```
  BoxTrue: LEMMA [](true) = true;
  BoxFalse: LEMMA [](false) = false;

  DiamondTrue: LEMMA <>(true) = true;
  DiamondFalse: LEMMA <>(false) = false;

  BoxConj: LEMMA [](A /\ B) = ([]A /\ []B)
```

4.3 Inference Rules

The inference rules used in the DC proofs fall into two groups: the rules of first-order logic and those specific to DC. The former rules are not explicitly stated. They are used indirectly by rewriting top-level DC formulas to PVS Boolean formulas so that they can be handled by the proof commands and strategies of PVS.

Almost all of the rules specific to DC are declared and proved with respect to the semantics. For instance, the monotonicity rule of DC is declared as:

```
  A,B,C,D: VAR IForm

  mono: LEMMA rule(
              seq(Gamma,[](A=>C) + Delta), seq(Gamma,[](B=>D) + Delta),
              seq(Gamma * A^B, C^D + Delta))
```

where `rule` denotes the skeleton of an inference rule and `seq` the sequent. The declarations for `rule` and `seq` are shown below.

```
X,Y,Z: VAR bool
i: VAR Interval

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

+(A,B)(i) : bool = (A(i) OR B(i));
*(A,B)(i) : bool = (A(i) AND B(i));

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

Note that the sequent representation given by `seq` contains an implicit universal quantification over all intervals.

The derived rules (such as `mono`) are proved as a regular PVS lemma. The operators are expanded out into their semantic definitions which introduces propositional operators and quantifiers over intervals and interval endpoints. The lemma is then Skolemized and simplified and the goal sequent is proved as a regular first-order sequent.

Note that since the duration operator has not been given semantic meaning in the encoding, the induction principles and the duration rules ($Ax1 - Ax4$) are treated as PVS axioms. They have therefore not been derived from the semantics in the manner described above.

4.4 Strategies

Strategies are used to form "intelligent" DC proof commands, that is, 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. Firstly, they allow users to apply the proof rules without requiring that they understand the semantical details. Secondly, they allow the user to apply inference rules without supplying much in the way of explicit guidance. As an example, the proof command applying the monotonicity of chop rule should (1) identify the first antecedent formula of the goal sequent that is a chopped formula, (2) search for a consequent formula which also has an outermost chop, and (3) instantiate and apply the inference rule appropriately using the gathered information.

A PC/DC proof command is a PVS strategy. Using the existing functionality of the primitive commands and strategies of PVS, most of the DC inference rules can be applied by providing only the most essential information. PVS strategies are essentially macros of primitive inferences. They do not in any way interfere with the prover internals and will therefore not jeopardize the soundness of the encoding or the proofs.

Each PC/DC proof command is divided up into 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.

For instance, the `mono` command applying the monotonicity rule is declared using the PVS strategy template consisting of the strategy name, arguments, body, help string, and the message displayed when using the strategy:

```
(defstep mono (&optional (sformnuml *)(sformnumr *))
  (let ((numsl (find-dc-chop-num sformnuml '-)))
    (if (car numsl)
        (let (<search phase begin> ...
                (numsr (find-dc-chop-num sformnumr '+))
                ... <search phase end>
                (insts (list "A" a "B" b "C" c "D" d "Delta" del "Gamma" g)))
          (apply-dc-rule$ "mono" insts))
        (skip-msg "Could not find matching DC formula")))
  "(mono &optional <fnumsl[*]> <fnumsr[r]>): Applies monotonicity to chopped
   formula <fnumsl> of the premiss and <fnumsr> of the consequent."
  "Applying monotonicity")
```

In the above strategy, `find-dc-chop-num` searches the formula for chopped formulas, and `skip-msg` prints a message to the user and leaves the goal sequent unchanged. The strategy `apply-dc-rule` contains the common application phase:

```
(defstep apply-dc-rule (name insts)
  (then* (lemma name)
         (expand "rule" -1)
         (expand "seq" -1)
         (expand "+" -1)
         (expand "*" -1)
         (quant? * insts)
         (do-rewrite -1)
         (split)
         (quant-interval)
         (assert)
         (skosimp*)
         (hide-index))
  "(apply-dc-rule <name> <terms>): Applies the DC rule <name> with the
```

```
  instantiations <terms>"
 "Applying the DC rule ~a")
```

The strategy takes the name of the lemma together with the instantiations necessary. First, the inference rule is brought in using the `lemma` command. Then the definitions used in the inference rule are expanded out (using `expand`) and is instantiated (by `quant?`) using the instantiations that were passed as arguments. The instantiated formula is rewritten into its semantic definition by the `do-rewrite` command and the *split* command splits it into its subgoal sequents. The first subgoal is to prove that the conclusion of the inference rule actually matches the goal sequent. The `quant-interval` step instantiates the universal quantification over intervals in the conclusion with the current interval Skolem constant. The remaining subgoals are the new leaves of the proof, that are waiting to be proved. The `hide-index` step hides the formulas from the original goal sequent and leaves behind only the formulas introduced by the inference rule.

4.5 User Interface

Some effort has gone into creating a convenient user interface. This includes modifying the PVS parser, hiding the details, and customizing the Emacs interface.

Parsing and unparsing permits the use of special operator symbols, which are not permitted in the PVS specification language. A customized parser enables us to obtain a concrete syntax in PC/DC which closely resembles the usual DC syntax. For example, the DC terms $\Box A$, $\Diamond A$, and $\lceil p \rceil$ are represented in PC/DC as `[]A`, `<>A`, and `'p'` respectively.

PVS employs the ERGO parser/unparser generator [27] to generate a PVS parser and unparser. The PC/DC parser and unparser was generated using the same system. ERGO takes a syntax description as input. To generate the PC/DC parser, we modified the PVS syntax description and the interface between the parser and the PVS internals.

Hiding the semantical details (i.e. masking all occurrences of interval indices) was also done by modifying the PVS-parser interface. This turned out to be a fairly trivial task: All occurrences of interval constants and variables are removed in the input to the unparser.

The PVS Emacs interface was modified to distinguish between several parsers[9]. 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.

Since this was the first time that the PVS user interface was being customized to a source logic, the modifications required an understanding of some of the PVS

[9] This work was carried out by Sam Owre.

internals. For future encodings of modal logics with simple semantics, it will be possible to provide an easier mechanism to help customize the user interface.

5 Using PC/DC

We now illustrate the use of the specification language and the PC/DC tool with an example. We use the specification and proof of a simple gas burner found in [5]. Note, that the proof contains an error which was later corrected. The revised proof can be found in [4]. Appendix C contains the details of the PC/DC specification and proof for this example. The PC/DC user guide [33] contains a more detailed description of how to use the system.

5.1 Example: Simple Gas Burner

Consider a possibly leaking gas burner. A safety requirement for the system is that gas only leaks for a twentieth of the time from the burner when monitored for a period of time of 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:

$$Req_1 \mathrel{\widehat{=}} \ell \geq 60 \Rightarrow Safe$$

where

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

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 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:

$$Des_1 \wedge Des_2 \Rightarrow Req_1$$

This is done by showing an invariant from the design decisions:

$$Des_1 \wedge Des_2 \Rightarrow Triple$$

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

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

5.2 The Specification Language

We have already seen how the semantics of DC can be defined within PVS. We now adapt the PVS specification language (by modifying the PVS parser) to serve as a specification language for DC in terms of the semantic definition given before. As in PVS, axioms, theorems and lemmas are grouped together in PC/DC into *theories*. A theory is described by a name followed by the keyword `DCTHEORY` (in place of the PVS keyword `THEORY`) and ends with the keyword `END` followed by the name. A DC theory is kept in a file with a `.dc` file extension. The system uses the filename extension to distinguish DC theories and formulas from the original PVS ones. No further reference to the DC definitions are required. In contrast to a PVS theory, a DC theory will automatically import all the definitions shown earlier as part of the semantic definition. The skeleton of a typical PC/DC theory is shown below:

```
mytheory : DCTHEORY
  BEGIN

    IMPORTING <names of importing theories>

    <type, variable and function declarations:>
    x: VAR Time
    Leak : State
    ...
    <axiom and lemma declarations>
    Lem1 : LEMMA ... <body> ...
    ...
  END mytheory
```

Constants, variables and functions are declared as in PVS. The predefined types available are shown in Figure 1. As in PVS, new types (and subtypes of existing types) may be declared. Function names must be alpha-numerical (or include underscores). Axioms and lemmas are declared exactly as in PVS, but are expected to be of the type `IForm` instead of `bool`. Predefined operators (see Figure 1) may be used to build DC formulas.

The specification language is illustrated in Figure 2, which contains an example specification of the simple gas burner. The `IMPORTING` clause in the figure refers to a previously defined theory `dcp_thms` of useful theorems. In order to use these theorems, this theory must be imported. Next, a couple of variables as well as the state `Leak` are declared. `Leak` is the only state variable in the example. 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 main proof obligation. The other lemmas are used in this proof. In general, everything

Syntax	Operator
`dur`	$\int$
`len`	ℓ
`;`	;
`'p'`	$\lceil p \rceil$
`/\`	$\wedge$
`\/`	$\vee$
`~`	$\neg$
`=>`	$\Rightarrow$
`<=>`	$\Leftrightarrow$
`-->`	$\rightarrow$
`-:T:->`	$\xrightarrow{T}$

Type	Declaration
`Time`	$\mathbf{R}$
`PosTime`	$\mathbf{R}_{+}$
`NZPosTime`	$\mathbf{R}_{+} \setminus \{0\}$
`State`	$\mathit{Time} \rightarrow \mathbf{Bool}$
`Interval`	$\mathit{Time} \times \mathit{Time}$
`ITime`	$\mathit{Interval} \rightarrow \mathit{Time}$
`IPosTime`	$\mathit{Interval} \rightarrow \mathit{PosTime}$
`NZIPosTime`	$\mathit{Interval} \rightarrow \mathit{NZPosTime}$

Figure 1: Predefined PC/DC operators and types, and their meaning

that is declared before a lemma or an axiom can be used in its proof. The comments are preceded by `%`.

5.3 Using the Prover

The look and feel of the PC/DC user interface is identical to the PVS user interface. All system and prover commands from PVS can be still be used. Extra prover commands have been added to PVS to customize it to DC proofs. These commands include commands for applying most of the specific DC proof rules. They are applied in a manner very similar to the PVS commands.

Figure 3 shows a proof step illustrating the use of the prover. 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:

$$\begin{array}{lll} & T\,;\,T & \mathit{defs} \\ \Rightarrow & (30\textstyle\int \mathit{Leak} \leq \int 1)\,;\,(30\int \mathit{Leak} \leq \int 1) & \mathit{Dur\text{-}Preserve} \\ \Rightarrow & (30\textstyle\int \mathit{Leak} \leq \int 1) & \mathit{defs} \\ \Rightarrow & T & \end{array}$$

As before, we refrain from writing that the binary predicate $30x \leq y$ is monotone in its arguments. The PC/DC proof reflects the same level of reasoning. 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` correspond to basic PVS proof steps. The `flatten` command breaks the implication and `assert` applies the built-in decision procedures. The `expand`

```
dcgasburner: DCTHEORY
BEGIN

  IMPORTING dcp_thms

  X: VAR IForm
  Leak: State
  p: VAR State

  %%% DEFINITIONS %%%

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

  Req_1: 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')

  %%% SUBLEMMAS FOR lem1 %%%
       ...

   lem1: LEMMA Des_1 /\ Des_2 => Triple

   %%% SUBLEMMAS FOR lem2 %%%

      help2: LEMMA T;T => T

      lem2_1: LEMMA (dur(Leak)<=len/n(30));(dur(Leak)<=n(1))/\ len>=n(60) => Safe

      lem2_2: LEMMA Des_1 /\ T;'Leak' /\ (len >= n(60)) => Safe

      lem2_3 : LEMMA Des_1 /\ T;'Leak';'not Leak' /\ (len >= n(60)) => Safe

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

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

  END dcgasburner
```

Figure 2: Excerpts of the Gas Burner Specification in PC/DC.

command 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)`.

```
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.
```

Figure 3: Example Proof

The commands `then*` and `apply` invoke PVS strategies. 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 and makes it one proof step.

`dur-preserve` is a complex PC/DC proof command which applies the duration preservation proof rule. A significant amount of work is carried out by this strategy. First, the sequent is searched for a chopped formula on the left hand side, which has identical predicates of two durations on each subinterval. Next the proof rule is brought in as a lemma, instantiated with the appropriate predicate and arguments and simplified, which provides two subgoals. One is to show that the predicate satisfies the monotonicity requirement and the second to continue the proof with the predicate satisfied on the full interval. The latter is trivially satisfied, since the same formula now appears on both sides of the sequent. This leaves the former proof obligation, which is returned by the strategy. In this case the proof obligation is trivially satisfied and can by determined by the decision procedures, which are invoked by `assert`.

The `dur-preserve` command is an example of a proof command, where underlying strategies alleviate the user from doing trivial computations. We believe that 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. Furthermore, the example also shows the benefit of using PVS with its tight integration of decision procedures as the semantic framework. In verification of real-time systems, arithmetic reasoning about durations of intervals plays a dominant role.

6 Discussion

We have implemented a mechanized proof assistant prototype in PVS. The proof assistant was obtained 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. 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 work has resulted in a tool which seems convenient to use and, hopefully, easy to learn for the DC expert. The polished user interface and the high degree of automation brings reasoning within PC/DC closer to pen-and-paper style reasoning.

The duration operator has not been semantically encoded; instead we have introduced (a complete set of) 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. As the first step, a system specification of an air traffic monitoring system [25] is currently being checked using the tool. 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 PVS 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 conversion, 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. Anders Ravn, Hans Rischel, and Hans Henrik Løvengreen provided us with many useful comments on the draft. Friedrich von Henke (of SRI and University of Ulm) and Jack Goldberg (SRI) also reviewed draft versions of the report and suggested numerous improvements.

Jens U. Skakkebæk: I wish to thank Natarajan Shankar, Sam Owre and John Rushby for many fruitful discussions and for making my visit at SRI's Computer Science Laboratory (CSL) possible. The rest of the CSL Formal Methods Group: Patrick Lincoln, Mandayam Srivas, David Cyrluk, and Judy Crow have provided significant inputs and a stimulating working environment. Narciso Marti-Oliet (CSL) educated me on the topic of logic frameworks and Sreeranga Rajan (visiting from Univ. of British Columbia) shared with me the realities of life as a Ph.D. student.

Many thanks go also to Director Mark Moriconi and the staff of the SRI International Computer Science Laboratory for their hospitality during my stay.

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. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[4] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. Revised June 3, 1992.

[5] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, December 1991.

[6] 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*, pages 71–80. IFIP/North-Holland, 1993.

[7] J. Michael Dunn. Relevance logic and entailment. In Dov M. Gabbay and Franz Guenthner, editors, *Handbook of Philosophical Logic–Volume III: Alternatives to Classical Logic*, volume 166 of *Synthese Library*, chapter III.3, pages 117–224. D. Reidel Publishing Company, Dordrecht, Holland, 1985.

[8] U. Engberg, P. Grønning, and L. Lamport. Mechanical verification of concurrent systems with TLA. In *Computer Aided Verification 1992*, volume 663 of *Lecture Notes in Computer Science*, 1992.

[9] M. Engel, A.P. Ravn, and H. Rischel. Timed-z specification language for real-time systems. ProCoS II, ESPRIT BRA 7071 DTH ME /0.1, Department of Computer Science, Technical University of Denmark, September 1993.

[10] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* Harper and Row, New York, 1986.

[11] Stephen J. Garland and John V. Guttag. LP: The Larch prover. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction (CADE)*, volume 310 of *Lecture Notes in Computer Science*, pages 748–749, Argonne, IL, May 1988. Springer-Verlag.

[12] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[13] D. Goldschlag. Mechanizing unity. In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods.* North-Holland, 1990.

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

[15] Michael 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.

[16] 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.

[17] 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.

[18] A. G. Hamilton. *Logic for Mathematicians.* Cambridge University Press, 1978.

[19] 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.

[20] M. R. Hansen and Zhou Chaochen. Lecture notes on the logical foundation of duration calculus. In Manuscript, 1993.

[21] 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, Princeton University, New Jersey, March 1992*, 1992.

[22] R. Harper, F. Honsell, and G. D. Plotkin. A framework for defining logics. In *IEEE Symposium on Logic in Computer Science*, Ithaca, NY, 1987.

[23] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–583, 1969.

[24] C. A. R. Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[25] Recep Inal. Modular specification of real-time systems, 1994. To appear in *Proc. of IEEE Workshop on Real-Time Systems.*

[26] Leslie Lamport. The Temporal Logic of Actions. research report 79, Digital Equipment Corporation, Systems Research Center, December 1991.

[27] 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.

[28] D. Miller and G. Nadathur. Higher-order logic programming. In *IEEE Symposium on Logic Programming*, 1986.

[29] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Transactions on Computers*, 18(2):10–19, 1985.

[30] 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, February 1993. Three volumes.

[31] L. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[32] 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. also ProCoS Document [DTH APR 19/1].

[33] J. U. Skakkebæk, S. Owre, and N. Shankar. The duration calculus proof checker: A user guide, 1993. In Manuscript.

[34] J.U. Skakkebaek, A.P. Ravn, H. Rischel, and Zhou Chaochen. Development of provably correct systems. In *IEEE Workshop on Real-Time Systems*, June 1991.

[35] Y. Venema. A modal logic for chopping intervals. *Journal of Logic Computation*, 1(4):453–476, 1991.

[36] L. Wallen. *Automated Deduction in Nonclassical Logics*. MIT Press, 1990.

A Proof System

A.1 Basic Inference Rules

Structural Rules

The structural rules allow the sequent to be modified or rearranged. A sequent may be weakened by introduction of new formulas in the conclusion, as follows:

$$\frac{\Gamma_1 \vdash \Delta_1}{\Gamma_2 \vdash \Delta_2}\ \textbf{Weakening} \qquad \text{if } \Gamma_1 \subseteq \Gamma_2 \text{ and } \Delta_1 \subseteq \Delta_2$$

The structural rules of contraction and exchange shown below are built into the sequent calculus since sequents are represented by means of *sets* of formulas.

$$\text{Contraction:} \quad \frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\ \mathbf{C}\vdash \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\ \vdash\mathbf{C}$$

$$\text{Exchange:} \quad \frac{\Gamma_1, B, A, \Gamma_2 \vdash \Delta}{\Gamma_1, A, B, \Gamma_2 \vdash \Delta}\ \mathbf{X}\vdash \qquad \frac{\Gamma \vdash \Delta_1, B, A, \Delta_2}{\Gamma \vdash \Delta_1, A, B, \Delta_2}\ \vdash\mathbf{X}$$

Propositional Rules

Let $A \equiv B$ if A and B are syntactically equivalent modulo renaming of bound variables, *e.g.*,$(\forall x. \int P = x) \equiv (\forall y. \int P = y)$. The axiom rule then has the form:

$$\frac{}{\Gamma, A \vdash B, \Delta}\ \textbf{Ax} \qquad \text{where } A \equiv B$$

The usual rules for propositional connectives are shown below.

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\ \wedge\vdash \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}\ \vdash\wedge$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}\ \vee\vdash \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}\ \vdash\vee$$

$$\frac{\Gamma, B \vdash \Delta \qquad \Gamma \vdash A, \Delta}{\Gamma, A \Rightarrow B \vdash \Delta}\ \Rightarrow\vdash \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta}\ \vdash\Rightarrow$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\ \neg\vdash \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\ \vdash\neg$$

The DC sequent calculus includes the well-known cut rule, which isn used in the proof checker to carry out a proof by cases.

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma \vdash A, \Delta}{\Gamma \vdash \Delta}\ \textbf{Cut}$$

Equality

The rules for equality shown below also apply to equivalences (also represented by equality) between duration formulas. The rules for equivalence can be derived from the propositional rules. Let an *expression* denote either a term or a formula. We define equality on expressions. Let a and b be expressions:

$$\frac{}{\Gamma \vdash a = b, \Delta}\,\textbf{Refl} \quad \text{if } a \equiv b$$

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

The following rules define top-level replacement:

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

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

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

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

Quantifiers

Since variables and terms can occur in modal contexts, the quantifier rules here differ slightly from those of first-order logic. These rules are

$$\frac{\Gamma, A\{x \leftarrow t\} \vdash \Delta}{\Gamma, (\forall x : A) \vdash \Delta}\,\forall \vdash \qquad \frac{\Gamma \vdash A\{x \leftarrow a\}, \Delta}{\Gamma \vdash (\forall x : A), \Delta} \vdash \forall$$

$$\frac{\Gamma, A\{x \leftarrow a\} \vdash \Delta}{\Gamma, (\exists x : A) \vdash \Delta}\,\exists \vdash \qquad \frac{\Gamma \vdash A\{x \leftarrow t\}, \Delta}{\Gamma \vdash (\exists x : A), \Delta} \vdash \exists$$

where

- a is constant not occurring in the conclusion of the $\vdash \forall$ and $\exists \vdash$ rules; and
- in the $\forall \vdash$ and $\vdash \exists$ rules, either t is a duration free term or all occurrences of x in A are in chop free subformulas.

Rules for Durations

Four axioms for $\int$ are presented below. These axioms are fairly self-evident.

$$\frac{}{\vdash\ \int 0 = 0}\ \textbf{Ax1}$$

$$\frac{}{\vdash\ \int p \geq 0}\ \textbf{Ax2}$$

$$\frac{}{\vdash\ \int p + \int q = \int(p \wedge q) + \int(p \vee q)}\ \textbf{Ax3}$$

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

where $r, s \geq 0$.

Rules for Chop

Below we define a number of inference rules for proving properties of chop formulas. The chop operator is monotonic in that if we have $\Box A \Rightarrow C$ and $\Box B \Rightarrow D$, then $A\,;\,B \Rightarrow C\,;\,D$. The chop operator is also associative, and distributes over disjunction:

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

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

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

The following rules relate existential quantification with the chop operator.

$$\frac{}{\vdash\ (\exists x : A)\,;\,B \Leftrightarrow (\exists x : A\,;\,B)}\ \exists ; \qquad \frac{}{\vdash\ B; (\exists x : A) \Leftrightarrow (\exists x : B\,;\,A)}\ ; \exists$$

where x does not occur free in B.

The following rules describe the propagation of duration free formulas out of the scope of a chop operator.

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

where C is duration free.

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

$$\frac{}{false; A \vdash}\ \textbf{false ; }\vdash \qquad \frac{}{A; false \vdash}\ \textbf{; false }\vdash$$

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

Chop distributes conditionally over conjunction:

$$\frac{}{\vdash\ (\exists x : A\,;\,(C \wedge \ell = x) \wedge B\,;\,(D \wedge \ell = x)) \Leftrightarrow (A \wedge B)\,;\,(C \wedge D)}\ \textbf{split}$$

Induction

The condition of finite variability of states allows for induction over behaviors of states: Given a state P, initially in a non-empty 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 an empty 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 $R(X)$ denote be the invariant, i.e. an induction formula in which X occurs. Also choose a state P over which to induct. Then:

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

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

A.2 Derived Rules

The rules shown below are convenient for doing proofs. They can be derived from the basic rules.

Rules for Modal Operators

The following inference rules are characteristic of S4 modal systems [36].

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:

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

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

Applying Monotonicity

The monotonicity of chop is used in certain typical ways that can be encapsulated into single inference rules. In most cases monotonicity is only applied to one subinterval of a chop, whereas the general rule of monotonicity applies to both subintervals. The following variations on monotonicity turn out to be convenient to use:

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

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

Duration Preservation

Often identical formulas on adjacent subintervals can be shown to hold on the full interval. It can be shown that in general, all expressions preserving the relationship between durations on intervals and which hold on any adjacent subintervals also hold on the full interval.

Let $\otimes$ be a relation such that:

$$r_1 \otimes s_1 \wedge r_2 \otimes s_2 \Rightarrow (r_1 + r_2) \otimes (s_1 + s_2)$$

for any $r_1, s_1, r_2, s_2 \in \mathbf{R}$ (e.g. $\otimes\ is\ <$). Then:

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

B DC Semantics in PVS

B.1 Basic Definitions

The semantic encoding is defined in five PVS files. Instead of importing each file separately, it is sufficient to import the `dcprover` theory shown below. Its only purpose is to import the right DC definitions, axioms, and proof rules. The `dcprover` theory is implicitly imported when typechecking a `DCTheory` in PC/DC.

```
dcprover : THEORY

  BEGIN

    IMPORTING dcp_defs,dcp_equality,dcp_not_rewrite,dcp_also_rewrite,
              dcp_quant,dcp_context_free,dcp_cf_reverse,dcp_rules

  END dcprover
```

The theory *dcp_defs* declares the basic operators and defines their semantics. Note that all operators declared in this theory are automatically rewritten during a PC/DC proof. `cf` is a function used to determine duration free subformulas. It is used within a strategy and should not be visible to the user.

```
dcp_defs : THEORY

  BEGIN

    Time : TYPE = real
    NZTime : TYPE = x:Time|x/=0 CONTAINING 1
    PosTime : TYPE = x:Time|x>=0 CONTAINING 1
    NZPosTime : TYPE = x:PosTime|x/=0 CONTAINING 1

    State : TYPE = [Time->bool];
    SReal : TYPE = [Time -> Time]

    Interval: TYPE = (b,e):[Time,Time]|b<=e CONTAINING (1,1)
    i,i1,i2,i3: VAR Interval

    IForm : TYPE = [Interval -> bool];
    ITime : TYPE = [Interval -> Time];
    NZITime : TYPE = [Interval -> NZTime];
    IPosTime : TYPE = [Interval -> PosTime];
    NZIPosTime : TYPE = [Interval -> NZPosTime];

    A,B,C,D: VAR IForm
    Gamma,Delta,Gamma1,Delta1: VAR IForm
    t1,t2: VAR ITime
    s1,s2: VAR SReal
    r: VAR Time
    nt: VAR NZTime
    nt1,nt2: VAR NZITime
    p,q: VAR State
```

```
  /\(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);

  >(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);

  *(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);

  >(nt1,nt2)(i): bool = nt1(i) > nt2(i);
  <(nt1,nt2)(i): bool = nt1(i) < nt2(i);
  >=(nt1,nt2)(i): bool = nt1(i) >= nt2(i);
  <=(nt1,nt2)(i): bool = nt1(i) <= nt2(i);

  nzreal_mult_closed: AXIOM FORALL (x,y: nzreal): x * y /= 0;
  nzreal_div_closed: AXIOM FORALL (x,y: nzreal): x / y /= 0;

  *(nt1,nt2)(i): NZTime = nt1(i) * nt2(i);
  /(nt1,nt2)(i): NZTime = nt1(i) / nt2(i);

  valid(A) : bool = (FORALL i: A(i))

  % Type coercion

   n(r)(i) : Time = r

  % Context-freeness

  f: VAR bool

  cf(r)(i) : Time = r
  cf(nt)(i) : NZTime = nt
  cf(f)(i) : bool = f

  % Semantics of states

  tt(r) : bool = true;
  ff(r) : bool = false;
  /\(p,q)(r) : bool = p(r) AND q(r);
  \/(p,q)(r) : bool = p(r) OR q(r);
  not (p)(r) : bool = not p(r);

  >(s1,s2)(r): bool = s1(r) > s2(r);
  <(s1,s2)(r): bool = s1(r) < s2(r);
  >=(s1,s2)(r): bool = s1(r) >= s2(r);
  <=(s1,s2)(r): bool = s1(r) <= s2(r);
  =(s1,s2)(r): bool = (s1(r) = s2(r));

  *(s1,s2)(r): Time = s1(r) * s2(r);
  +(s1,s2)(r): Time = s1(r) + s2(r);
  -(s1,s2)(r): Time = s1(r) - s2(r);

  svalid(p) : bool = (FORALL r: p(r));

END dcp_defs
```

Operators which should not be expanded automatically during a proof, are kept in the `dcp_not_rewrite` theory shown below. For instance, many proofs will use proof rules manipulating formulas preceded by the `[]` operator, without expanding `[]` into its definition `not <> not`. `^` below corresponds to chop, and `pr(p)` to $\lceil p \rceil$. During a proof, however, `^` will be unparsed as `;` and `pr(p)` as `'p'`.

Note, that `dur` is for now declared as an uninterpreted function without a semantic definition. Instead of providing the semantic meaning and proving soundness of the four duration axioms from this definition, the four axioms have been encoded as PVS axioms.

```
dcp_not_rewrite : THEORY

  BEGIN

    IMPORTING dcp_defs,dcp_equality

    p,q: VAR State
    A,B : VAR IForm
    i: VAR Interval
    r: VAR Time

    =>(p,q) : State = (not p \/ q);
    <=>(p,q) : State = (p => q) /\ (q => p);

    =>(A,B): IForm = (not A \/ B);
    <=>(A,B): IForm = (A => B) /\ (B => A);

    dur(p) : IPosTime
    len : IPosTime = dur(tt)

    true : IForm = (len >= n(0))
    false : IForm = (not true)

    pr(p) : IForm = dur(p) = len /\ (len > n(0));
    empty : IForm = (len = n(0));

    ^(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)))

    <>(A) : IForm = true^A^true
    [](A) : IForm = not <>not A

    leadsto(A,B) : IForm = ((A^TRUE) => (A \/ A^B^TRUE))

    tprogress(A,r,B) : IForm =
               []((A/\len=n(r))^(len>n(0)) => (len=n(r))^B^TRUE)

  END dcp_not_rewrite
```

Equality and the if-construct are defined in a parametric theory. The parameter `t` is a type variable, which is used to define the operators. The equality operator $=$ can be used to express equality of two arguments of type `t`. Also, if-constructs with leafs of type `t` are defined.

```
dcp_equality [t : TYPE]: THEORY

  BEGIN

    IMPORTING dcp_defs

    a : VAR t
    b : VAR bool
    A,B : VAR [Interval -> t]
    C : VAR IForm
    p,q : VAR [Time -> t]
    p1 : VAR State
    r : VAR Time
    i : VAR Interval;

    =(p,q)(r) : bool = (p(r) = q(r));

    =(A,B)(i) : bool = (A(i) = B(i));

    if(b,A,B)(i): t = IF b THEN A(i) ELSE B(i) ENDIF
    if(C,A,B)(i): t = IF C(i) THEN A(i) ELSE B(i) ENDIF

    if(b,p,q)(r): t = IF b THEN p(r) ELSE q(r) ENDIF
    if(p1,p,q)(r): t = IF p1(r) THEN p(r) ELSE q(r) ENDIF

  END dcp_equality
```

Quantifiers are declared in the `dcp_quant` theory. A universally quantified DC formula $\forall x : A$ is encoded in PVS as the formula `fall(lambda(x):A)`. An existentially quantified DC formula $\exists x : A$ is encoded in PVS as the formula `ex(lambda(x):A)`. A user will, however, not see these encodings. The parsing/unparsing mechanism will present the user with the pretty-printed version. So the formula `ex(lambda(x:Time) : len = n(x))` is unparsed as the formula `(EXISTS (x:Time) : len = n(x))`. The system will likewise understand the formula `(EXISTS (x:Time) : len = n(x))` as input and translate it into its correct PVS representation.

The four proof rules in the last part of the theory are used by a strategy to move existential quantifiers in and out of chopped subintervals. This proof rules have been proven correct with respect to the meaning of `ex` and `^` .

```
dcp_quant  [ t: TYPE ]: THEORY

  BEGIN

    IMPORTING dcp_not_rewrite

    fall(A:[t -> IForm])(i:Interval) : bool = (FORALL (r: t): A(r)(i));
    ex(A:[t -> IForm])(i: Interval) : bool = (EXISTS (r:t) : A(r)(i));

    fall(A:[t -> bool]) : bool = (FORALL (r: t): A(r));
    ex(A:[t -> bool]) : bool = (EXISTS (r:t) : A(r));

    F: VAR [t->IForm]
    r,s: VAR t
    A,B : VAR IForm
```

```
    i : VAR Interval

    exists_chop: LEMMA (ex(F)^B) = (ex(LAMBDA (r:t): F(r)^B))
    exists_chop_dc: LEMMA (ex(F)^B)(i) = (EXISTS (r:t): (F(r)^B)(i))

    chop_exists: LEMMA (A^ex(F)) = (ex(LAMBDA (r:t): A^F(r)))
    chop_exists_dc: LEMMA (A^ex(F))(i) = (EXISTS (r:t): (A^F(r))(i))

  END dcp_quant
```

Very often, it is convenient to lift elements of an existing type to the range of a function type. In PC/DC this is, for instance, found in the definition of `ITime`. We want to use, say `3`, as the function that always return the value 3 when applied to a given interval. And still we want to be able to use the decision procedures on natural numbers to determine that `3 + 3 > 3` on a given interval.

However, the PVS decision procedures can at present not be applied directly to such expressions. Instead we define a type coercion operator `n`. `n(3)` is the identity function that given an interval `i` will return the natural number `3`. The DC formula $\ell > 3$ is represented in PVS as `len > n(3)`, where `3` is a PVS natural number.

`n` is a declared in parametric theory, which a user may instantiate with any type needed.

```
dcp_coercion [t : Type ]: THEORY

  BEGIN

    IMPORTING dcp_defs

    n(a:t)(i:Interval) : t = a

  END dcp_coercion
```

B.2 Rewrite Rules

The encoding includes a number of simplifying rewrite rules, which are applied automatically during a PC/DC proof. They are intended to be used to apply trivial propositional simplifications of DC formulas.

```
dcp_also_rewrite : THEORY

  BEGIN

    IMPORTING dcp_quant

    A,B : VAR IForm

    i : VAR Interval
    r,s : VAR Time
    nr : VAR NZTime
    p,q : VAR State

    State_Impl : LEMMA (p=>q)(r)  = (p(r) IMPLIES q(r));
    State_Iff : LEMMA (p<=>q)(r)  = (p(r) IFF q(r));

    LemTrue : LEMMA true(i) = true;
    LemFalse : LEMMA false(i) = false;

    BoxTrue: LEMMA [](true) = true;
    BoxFalse: LEMMA [](false) = false;

    DiamondTrue: LEMMA <>(true) = true;
    DiamondFalse: LEMMA <>(false) = false;

    BoxConj: LEMMA [](A/\B) = ([]A /\ []B)

    IForm_Impl : LEMMA (A => B)(i) = (A(i) IMPLIES B(i));
    IForm_Iff : LEMMA (A <=> B)(i) = (A(i) IFF B(i));

    not_not: LEMMA (not (not p)) = p;
    not_true:  LEMMA (not tt) = ff;
    not_false:  LEMMA (not ff) = tt;
    and_not: LEMMA (p /\ (not p)) = ff;
    and_drop: LEMMA (p /\ p) = p;
    or_not: LEMMA (p \/ (not p)) = tt;
    or_drop: LEMMA (p \/ p) = p;
    and_true:  LEMMA (p /\ tt) = p;
    true_and:  LEMMA (tt /\ p) = p;
    or_false:  LEMMA (p \/ ff) = p;
    false_or:  LEMMA (ff \/ p) = p;

    int_not_not:  LEMMA (not (not A)) = A;
    int_not_true:  LEMMA (not dcp_not_rewrite.TRUE) = FALSE;
    int_not_false:  LEMMA (not dcp_not_rewrite.FALSE) = TRUE;
    int_and_not:  LEMMA (A /\ (not A)) = FALSE;
    int_and_drop: LEMMA (A /\ A) = A;
    int_or_not:   LEMMA (A \/ (not A)) = TRUE;
    int_or_drop:  LEMMA (A \/ A) = A;
    int_and_true:  LEMMA (A /\ TRUE) = A;
    int_true_and:  LEMMA (TRUE /\ A) = A;
    int_and_false:  LEMMA (A /\ FALSE) = FALSE;
    int_false_and:  LEMMA (FALSE /\ A) = FALSE;
    int_or_false:  LEMMA (A \/ FALSE) = A;
    int_false_or:  LEMMA (FALSE \/ A) = A;
    int_or_true:  LEMMA (A \/ TRUE) = TRUE;
    int_true_or:  LEMMA (TRUE \/ A) = TRUE;

    n1 : LEMMA n(r) + n(s) = n(r + s)
    n2 : LEMMA n(r) - n(s) = n(r - s)
    n3 : LEMMA n(r) * n(s) = n(r * s)
    n4 : LEMMA n(r) / n(nr) = n(r / nr)
```

```
  END dcp_also_rewrite
```

As described earlier, a duration free formula true on a subinterval may be moved in and out of a chopped formula. A duration free formula can be described recursively as follows:

- Any number is a duration free expression,
- An arithmetic expression is duration free, if both its arguments are duration free,
- A comparative expression is duration free, if both its arguments are duration free,
- A negated formula is duration free if its argument is duration free,
- A disjunctive formula is duration free if both disjuncts are duration free.

This definition is encoded in PVS as a number of rewrite rules. Whenever the user want to apply a rule involving duration free formulas, the rewrite rules shown in the theory `dcp_context_free` are applied repeatedly until no more rewrites occur. After the proof rule has been applied, the rewrite rules encoded in the theory `dcp_cf_reverse` are applied. These latter rewrite rules are identical to the former, except the rewriting direction.

```
dcp_context_free : THEORY

  BEGIN

    IMPORTING dcp_defs,dcp_not_rewrite

    r,r1,r2: VAR Time
    ir,ir1,ir2: VAR ITime
    f,f1,f2: VAR bool
    i: VAR Interval
    A,B,C: VAR IForm

    cf1 : LEMMA n(r) = cf(r)

    cf2 : LEMMA cf(r1) * cf(r2) = cf(r1 * r2)
    cf3 : LEMMA cf(r1) + cf(r2) = cf(r1 + r2)
    cf4 : LEMMA cf(r1) - cf(r2) = cf(r1 - r2)
    cf5 : LEMMA (r2 /= 0) IMPLIES (cf(r1) / cf(r2) = cf(r1 / r2))

    cfc1 : LEMMA (cf(r1) <= cf(r2)) = cf(r1 <= r2);
    cfc2 : LEMMA (cf(r1) < cf(r2)) = cf(r1 < r2);
    cfc3 : LEMMA (cf(r1) >= cf(r2)) = cf(r1 >= r2);
    cfc4 : LEMMA (cf(r1) > cf(r2)) = cf(r1 > r2);
    cfc5 : LEMMA (cf(r1) = cf(r2)) = cf(r1 = r2);
```

```
    cfb1 : LEMMA (cf(f1) /\ cf(f2)) = cf(f1 AND f2);
    cfb2 : LEMMA (cf(f1) \/ cf(f2)) = cf(f1 OR f2);
    cfb3 : LEMMA (cf(f1) => cf(f2)) = cf(f1 IMPLIES f2);
    cfb4 : LEMMA (cf(f1) <=> cf(f2)) = cf(f1 IFF f2);
    cfb5 : LEMMA (not cf(f)) = cf(NOT f);

  END dcp_context_free
```

```
dcp_cf_reverse: THEORY

  BEGIN

    IMPORTING dcp_defs,dcp_not_rewrite

    r,r1,r2: VAR Time
    nr,nr1,nr2: VAR NZTime
    ir,ir1,ir2: VAR ITime
    f,f1,f2: VAR bool
    i: VAR Interval

    cf1 : LEMMA cf(r) = n(r)

    cf2 : LEMMA cf(r1 * r2) = cf(r1) * cf(r2)
    cf3 : LEMMA cf(r1 + r2) = cf(r1) + cf(r2)
    cf4 : LEMMA cf(r1 - r2) = cf(r1) - cf(r2)
    cf5 : LEMMA (r2 /= 0) IMPLIES cf(r1 / r2) = (cf(r1) / cf(r2))

    cfc1 : LEMMA cf(r1 <= r2) = (cf(r1) <= cf(r2));
    cfc2 : LEMMA cf(r1 < r2) = (cf(r1) < cf(r2));
    cfc3 : LEMMA cf(r1 >= r2) = (cf(r1) >= cf(r2));
    cfc4 : LEMMA cf(r1 > r2) = (cf(r1) > cf(r2));
    cfc5 : LEMMA cf(r1 = r2) = (cf(r1) = cf(r2));

    cfb1 : LEMMA cf(f1 AND f2) = (cf(f1) /\ cf(f2));
    cfb2 : LEMMA cf(f1 OR f2) = (cf(f1) \/ cf(f2));
    cfb3 : LEMMA cf(f1 IMPLIES f2) = (cf(f1) => cf(f2));
    cfb4 : LEMMA cf(f1 IFF f2) = (cf(f1) <=> cf(f2));
    cfb5 : LEMMA cf(NOT f) = (not cf(f));

  END dcp_cf_reverse
```

B.3 Proof Rules

The DC inference rules described earlier are encoded in PVS as lemmas. The inference rules are encoded either as PVS implications (using the `rule` construct) or as rewrite rules.

`Gamma` and `Delta` are formulas representing a conjunct and disjunct, respectively.

Some rewrite rules are encoded both in a rewritten form and the original form. Associativity of chop is, for example, encoded by the rewrite rules `assoc` and `assoc_dc`. The former can be used to rewrite subformulas within, say a chopped formula, and the latter is used to rewrite top-level formulas, which have already been rewritten into its semantic definition.

```
dcp_rules : THEORY

  BEGIN

    IMPORTING dcp_defs,dcp_not_rewrite,dcp_context_free,dcp_quant

    Gamma,Gamma1,Delta,Delta1: VAR IForm
    A,B,C,D: VAR IForm
    X,Y,Z: VAR bool
    p,q: VAR State
    i: VAR Interval

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

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

    +(A,B)(i) : bool = (A(i) OR B(i));
    *(A,B)(i) : bool = (A(i) AND B(i));

    dur_lemma : LEMMA dur(p)(i) >= 0


    %%%% MONOTONICITY OF CHOP %%%%

    mono: LEMMA rule(
                  seq(Gamma,[](A=>C) + Delta),
                  seq(Gamma,[](B=>D) + Delta),
                  seq(Gamma * A^B, C^D + Delta))

    mono_bang: LEMMA rule(
                  seq(true,A=>C),
                  seq(true,B=>D),
                  seq(A^B, C^D))


    %%%% ASSOCIATIVITY OF CHOP %%%%

    assoc: LEMMA ((A^B)^C) = (A^(B^C))
    assoc_dc: LEMMA ((A^B)^C)(i) = (A^(B^C))(i)

    %%%% DISTRIBUTIVITY OF CHOP %%%%

    disj_chop: LEMMA ((A\/B)^C) = (A^C \/ B^C)
    disj_chop_dc: LEMMA ((A\/B)^C)(i) = ((A^C)(i) OR (B^C)(i))

    chop_disj: LEMMA (A^(B\/C)) = (A^B \/ A^C)
    chop_disj_dc: LEMMA (A^(B\/C))(i) = ((A^B)(i) OR (A^C)(i))


    %%%% CONTEXT FREE FORMULAS %%%%

    f: VAR bool

    free_chop: LEMMA ((cf(f) /\ A)^B) = (cf(f) /\ A^B)
    free_chop_dc: LEMMA ((cf(f) /\ A)^B)(i) = (f AND (A^B)(i))

    free_chop_true: LEMMA (cf(f)^B) = (cf(f) /\ true^B)
    free_chop_true_dc: LEMMA (cf(f)^B)(i) = (f AND (true^B)(i))

    chop_free: LEMMA (A^(cf(f) /\ B)) = (cf(f) /\ A^B)
```

```
chop_free_dc: LEMMA (A^(cf(f) /\ B))(i) = (f AND (A^B)(i))

chop_free_true: LEMMA (A^cf(f)) = (cf(f) /\ A^true)
chop_free_true_dc: LEMMA (A^cf(f))(i) = (f AND (A^true)(i))


%%%% FALSE - CHOP %%%%

false_chop: LEMMA rule(seq(Gamma * false^A, Delta))

chop_false: LEMMA rule(seq(Gamma * A^false, Delta))


%%%% DC AXIOMS %%%%

axiom1: AXIOM dur(ff) = n(0)
axiom1_dc: AXIOM dur(ff)(i) = 0

axiom2: AXIOM valid(dur(p)>=n(0))

axiom3: AXIOM dur(p)+dur(q)=dur(p/\q)+dur(p\/q)
axiom3_dc : LEMMA dur(p)(i) + dur(q)(i) = dur(p /\ q)(i) + dur(p \/ q)(i)

r,s : VAR PosTime

axiom4: AXIOM (dur(p)=n(r))^(dur(p)=n(s)) = (dur(p)= n(r)+n(s))
axiom4_dc: LEMMA ((dur(p)=n(r))^(dur(p)=n(s)))(i) = (dur(p)(i)= r+s)

lenax : AXIOM len(i) = proj_2(i) - proj_1(i)


%%%% INDUCTION %%%%

R: VAR [IForm -> IForm]

induct_chop : AXIOM rule(
              seq(Gamma,[]R(empty) + Delta),
              (FORALL (X:IForm):
              seq(Gamma * []R(X), R(X\/pr(p)^X\/pr(not p)^X) + Delta)),
              seq(Gamma,[]R(true) + Delta))

chop_induct : AXIOM rule(
              seq(Gamma,[]R(empty) + Delta),
              (FORALL (X:IForm):
              seq(Gamma * []R(X), R(X\/X^pr(p)\/X^pr(not p)) + Delta)),
              seq(Gamma,[]R(true) + Delta))


%%%% CHOP - EMPTY %%%%

empty_chop: LEMMA empty^A = A

chop_empty: LEMMA A^empty = A


%%%% RULES FOR MODALITIES %%%%

box_left : LEMMA rule(
              seq(Gamma * A,Delta),
              seq(Gamma * []A,Delta))

box_right : LEMMA rule(
              seq([]Gamma,A + <>Delta),
              seq([]Gamma,[]A + <>Delta))
```

```
  diamond_left : LEMMA rule(
                seq([]Gamma * A, <>Delta),
                seq([]Gamma * <>A, <>Delta))

  diamond_right : LEMMA rule(
                seq(Gamma,A + Delta),
                seq(Gamma,<>A + Delta))

  %%%% APPLY MONOTONICITY %%%%

  chop_mono: LEMMA rule(
                   seq(Gamma,[](B=>C) + Delta),
                   seq(Gamma * A^B * A^C, Delta),
                   seq(Gamma * A^B, Delta))

  chop_mono_bang: LEMMA rule(
                        seq(true,B=>C),
                        seq(Gamma * A^B * A^C, Delta),
                        seq(Gamma * A^B, Delta))

  mono_chop: LEMMA rule(
                  seq(Gamma,[](A=>C) + Delta),
                  seq(Gamma * A^B * C^B, Delta),
                  seq(Gamma * A^B, Delta))

  mono_chop_bang: LEMMA rule(
                        seq(true,A=>C),
                        seq(Gamma * A^B * C^B, Delta),
                        seq(Gamma * A^B, Delta))

  exists_intro : LEMMA  A^B = ex(lambda(r: PosTime):(dur(p)=n(r) /\ A)^B)

  intro_exists : LEMMA A^B = ex(lambda(r: PosTime): A^(dur(p)=n(r) /\ B))

  intro_exists_intro : LEMMA A^B^C = ex(lambda(r: PosTime): A^(dur(p)=n(r) /\ B)^C)

  mono_chop_mono: LEMMA rule(
              seq(Gamma,[](B=>D) + Delta),
              seq(Gamma * A^B^C * A^D^C, Delta),
              seq(Gamma * A^B^C, Delta))

  mono_chop_mono_right: LEMMA rule(
              seq(Gamma,[](B=>D) + Delta),
              seq(Gamma, A^B^C + Delta),
              seq(Gamma, A^D^C + Delta))

  %%%% ADDING DURATIONS  %%%%

  ax4 : LEMMA valid((dur(p)=n(r))^(dur(p)=n(s))<=>(dur(p)=n(r)+n(s)))

  G : VAR [IPosTime,IPosTime -> IForm]
  G1 : VAR [PosTime,PosTime -> bool]
  r1,s1,r2,s2: VAR PosTime
  ir1,is1: VAR IPosTime

  dur_preserve: LEMMA ((FORALL i,ir1,is1: G(ir1,is1)(i) = G1(ir1(i),is1(i))) IMPLIES
                rule(
                (FORALL r1,s1,r2,s2: seq(Gamma * G(n(r1),n(s1)) * G(n(r2),n(s2)),
                                         G(n(r1)+n(r2),n(s1)+n(s2)) + Delta)),
                seq(Gamma * G(dur(p),dur(q)), Delta),
                seq(Gamma * G(dur(p),dur(q))^G(dur(p),dur(q)), Delta)))

END dcp_rules
```

B.4 Strategies

The PC/DC proof commands are implemented as PVS strategies, which are macros of PVS commands. Below we show the strategies which are used in PVS. Each strategy consists of a search phase (a `let` assignment) and an application phase (either a call to `apply-dc-rule` or rewriting using `rewrite`. The search phase uses Lisp utilities to take out all the information of the goal sequent that is needed to instantiate the inference rule. The application phase uses this information to instantiate the inference rule, rewrite it, and simplify the formula.

A PVS strategy is defined by the `defstep` command. It consists of

- the name of the strategy,
- a list arguments,
- a body, which defines the functionality,
- a string of help information for the strategy, and
- a string which is displayed whenever the formula is applied.

```
(in-package 'pvs)

(setq *rewrite-msg-off* t)

(defstep dc-sem ()
  (let ((x (dcsem)))
    (skip))
  "(dc-sem): Turns hiding of semantics on/off."
  "Turning hiding on/off")

(defun dcsem ()
  (if *hide-dc-semantics*
      (setq *hide-dc-semantics* nil)
      (setq *hide-dc-semantics* t)))

(defstep dcinit ()
  (then* (auto-rewrite-theory "dcp_defs")
         (auto-rewrite-theory "dcp_also_rewrite")
         (auto-rewrite-theory "dcp_equality" :defs-only? t)
;        (auto-rewrite-theory "dcp_extensions" :defs-only? t)
         (auto-rewrite! "ex" "fall")
         (expand "valid")(repeat (skolem!)))
  "(dc-init): Initializes the prover for proving DC theorems."
  "Initializing the DC formula")


(defstep dclemma (name &optional subst) (then* (lemma name)
                                   (do-rewrite -1)
                                   (quant-interval subst))
```

```
        "(dclemma <name>): Adds DC lemma named <name> to the premise"
        "Introducing lemma ~a")

(defstep equiv (st1 st2)
  (let ((st (format nil "not((~a) = (~a))" st1 st2)))
    (then* (case st)
           (apply (APPLY-EXTENSIONALITY) (IFF) (GROUND))
           ))
  "equiv"
  "equiv")

(defstep apply-dc-rule (name insts)
  (then* (lemma name)
         (expand "rule" -1)
         (expand "seq" -1)
         (expand "+" -1)
         (expand "*" -1)
         (quant? * insts)
         (do-rewrite -1)
         (split)
         (quant-interval)
         (assert)
         (skosimp*)
         (hide-index))
  "(apply-dc-rule <name> <terms>): Applies the DC rule <name> with the instantiations <terms>"
  "Applying the DC rule ~a")

(defstep dcinst? (&optional (fnums *) subst (where *) copy?  if-match)
  (let (;(sformnum (find-?quant fnum subst *ps*))
         (sforms (select-seq (s-forms (current-goal *ps*))
                             fnums))
         (search (find-quant-terms* sforms subst where
                                     if-match *ps*))
         (sformnum (when search (car search)))
         (newterms (when search (cdr search))))
    (if newterms
        (let ((x (cons 'then*
                       (if copy?
                           (cons `(dcinst ,sformnum
                                   ,(car newterms) ,copy?)
                                 (loop for qterms in (cdr newterms)
                                       collect `(dcinst
                                                 ,(if (< sformnum 0)
                                                      (1- sformnum)
                                                      (1+ sformnum))
                                                 ,qterms)))
                           (loop for qterms in newterms
                                 collect `(dcinst ,sformnum ,(car qterms)))))))
         x)
        (skip)))
  "(dcinst? &optional <fnums[*]> subst <where[*]>  copy? if-match):
Behaves as inst? except that DC formulas are evaluated as top-level formulas
and rewriting is taking place."
  "~%DC-instantiating quantified variables")


(defstep dcinst (fnums &rest terms)
  (let ((sformnum (find-quant fnums terms *ps*)))
    (if sformnum
        (let ((pt (mapcar #'(lambda(x)(pc-parse x 'expr)) terms))
              (fmla (formula (car (select-seq (s-forms *goal*) (list sformnum)))))
              (ql (make-dc-quant-alist (bindings fmla) pt *current-context*))
              (exp (expression fmla))
              (il (mapcar `(lambda(x)
```

```
                            (if (not (is-dur-dependent? (cdr x)))
                                (cdr x)
                                (when (all-occurrences-at-top-level (car x) ,exp)
                                  (cdr x))))
                          ql)))
          (if (some #'not il)
              (skip-msg "Instantiating with duration dependent term inside a chopped formula is not allowed")
              (then (quant fnums il) (do-rewrite))))
              (skip-msg "Formula not instantiable")))
  "(dcinst <fnum> &rest <terms>): Behaves as inst, except that rewriting is
taking place."
  "~%Instantiating the top quantifier in ~a with the terms: ~% ~~a, ~")

(defstep false-chop (&optional (sformnum *))
  (let ((nums (find-dc-false-chop-num sformnum '-)))
    (if (car nums)
        (let ((chop (find-dc-fmla  nums))
              (a (args2 chop))
              (s (delete-dc-seq nums))
              (g (left-dc-conj  s))
              (d (right-dc-disj s))
              (insts (list "A" a "Delta" d "Gamma" g)))
          (apply-dc-rule$ "false_chop" insts))
        (skip-msg "Could not find matching DC formula")))
  "(false-chop &optional <fnums[*]>): Propagates false to the left of a chop
 through on lhs of sequent."
  "~%Propagating left false through chop")

(defstep chop-false (&optional (sformnum *))
  (let ((nums (find-dc-chop-false-num sformnum '-)))
    (if (car nums)
        (let ((chop (find-dc-fmla  nums))
              (a (args1 chop))
              (s (delete-dc-seq nums))
              (g (left-dc-conj  s))
              (d (right-dc-disj s))
              (insts (list "A" a "Delta" d "Gamma" g)))
          (apply-dc-rule$ "chop_false" insts))
        (skip-msg "Could not find matching DC formula")))
  "(chop-false &optional <fnums[*]>): Propagates false to the right of a chop
 through on lhs of sequent."
  "~%Propagating right false through chop")


(defstep box-right (&optional (sformnum *))
  (let ((nums (find-dc-box-num sformnum '+)))
    (if (car nums)
        (let ((a (args1 (find-dc-fmla nums)))
              (s (select-seq (s-forms *goal*) '*))
              (gs (gather-seq s (find-all-dc-box-nums '* '-) nil))
              (ds (gather-seq s (find-all-dc-diamond-nums '* '+) nil))
              (g (make-dc-conjunction
                  (dc-args (dc-operators (dc-args (formulas gs))))))
              (d (make-dc-disjunction
                  (dc-args (dc-operators (formulas ds)))))
              (insts (list "A" a "Gamma" g "Delta" d)))
          (apply-dc-rule$ "box_right" insts))
        (skip-msg "Could not find matching DC formula")))
  "(box-right &optional <fnums[*]>): Removes box from first box'ed formula in
    <fnums> in consequent of the sequent."
  "Removing consequent box")

(defstep box-left (&optional (sformnum *))
  (let  ((nums (find-dc-box-num sformnum '-)))
```

```
    (if (car nums)
        (let ((a (args1 (find-dc-fmla  nums)))
              (s (delete-dc-seq nums))
              (g (left-dc-conj  s))
              (d (right-dc-disj s))
              (insts (list "A" a "Gamma" g "Delta" d)))
          (apply-dc-rule$ "box_left" insts))
        (skip-msg "Could not find matching DC formula")))
  "(box-left &optional <fnums[*]>): Removes box from first box'ed formula in
    <fnums> in premiss of the sequent."
  "Removing antecedent box")

(defstep diamond-left (&optional (sformnum *))
  (let ((nums (find-dc-diamond-num sformnum '-)))
    (if (car nums)
        (let ((a (args1 (find-dc-fmla nums)))
              (s (select-seq (s-forms *goal*) '*))
              (gs (gather-seq s (find-all-dc-box-nums '* '-) nil))
              (ds (gather-seq s (find-all-dc-diamond-nums '* '+) nil))
              (g (make-dc-conjunction
                  (dc-args (dc-operators (dc-args (formulas gs))))))
              (d (make-dc-disjunction
                  (dc-args (dc-operators (formulas ds)))))
              (insts (list "A" a "Gamma" g "Delta" d)))
          (apply-dc-rule$ "diamond_left" insts))
        (skip-msg "Could not find matching DC formula")))
  "(diamond-left &optional <fnums[*]>): Removes box from first diamond'ed formula in
    <fnums> in premiss of the sequent."
  "Removing antecedent diamond")

(defstep diamond-right (&optional (sformnum *))
  (let ((nums (find-dc-diamond-num sformnum '+)))
    (if (car nums)
        (let ((a (args1 (find-dc-fmla nums)))
              (s (delete-dc-seq nums))
              (g (left-dc-conj  s))
              (d (right-dc-disj s))
              (insts (list "A" a "Gamma" g "Delta" d)))
          (apply-dc-rule$ "diamond_right" insts))
        (skip-msg "Could not find matching DC formula")))
  "(diamond-right &optional <fnums[*]>): Removes box from first diamond'ed formula in
    <fnums> in consequent of the sequent."
  "Removing consequent diamond")

(defstep mono (&optional (sformnuml *)(sformnumr *))
  (let ((numsl (find-dc-chop-num sformnuml '-)))
    (if (car numsl)
        (let ((chopl (find-dc-fmla  numsl))
              (a (args1 chopl))
              (b (args2 chopl))
              (numsr (find-dc-chop-num sformnumr '+))
              (chopr (find-dc-fmla numsr))
              (c (args1 chopr))
              (d (args2 chopr))
              (s (delete-dc-seq numsl))
              (g (left-dc-conj  s))
              (del (right-dc-disj s))
              (insts (list "A" a "B" b "C" c "D" d "Delta" del "Gamma" g)))
          (apply-dc-rule$ "mono" insts))
        (skip-msg "Could not find matching DC formula")))
  "(mono &optional <fnumsl[*]> <fnumsr[r]>): Applies monotonicity to chopped
   formula <fnumsl> of the premiss and <fnumsr> of the consequent."
  "Applying monotonicity")
```

```
(defstep mono! (&optional (sformnuml *)(sformnumr *))
  (let ((numsl (find-dc-chop-num sformnuml '-)))
    (if (car numsl)
        (let ((chopl (find-dc-fmla  numsl))
              (a (args1 chopl))
              (b (args2 chopl))
              (numsr (find-dc-chop-num sformnumr '+))
              (chopr (find-dc-fmla numsr))
              (c (args1 chopr))
              (d (args2 chopr))
              (insts (list "A" a "B" b "C" c "D" d)))
          (apply-dc-rule$ "mono_bang" insts))
        (skip-msg "Could not find matching DC formula")))
  "(mono! &optional <fnumsl[*]> <fnumsr[r]>): Applies monotonicity to chopped
   formula <fnumsl> of the premiss and <fnumsr> of the consequent."
  "Applying monotonicity")

(defstep mono-chop (&optional (st "")(sformnuml *))
  (let ((numsl (find-dc-chop-num sformnuml '-)))
    (if (car numsl)
        (let ((chopl (find-dc-fmla  numsl))
              (a (args1 chopl))
              (b (args2 chopl))
              (c (parse-iform st))
              (s (delete-dc-seq numsl))
              (g (left-dc-conj  s))
              (d (right-dc-disj s))
              (insts (list "A" a "B" b "C" c "Delta" d "Gamma" g)))
          (apply-dc-rule$ "mono_chop" insts))
        (skip-msg "Could not find matching DC formula")))
  "(mono-chop &optional <dcformula[\"\"]> <fnums[*]>):
    Applies monotonicity on the first chopped formula in <fnums> by introducing
    <dcformula> on the left hand side of the chop."
  "Applying left monotonicity of chop by introducing ~a")

(defstep mono-chop! (&optional (st "")(sformnuml *))
  (let ((numsl (find-dc-chop-num sformnuml '-)))
    (if (car numsl)
        (let ((chopl (find-dc-fmla  numsl))
              (a (args1 chopl))
              (b (args2 chopl))
              (c (parse-iform st))
              (s (delete-dc-seq numsl))
              (g (left-dc-conj  s))
              (d (right-dc-disj s))
              (insts (list "A" a "B" b "C" c "Delta" d "Gamma" g)))
          (apply-dc-rule$ "mono_chop_bang" insts))
        (skip-msg "Could not find matching DC formula")))
  "(mono-chop! &optional <st[\"\"]> <sformnuml[*]>): Tries to prove a trivial
monotonicity proof obligation."
  "Applying right monotonicity of chop and removing box")

(defstep chop-mono (&optional (st "")(sformnuml *))
  (let ((numsl (find-dc-chop-num sformnuml '-)))
    (if (car numsl)
        (let ((chopl (find-dc-fmla  numsl))
              (a (args1 chopl))
              (b (args2 chopl))
              (c (parse-iform st))
              (s (delete-dc-seq numsl))
              (g (left-dc-conj  s))
              (d (right-dc-disj s))
              (insts (list "A" a "B" b "C" c "Delta" d "Gamma" g)))
          (apply-dc-rule$ "chop_mono" insts))
```

```
           (skip-msg "Could not find matching DC formula")))
  "(chop-mono &optional <dcformula[\"\"]> <fnums[*]>):
    Applies monotonicity on the first chopped formula in <fnums> by introducing
   <dcformula> on the right hand side of the chop."
  "Applying right monotonicity of chop by introducing ~a")

(defstep chop-mono! (&optional (st "")(sformnuml *))
  (let ((numsl (find-dc-chop-num sformnuml '-)))
    (if (car numsl)
        (let ((chopl (find-dc-fmla  numsl))
              (a (args1 chopl))
              (b (args2 chopl))
              (c (parse-iform st))
              (s (delete-dc-seq numsl))
              (g (left-dc-conj  s))
              (d (right-dc-disj s))
              (insts (list "A" a "B" b "C" c "Delta" d "Gamma" g)))
          (apply-dc-rule$ "chop_mono_bang" insts))
           (skip-msg "Could not find matching DC formula")))
  "(chop-mono! &optional <st[\"\"]> <sformnuml[*]>): Tries to prove a trivial
monotonicity proof obligation."
  "Applying right monotonicity of chop and removing box")


(defstep mono-chop-mono (&optional (st "") (sformnuml *))
  (let ((numsl (find-dc-chop-chop-left-num sformnuml '-)))
    (if (car numsl)
        (let ((chopl (find-dc-fmla  numsl))
              (a (args1 (args1 chopl)))
              (b (args2 (args1 chopl)))
              (c (args2 chopl))
              (d (parse-iform st))
              (s (delete-dc-seq numsl))
              (g (left-dc-conj  s))
              (del (right-dc-disj s))
              (insts (list "A" a "B" b "C" c "D" d "Delta" del "Gamma" g)))
          (apply-dc-rule$ "mono_chop_mono" insts))
           (skip-msg "Could not find matching DC formula")))
  "(mono-chop-mono &optional <dcformula[\"\"]> <fnums[*]>):
    Applies monotonicity on the first chopped formula in <fnums> by introducing
   <dcformula> in center subinterval."
  "Applying center monotonicity of chop")


(defstep assoc (&optional (sformnuml *) (dir LR))
  (if (eq dir 'LR)
      (let ((numsl (find-dc-chop-chop-left-num sformnuml '*)))
        (if (car numsl)
            (let ((chopl (find-dc-fmla  numsl))
                  (a (args1 (args1 chopl)))
                  (b (args2 (args1 chopl)))
                  (c (args2 chopl))
                  (insts (list "A" a "B" b "C" c)))
              (rewrite "assoc_dc" numsl insts * dir))
            (skip-msg "Could not find matching DC formula")))
      (let ((numsl (find-dc-chop-chop-right-num sformnuml '*)))
        (if (car numsl)
            (let ((chopl (find-dc-fmla  numsl))
                  (b (args1 (args2 chopl)))
                  (c (args2 (args2 chopl)))
                  (a (args1 chopl))
                  (insts (list "A" a "B" b "C" c)))
              (rewrite "assoc_dc" numsl insts * dir))
            (skip-msg "Could not find matching DC formula"))))
```

```
  "(assoc &optional <fnums[*]>): Rewrites using associativity of chop
    the first double-chopped formula in <fnums> in the given direction."
  "~%Applying associativity of chop")


(defstep exists-intro (st &optional (sformnum *) (dir LR))
  (if (eq dir 'LR)
      (let ((nums (find-dc-chop-num sformnum '*)))
        (if (car nums)
            (let ((chop (find-dc-fmla  nums))
                  (n (car nums))
                  (b (args2 chop))
                  (a (args1 chop))
                  (p (parse-state st))
                  (insts (list "B" b "A" a "p" p)))
              (then* (rewrite "exists_intro" :fnums n :subst insts) (do-rewrite)
                     (skolem! n)))
            (skip-msg "Could not find matching DC formula")))
              (then (rewrite "exists_intro" :target-fnums sformnum :dir dir)
                    (do-rewrite)))
  "(exists-intro <string> &optional <fnums[*]>): Introduces left binding of duration
    measure of chopped formula <fnums>."
  "Introducing left existential binding of duration measure of ~a")

(defstep intro-exists (st &optional (sformnum *) (dir LR))
  (if (eq dir 'LR)
      (let ((nums (find-dc-chop-num sformnum '*)))
        (if (car nums)
            (let ((chop (find-dc-fmla  nums))
                  (n (car nums))
                  (b (args2 chop))
                  (a (args1 chop))
                  (p (parse-state st))
                  (insts (list "B" b "A" a "p" p)))
              (then* (rewrite "intro_exists" :fnums n :subst insts) (do-rewrite) (skolem! n)))
            (skip-msg "Could not find matching DC formula")))
              (then (rewrite "intro_exists" :target-fnums sformnum :dir dir)
                    (do-rewrite)))
  "(intro-exists <string> &optional <fnums[*]>): Introduces left binding of duration
    measure of chopped formula <fnums>."
  "Introducing left existential binding of duration measure of ~a")

(defstep intro-exists-intro (st &optional (sformnum *) (dir LR))
  (if (eq dir 'LR)
      (let ((nums (find-dc-chop-chop-left-num sformnum '*)))
        (if (car nums)
            (let ((chop (find-dc-fmla  nums))
                  (a (args1 (args1 chop)))
                  (b (args2 (args1 chop)))
                  (c (args2 chop))
                  (n (car nums))
                  (p (parse-state st))
                  (insts (list "B" b "A" a "C" c "p" p)))
              (then* (rewrite "intro_exists_intro" :fnums n :subst insts) (do-rewrite) (skolem! n)))
            (skip-msg "Could not find matching DC formula")))
              (then (rewrite "intro_exists_intro" :target-fnums sformnum :dir dir)
                    (do-rewrite)))
  "(intro-exists-intro <string> &optional <fnums[*]>): Introduces center binding of duration
    measure of chopped formula <fnums>."
  "Introducing center existential binding of duration measure of ~a")

(defstep exists-chop (&optional (sformnum *) (dir LR))
  (if (eq dir 'LR)
      (let ((nums (find-dc-exists-chop-num sformnum '*)))
```

```
        (if (car nums)
            (let ((chop (find-dc-fmla  nums))
                  (b (args2 chop))
                  (a (args1 chop))
                  (f (args1 a))
                  (insts (list "B" b "F" f)))
              (then (rewrite "exists_chop_dc" :subst insts) (do-rewrite)))
            (skip-msg "Could not find matching DC formula")))
              (then (rewrite "exists_chop_dc" :target-fnums sformnum :dir dir)
                    (do-rewrite)))
  "(exists-chop &optional <fnums[*]>): Moves existential quantifier
    in and out of left hand side of chopped formula <fnums>."
  "Moving left existential quantifier")

(defstep chop-exists (&optional (sformnum *) (dir LR))
  (if (eq dir 'LR)
      (let ((nums (find-dc-chop-exists-num sformnum '*)))
        (if (car nums)
            (let ((chop (find-dc-fmla  nums))
                  (b (args2 chop))
                  (a (args1 chop))
                  (f (args1 b))
                  (insts (list "A" a "F" f)))
              (then (rewrite "chop_exists_dc" :subst insts) (do-rewrite)))
            (skip-msg "Could not find matching DC formula")))
              (then (rewrite "chop_exists_dc" :target-fnums sformnum :dir dir)
                    (do-rewrite)))
  "(chop-exists &optional <fnums[*]> <dir[LR]>): Moves existential quantifier
    in and out of right hand side of chopped formula <fnums>."
  "Moving right existential quantifier")

(defstep axiom4 (&optional (sformnum *) (dir LR))
  (if (eq dir 'LR)
      (let ((nums (find-dc-chop-dur-num sformnum '*))
            (chop (find-dc-fmla  nums))
            (b (args2 chop))
            (a (args1 chop))
            (p (find-dur-state a))
            (r (find-dur-value a))
            (s (find-dur-value b))
            (insts (list "p" p "r" r "s" s)))
        (then (rewrite "axiom4_dc" :subst insts :dir dir)
              (do-rewrite)))
      (then (rewrite "axiom4_dc" :target-fnums sformnum :dir dir)
            (do-rewrite)))
  "(axiom4 <fnums[*]> <dir[LR]>): Rewrites <fnums> using Axiom 4."
  "Rewriting using Axiom 4")

(defstep induct-chop (st p)
  (let ((s (s-forms *goal*))
        (g (left-dc-conj  s))
        (d (right-dc-disj s))
        (r (parse-iform-iform st))
        (insts (list "R" r "p" p "Delta" d "Gamma" g)))
    (apply-dc-rule$ "induct_chop" insts))
  "(induct-chop <string1> <string2>): Uses left DC induction where <string1>
   contains the induction formula of the type [IForm -> IForm], and <string2>
   provides the state to induct over."
  "~%Using left DC induction")

(defstep chop-induct (st p)
  (let ((s (s-forms *goal*))
        (g (left-dc-conj  s))
        (d (right-dc-disj s))
```

```
        (r (parse-iform-iform st))
        (insts (list "R" r "p" p "Delta" d "Gamma" g)))
    (apply-dc-rule$ "chop_induct" insts))
  "(chop-induct <string1> <string2>): Uses right DC induction where <string1>
   contains the induction formula of the type [IForm -> IForm], and <string2>
   provides the state to induct over."
  "~%Using right DC induction")

(defstep disj-chop (&optional (sformnum *) (dir 'LR))
  (if (eq dir 'LR)
      (let ((numsl (find-dc-chop-left-disj-num sformnum '*)))
        (if (car numsl)
            (let ((chopl (find-dc-fmla  numsl))
                  (a (args1 (args1 chopl)))
                  (b (args2 (args1 chopl)))
                  (c (args2 chopl))
                  (insts (list "A" a "B" b "C" c)))
              (rewrite "disj_chop_dc" :subst insts :dir dir))
            (skip-msg "Could not find matching DC formula")))
      (rewrite "disj_chop_dc" :target-fnums sformnum :dir dir))
  "(disj-chop <fnums[*]> <dir[LR]>): Distributes chop over left disjunction"
  "~%Distributing chop over left disjunction")

(defstep chop-disj (&optional (sformnum *) (dir 'LR))
  (if (eq dir 'LR)
      (let ((numsl (find-dc-chop-right-disj-num sformnum '*)))
        (if (car numsl)
            (let ((chopl (find-dc-fmla  numsl))
                  (b (args1 (args2 chopl)))
                  (c (args2 (args2 chopl)))
                  (a (args1 chopl))
                  (insts (list "A" a "B" b "C" c)))
              (rewrite "chop_disj_dc" :subst insts :dir dir))
            (skip-msg "Could not find matching DC formula")))
      (rewrite "disj_chop_dc" :target-fnums sformnum :dir dir))
  "(chop-disj <fnums[*]>): Distributes chop over right disjunction"
  "~%Distributing chop over right disjunction")

(defstep free-chop (&optional (sformnuml *) (dir LR))
  (if (eq dir 'LR)
      (let ((numsl (find-dc-chop-num sformnuml '*))
            (n (car numsl)))
        (if n
            (then* (auto-rewrite-theory "dcp_context_free")
                   (do-rewrite n)
                   (stop-rewrite-theory "dcp_context_free")
                   (let ((chopl (find-dc-fmla  numsl))
                         (f (find-context-free-conj (args1 chopl))))
                     (if f
                         (let ((f1 (args1 f))
                               (a (gather-not-context-free (args1 chopl) f))
                               (b (args2 chopl))
                               (instst (list "B" b "f" f1))
                               (insts (list "A" a "B" b "f" f1)))
                           (then* (if (is-dc-true? a)
                                      (rewrite "free_chop_true_dc" :subst instst :dir dir)
                                      (rewrite "free_chop_dc" :subst insts :dir dir))
                                  (auto-rewrite-theory "dcp_cf_reverse")
                                  (do-rewrite)
                                  (stop-rewrite-theory "dcp_cf_reverse")))
                         (skip-msg "No left context free expression found"))))
            (skip-msg "Could not find matching DC formula")))
      (then* (rewrite "free_chop_dc" :dir dir)
             (auto-rewrite-theory "dcp_cf_reverse")
```

```
             (do-rewrite)
             (stop-rewrite-theory "dcp_cf_reverse")))
      "(free-chop <fnums[*]> <dir[LR]>): Propagates left context free expression
through chop."
      "~%Propagating left context free expression through chop")

(defstep chop-free (&optional (sformnuml *) (dir LR))
  (if (eq dir 'LR)
      (let ((numsl (find-dc-chop-num sformnuml '-))
            (n (car numsl)))
        (if n
            (then* (auto-rewrite-theory "dcp_context_free")
                   (do-rewrite n)
                   (stop-rewrite-theory "dcp_context_free")
                   (let ((chopl (find-dc-fmla  numsl))
                         (f (find-context-free-conj (args2 chopl))))
                     (if f
                         (let ((f1 (args1 f))
                               (b (gather-not-context-free (args2 chopl) f))
                               (a (args1 chopl))
                               (instst (list "A" a "f" f1))
                               (insts (list "A" a "B" b "f" f1)))
                           (then* (if (is-dc-true? b)
                                      (rewrite "chop_free_true_dc" :subst instst :dir dir)
                                      (rewrite "chop_free_dc" :subst insts :dir dir))
                                  (auto-rewrite-theory "dcp_cf_reverse")
                                  (do-rewrite)
                                  (stop-rewrite-theory "dcp_cf_reverse")))
                         (skip-msg "No right context free expression found"))))
            (skip-msg "Could not find matching DC formula")))
      (then* (rewrite "chop_free_dc" :dir dir)
             (auto-rewrite-theory "dcp_cf_reverse")
             (do-rewrite)
             (stop-rewrite-theory "dcp_cf_reverse")))
  "(chop-free <fnums[*]> <dir[LR]>): Propagates right context free expression
through chop."
  "~%Propagating right context free expression through chop")

(defstep dc-case (str)
   (let ((obj (add-index (parse-iform str))))
     (then* (case obj) (do-rewrite)))
   "(dc-case string): Behaves as case, except evalues DC formulas as top-most
formulas."
   "~%DC case splitting on ~@~%   ~a, ~")

(defstep dur-preserve (&optional (sformnum *))
  (let ((nums (find-dc-chop-num sformnum '-)))
    (if (car nums)
        (let ((chop (find-dc-fmla nums))
              (states (get-states chop))
              (p (args1 (car states)))
              (q (args1 (cadr states)))
              (g (make-lambda-from-expr-iform (args1 chop)))
              (g1 (make-lambda-from-expr-bool (args1 chop)))
              (s (delete-dc-seq nums))
              (gam (left-dc-conj  s))
              (del (right-dc-disj s))
              (insts (list "G" g "G1" g1 "p" p "q" q
                           "Delta" del "Gamma" gam)))
          (apply-dc-rule$ "dur_preserve" insts))
        (skip-msg "Could not find matching DC formula")))
  "(dur-preserve <fnums[*]>): Makes duration preserving predicate of two
sub-intervals available on the whole interval."
  "~%Using duration preservation")
```

C Gas Burner

C.1 Specification

The full gasburner specification in PC/DC is shown below.

```
gasburner: DCTHEORY
BEGIN

  IMPORTING dcp_thms

    X: VAR IForm
    Leak: State
    p: VAR State


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

    Req_1: 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')

      %%% Sublemmas for lem1 %%%

        lem1_1: LEMMA (len<=n(1));('not Leak'/\len>=n(30))
                                  => (dur(Leak)<=n(1) /\ len>=n(30))

        %%% Sublemmas for lem1_2 %%%

          lem1_2_1: LEMMA (([](Des_1 /\ Des_2 /\ X => Triple)) /\
                                  Des_1 /\ Des_2 /\ (X;'Leak') =>
                          (T;'Leak'\/ (T;'Leak';'Leak') \/
                           (T;'Leak';'not Leak';'Leak')))

          lem1_2_2: LEMMA T;'Leak' => Triple


          lem1_2_3: LEMMA T;'Leak';'Leak' => Triple


          lem1_2_4: LEMMA T;T => T


          lem1_2_5: LEMMA Des_1 /\ Des_2 /\
                                  (T;'Leak';'not Leak';'Leak') => Triple

        lem1_2: LEMMA ([](Des_1 /\ Des_2 /\ X => Triple)) /\
```

```
                        Des_1 /\ Des_2 /\ (X;'Leak') => Triple

    %%% Sublemmas for lem1_3 %%%

      lem1_3_1: LEMMA (([](Des_1 /\ Des_2 /\ X => Triple)) /\
                             Des_1 /\ Des_2 /\ (X;'not Leak') =>
                     (T;' not Leak'\/ (T;'Leak';'not Leak') \/
                      (T;'Leak';'not Leak';'not Leak')))

      lem1_3_2: LEMMA T;'not Leak' => Triple

      lem1_3_3: LEMMA T;'Leak';'not Leak' => Triple

      lem1_3_4: LEMMA (T;'Leak';'not Leak';'not Leak') => Triple

    lem1_3: LEMMA (([](Des_1 /\ Des_2 /\ X => Triple)) /\
                     Des_1 /\ Des_2 /\ (X;'not Leak') => Triple)

  lem1: LEMMA Des_1 /\ Des_2 => Triple

  %%% Sublemmas for Lemma 2 %%%

    lem2_1: LEMMA (dur(Leak)<=len/n(30));(dur(Leak)<=n(1))/\ len>=n(60) => Safe

    lem2_2: LEMMA Des_1 /\ T;'Leak' /\ (len >= n(60)) => Safe

    lem2_3 : LEMMA Des_1 /\ T;'Leak';'not Leak' /\ (len >= n(60)) => Safe

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

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

End gasburner
```

PVS provides the ability to produce LaTeX output of theories. The LaTeX version of the gasburner specification is shown below.

gasburner : **DCTheory**

BEGIN

IMPORTING dcp_thms

X : VAR IForm

Leak : State

p : VAR State

Safe : IForm $= 20 \times \int(\mathrm{Leak}) \leq \ell$

Req_1 : IForm $= \ell \geq 60 \Rightarrow \mathrm{Safe}$

Des_1 : IForm $= \Box\,(\lceil \mathrm{Leak} \rceil \Rightarrow \ell \leq 1)$

Des_2 : IForm $= \Box\,((\Diamond\lceil \mathrm{Leak} \rceil);(\Diamond\lceil \neg\, \mathrm{Leak} \rceil);(\Diamond\lceil \mathrm{Leak} \rceil) \Rightarrow \ell \geq 30)$

T : IForm $= 30 \times \int(\mathrm{Leak}) \leq \ell$

Triple : IForm $= T \vee (T; \lceil \text{Leak} \rceil) \vee (T; \lceil \text{Leak} \rceil; \lceil \neg \text{ Leak} \rceil)$

lem1_1 : LEMMA
$(\ell \leq 1); (\lceil \neg \text{ Leak} \rceil \wedge \ell \geq 30) \Rightarrow (\int(\text{Leak}) \leq 1 \wedge \ell \geq 30)$

lem1_2_1 : LEMMA
$((\Box (\text{Des\_1} \wedge \text{Des\_2} \wedge X \Rightarrow \text{Triple})) \wedge \text{Des\_1} \wedge \text{Des\_2} \wedge (X; \lceil \text{Leak} \rceil)$
$\Rightarrow$
$(T; \lceil \text{Leak} \rceil \vee (T; \lceil \text{Leak} \rceil; \lceil \text{Leak} \rceil)$
$\vee (T; \lceil \text{Leak} \rceil; \lceil \neg \text{ Leak} \rceil; \lceil \text{Leak} \rceil)))$

lem1_2_2 : LEMMA $T; \lceil \text{Leak} \rceil \Rightarrow \text{Triple}$

lem1_2_3 : LEMMA $T; \lceil \text{Leak} \rceil; \lceil \text{Leak} \rceil \Rightarrow \text{Triple}$

lem1_2_4 : LEMMA $T; T \Rightarrow T$

lem1_2_5 : LEMMA $\text{Des\_1} \wedge \text{Des\_2} \wedge (T; \lceil \text{Leak} \rceil; \lceil \neg \text{ Leak} \rceil; \lceil \text{Leak} \rceil) \Rightarrow \text{Triple}$

lem1_2 : LEMMA
$(\Box (\text{Des\_1} \wedge \text{Des\_2} \wedge X \Rightarrow \text{Triple})) \wedge \text{Des\_1} \wedge \text{Des\_2} \wedge (X; \lceil \text{Leak} \rceil)$
$\Rightarrow \text{Triple}$

lem1_3_1 : LEMMA
$((\Box (\text{Des\_1} \wedge \text{Des\_2} \wedge X \Rightarrow \text{Triple})) \wedge \text{Des\_1} \wedge \text{Des\_2} \wedge (X; \lceil \neg \text{ Leak} \rceil)$
$\Rightarrow$
$(T; \lceil \neg \text{ Leak} \rceil \vee (T; \lceil \text{Leak} \rceil; \lceil \neg \text{ Leak} \rceil)$
$\vee (T; \lceil \text{Leak} \rceil; \lceil \neg \text{ Leak} \rceil; \lceil \neg \text{ Leak} \rceil)))$

lem1_3_2 : LEMMA $T; \lceil \neg \text{ Leak} \rceil \Rightarrow \text{Triple}$

lem1_3_3 : LEMMA $T; \lceil \text{Leak} \rceil; \lceil \neg \text{ Leak} \rceil \Rightarrow \text{Triple}$

lem1_3_4 : LEMMA $(T; \lceil \text{Leak} \rceil; \lceil \neg \text{ Leak} \rceil; \lceil \neg \text{ Leak} \rceil) \Rightarrow \text{Triple}$

lem1_3 : LEMMA
$((\Box (\text{Des\_1} \wedge \text{Des\_2} \wedge X \Rightarrow \text{Triple})) \wedge \text{Des\_1} \wedge \text{Des\_2} \wedge (X; \lceil \neg \text{ Leak} \rceil)$
$\Rightarrow \text{Triple})$

lem1 : LEMMA $\text{Des\_1} \wedge \text{Des\_2} \Rightarrow \text{Triple}$

lem2_1 : LEMMA $(\int(\text{Leak}) \leq \ell / 30); (\int(\text{Leak}) \leq 1) \wedge \ell \geq 60 \Rightarrow \text{Safe}$

lem2_2 : LEMMA $\text{Des\_1} \wedge T; \lceil \text{Leak} \rceil \wedge (\ell \geq 60) \Rightarrow \text{Safe}$

lem2_3 : LEMMA Des_1 $\wedge\ T; \lceil \text{Leak} \rceil; \lceil \neg\ \text{Leak} \rceil \wedge (\ell\ \geq\ 60)\ \Rightarrow$ Safe

lem2 : LEMMA Des_1 $\wedge$ Triple $\wedge\ (\ell\ \geq\ 60)\ \Rightarrow$ Safe

Gasburner : LEMMA (Des_1 $\wedge$ Des_2 $\wedge\ (\ell\ \geq\ 60))\ \Rightarrow$ Safe

END gasburner

The following lemmas are widely applicable. Some of the lemmas have been used in the correctness proofs of the gasburner specification. They have all been proved correct in PC/DC.

dcp_thms : **DCTheory**
BEGIN

p, q : VAR State
A, B : VAR IForm
r, s : VAR PosTime
dur_add_gt_gt : LEMMA $((\int(p)\ >\ r); (\int(p)\ >\ s))\ \Rightarrow\ (\int(p)\ >\ r\ +\ s)$

dur_add_ge_gt : LEMMA $((\int(p)\ \geq\ r); (\int(p)\ >\ s))\ \Rightarrow\ (\int(p)\ >\ r\ +\ s)$

dur_add_eq_gt : LEMMA $((\int(p)\ =\ r); (\int(p)\ >\ s))\ \Rightarrow\ (\int(p)\ \geq\ r\ +\ s)$

dur_add_ge_ge : LEMMA $((\int(p)\ \geq\ r); (\int(p)\ \geq\ s))\ \Rightarrow\ (\int(p)\ \geq\ r\ +\ s)$

dur_add_gt_ge : LEMMA $((\int(p)\ >\ r); (\int(p)\ \geq\ s))\ \Rightarrow\ (\int(p)\ >\ r\ +\ s)$

dur_add_eq_ge : LEMMA $((\int(p)\ =\ r); (\int(p)\ \geq\ s))\ \Rightarrow\ (\int(p)\ \geq\ r\ +\ s)$

dur_add_le_le : LEMMA $((\int(p)\ \leq\ r); (\int(p)\ \leq\ s))\ \Rightarrow\ (\int(p)\ \leq\ r\ +\ s)$

Thm1a : LEMMA $\int(p)\ +\ \int(\neg\ p)\ =\ \ell$

Thm1b : LEMMA $\int(p)\ \leq\ \ell$

Thm2 : LEMMA $\lceil \text{tt} \rceil \vee \lceil\rceil$

Thm3a : LEMMA $\lceil p \rceil\ \Rightarrow\ (\int(\neg\ p)\ =\ 0)$

Thm3b : LEMMA $\lceil\rceil\ \Rightarrow\ (\int(p)\ =\ 0)$

Thm4 : LEMMA $\lceil p\ \Rightarrow\ q \rceil\ \Rightarrow\ (\int(p)\ \leq\ \int(q))$

Thm6a : LEMMA $\lceil p \rceil \;\Rightarrow \lceil p \rceil ; \lceil p \rceil$

Thm6 : LEMMA $\lceil p \rceil ; \lceil p \rceil \;\Leftrightarrow \lceil p \rceil$

pred_box : LEMMA $\lceil p \rceil \;\Rightarrow\; \Box\, (\lceil p \rceil \vee \ell \;=\; 0)$

Thm6_len : LEMMA
$(r \;>\; 0 \wedge s \;>\; 0)$
$\Rightarrow\; ((\lceil p \rceil \wedge \ell \;=\; r);(\lceil p \rceil \wedge \ell \;=\; s) \;\Leftrightarrow\; (\lceil p \rceil \wedge (\ell \;=\; r \;+\; s)))$

Thm7a : LEMMA (TRUE; $\lceil p \rceil$) $\vee$ (TRUE; $\lceil \neg\, p \rceil$) $\vee\; \lceil\rceil$

Thm7b : LEMMA ($\lceil p \rceil$; TRUE) $\vee$ ($\lceil \neg\, p \rceil$; TRUE) $\vee\; \lceil\rceil$

$m(r, s)$: Time $=$ (IF $r \;<\; s$ THEN $r \;/\; 2$ ELSE $s \;/\; 2$ ENDIF)

lowbnd : LEMMA
$(\Box\, ((\lceil \neg\, p \rceil ; \lceil p \rceil ; \lceil \neg\, p \rceil) \;\Rightarrow\; \ell \;\geq\; r) \wedge (\lceil \neg\, p \rceil ; \lceil p \rceil ; \lceil \neg\, p \rceil))$
$\Rightarrow\; (\lceil \neg\, p \rceil ; (\lceil p \rceil \wedge \ell \;\geq\; r); \lceil \neg\, p \rceil)$

pr_impl : LEMMA $(\lceil p \rceil \wedge \lceil p \;\Rightarrow\; q \rceil) \;\Rightarrow \lceil q \rceil$

true_chop_dur_gt : LEMMA TRUE; $(\int(p) \;>\; r) \;\Leftrightarrow\; \int(p) \;>\; r$

chop_true_dur_gt : LEMMA $(\int(p) \;>\; r)$; TRUE $\;\Leftrightarrow\; \int(p) \;>\; r$

not_pr : LEMMA $(\int(p) \;=\; \ell) \;\Leftrightarrow\; \Box\, (\neg\, \lceil \neg\, p \rceil)$

not_pr_diamond : LEMMA $(\neg\, \lceil p \rceil) \;\Leftrightarrow\; (\Diamond \lceil \neg\, p \rceil \vee \lceil\rceil)$

pr_or_dur : LEMMA $\lceil p \vee q \rceil \;\Rightarrow\; \ell \;\leq\; (\int(p) \;+\; \int(q))$

pr_true_len : LEMMA ($\lceil p \rceil$; TRUE) $\Rightarrow\; (\ell \;>\; 0)$

pr_diamond_pr : LEMMA $\lceil p \rceil \;\Rightarrow\; \Diamond \lceil p \rceil$

pr_and : LEMMA $\lceil p \wedge q \rceil \;\Leftrightarrow \lceil p \rceil \wedge \lceil q \rceil$

len_ge_chop : LEMMA $\ell \;\geq\; r \;\Leftrightarrow\; (\ell \;=\; r);(\ell \;\geq\; 0)$

len_gt_chop : LEMMA $\ell \;>\; r \;\Leftrightarrow\; (\ell \;=\; r);(\ell \;>\; 0)$

END dcp_thms

C.2 An Example Proof

We refrain from showing the lengthy output of all gasburner proofs. Instead we show an example of a proof and all the proof scripts. Below follows first a PVS-generated LaTeX version of the proof of lemma `lem2_1`.

Terse proof for `lem2_1`.

`lem2_1:`

$$\begin{array}{|ll}
 & \\ \hline
\{1\} & (\int(\text{Leak}) \leq \ell / 30);(\int(\text{Leak}) \leq 1) \wedge \ell \geq 60 \Rightarrow \text{Safe}
\end{array}$$

Initializing the DC formula,

Applying disjunctive simplification to flatten sequent,

Introducing left existential binding of duration measure of Leak,

Introducing left existential binding of duration measure of Leak,

Expanding the definition of ℓ,

Introducing left existential binding of duration measure of tt,

Expanding the definition of Safe,

Expanding the definition of ℓ,

Introducing left existential binding of duration measure of tt,

DC case splitting on $(\int(\text{tt}) = r''');(\int(\text{tt}) = r'''')$, ,

we get 2 subgoals:

`lem2_1.1:`

$$\begin{array}{|ll}
\{-1\} & (\int(\text{tt}) = r''');(\int(\text{tt}) = r'''') \\
\{-2\} & (\int(\text{tt}) = r''' \\
 & \quad \wedge (\int(\text{Leak}) = r' \wedge (\int(\text{Leak}) \leq \int(\text{tt}) / 30)));(\int(\text{tt}) = r'''' \\
 & \quad\quad \wedge \\
 & \quad (\int(\text{Leak}) = r'' \\
 & \quad\quad\quad \wedge \\
 & \quad\quad (\int(\text{Leak}) \leq 1))) \\
\{-3\} & \int(\text{tt}) \geq 60 \\ \hline
\{1\} & 20 \times \int(\text{Leak}) \leq \int(\text{tt})
\end{array}$$

DC case splitting on $(\int(\text{Leak}) = r');(\int(\text{Leak}) = r'')$, ,

we get 2 subgoals:

`lem2_1.1.1:`

$$\begin{array}{|ll}
\{-1\} & (\int(\text{Leak}) = r');(\int(\text{Leak}) = r'') \\
 & \vdots \\ \hline
 & \vdots
\end{array}$$

Rewriting using Axiom 4,

Rewriting using Axiom 4,

DC case splitting on $(30 \times r' \leq r'''); (r'''' \geq 0 \wedge r'' \leq 1)$, ,

we get 2 subgoals:

`lem2_1.1.1.1:`

$$\begin{array}{|ll}
\{\text{-}1\} & (30 \times r' \leq r'''); (r'''' \geq 0 \wedge r'' \leq 1) \\
\{\text{-}2\} & (\int(\mathrm{Leak}) = r' + r'') \\
\{\text{-}3\} & (\int(\mathrm{tt}) = r''' + r'''') \\
 & \vdots \\
\hline
 & \vdots
\end{array}$$

Propagating left context free expression through chop,

Applying disjunctive simplification to flatten sequent,

Propagating right context free expression through chop,

This completes the proof of `lem2_1.1.1.1`.

`lem2_1.1.1.2:`

$$\begin{array}{|ll}
\{\text{-}1\} & (\int(\mathrm{Leak}) = r' + r'') \\
\{\text{-}2\} & (\int(\mathrm{tt}) = r''' + r'''') \\
 & \vdots \\
\hline
\{1\} & (30 \times r' \leq r'''); (r'''' \geq 0 \wedge r'' \leq 1) \\
 & \vdots
\end{array}$$

Hiding formulas: -1, -2,

Applying monotonicity,

we get 2 subgoals:

`lem2_1.1.1.2.1:`

$$\begin{array}{|ll}
\{\text{-}1\} & (\int(\mathrm{tt}) = r''') \\
\{\text{-}2\} & (\int(\mathrm{Leak}) = r') \\
\{\text{-}3\} & \int(\mathrm{Leak}) \leq \int(\mathrm{tt}) \;/\; 30 \\
\hline
\{1\} & 30 \times r' \leq r'''
\end{array}$$

Invoking decision procedures,

This completes the proof of `lem2_1.1.1.2.1`.

`lem2_1.1.1.2.2:`

$$\begin{array}{|ll}
\{\text{-}1\} & (\int(\mathrm{tt}) = r'''') \\
\{\text{-}2\} & (\int(\mathrm{Leak}) = r'') \\
\{\text{-}3\} & \int(\mathrm{Leak}) \leq 1 \\
\hline
\{1\} & r'' \leq 1
\end{array}$$

Invoking decision procedures,

This completes the proof of `lem2_1.1.1.2.2`.

`lem2_1.1.2`:

$$
\begin{array}{|ll}
 & \vdots \\
\hline
\{1\} & (\int(\mathrm{Leak}) \; = \; r'); (\int(\mathrm{Leak}) \; = \; r'') \\
 & \vdots
\end{array}
$$

Hiding formulas: -1,

Applying monotonicity,

This completes the proof of `lem2_1.1.2`.

`lem2_1.2`:

$$
\begin{array}{|ll}
\{\text{-}1\} & (\int(\mathrm{tt}) \; = \; r''' \\
 & \quad \wedge (\int(\mathrm{Leak}) \; = \; r' \wedge (\int(\mathrm{Leak}) \; \leq \; \int(\mathrm{tt}) \; / \; 30))); (\int(\mathrm{tt}) \; = \; r'''' \\
 & \qquad \wedge \\
 & \quad\; (\int(\mathrm{Leak}) \; = \; r'' \\
 & \qquad\quad \wedge \\
 & \qquad\; (\int(\mathrm{Leak}) \; \leq \; 1))) \\
\{\text{-}2\} & \int(\mathrm{tt}) \; \geq \; 60 \\
\hline
\{1\} & (\int(\mathrm{tt}) \; = \; r'''); (\int(\mathrm{tt}) \; = \; r'''') \\
\{2\} & 20 \; \times \; \int(\mathrm{Leak}) \; \leq \; \int(\mathrm{tt})
\end{array}
$$

Applying monotonicity,

This completes the proof of `lem2_1.2`.

Q.E.D.

C.3 Proof Scripts

A proof script is a listing of all the commands given to the prover to complete a proof. We include below a listing of all the proof scripts used to prove the lemmas of the gasburner specification. Note, that the numbering in the proof scripts indicate the subgoal number. This number is not typed in by the user, but is used by PVS to keep track of the proof.

The following listing is obtained by the Emacs command `M-x show-proofs-pvs-file`.

```
Proof scripts for file gasburner.dc:


gasburner.lem1_1: PROVED
```

```
("" (DCINIT)
    (FLATTEN)
    (EXISTS-INTRO "tt")
    (EXISTS-INTRO "Leak")
    (INTRO-EXISTS "tt")
    (DC-CASE "dur(Leak) = n(r!2);dur(Leak) = n(0)")
    (("1" (DC-CASE "dur(tt) = n(r!1);dur(tt) = n(r!3)")
          (("1" (DC-CASE "(n(r!2)<=n(r!1) /\ n(r!2)<=n(1));(n(r!3)>=n(30))")
                (("1" (AXIOM4 -2)
                      (AXIOM4 -3)
                      (FREE-CHOP)
                      (FLATTEN)
                      (CHOP-FREE)
                      (FLATTEN)
                      (EXPAND "len")
                      (ASSERT)
                      (PROPAX))
                  ("2" (HIDE -1 -2)
                       (MONO!)
                       (("1" (DCLEMMA "Thm1b")
                             (EXPAND "len")
                             (ASSERT)
                             (PROPAX))
                         ("2" (EXPAND "len")
                              (ASSERT))))))
           ("2" (HIDE -1) (MONO!))))
      ("2" (MONO!)
           (DCLEMMA "Thm3a" ("p" "not Leak"))
           (ASSERT)
           (PROPAX))))
```

gasburner.lem1_2_1: PROVED

```
("" (APPLY (THEN* (SKOLEM 1 ("X!1"))
                  (DCINIT)
                  (FLATTEN)))
    (MONO-CHOP "Triple")
    (("1" (APPLY (THEN* (EXPAND "Des_1")
                        (EXPAND "Des_2")))
          (BOX-RIGHT)
          (BOX-LEFT)
          (GROUND))
      ("2" (APPLY (THEN* (HIDE -4)
                         (EXPAND "Triple")))
           (DISJ-CHOP)
           (ASSERT)
           (DISJ-CHOP))))
```

gasburner.lem1_2_2: PROVED

```
("" (DCINIT)
    (FLATTEN)
    (EXPAND "Triple")
    (ASSERT)
    (PROPAX))
```

gasburner.lem1_2_3: PROVED

```
("" (DCINIT)
    (FLATTEN)
    (ASSOC)
```

```
    (CHOP-MONO! "'Leak'")
    (("1" (DCLEMMA "Thm6") (GROUND)) ("2" (EXPAND "Triple")
                                          (GROUND))))
```

gasburner.lem1_2_4: PROVED

```
("" (DCINIT)
    (FLATTEN)
    (APPLY (THEN* (EXPAND "T")
                  (EXPAND "len")))
    (DUR-PRESERVE)
    (ASSERT))
```

gasburner.lem1_2_5: PROVED

```
("" (DCINIT)
    (FLATTEN)
    (ASSOC)
    (ASSOC)
    (EXPAND "Des_1")
    (EXPAND "Des_2")
    (CHOP-MONO "T;'Leak'")
    (("1" (BOX-RIGHT)
          (ASSOC :DIR RL)
          (DCLEMMA "lowbnd"
                   ("p" "not Leak" "r" "30"))
          (ASSERT)
          (SPLIT)
          (("1" (ASSOC)
                (MONO-CHOP "len<=n(1)")
                (ASSOC -5 :DIR RL)
                (MONO-CHOP! "T" -5)
                (DCLEMMA "lem1_1")
                (EXPAND "T")
                (GROUND))
            ("2" (BOX-RIGHT)
                 (BOX-LEFT -2)
                 (GROUND)
                 (MONO!)
                 (("1" (MONO!)
                       (("1" (DCLEMMA "pr_diamond_pr")
                             (GROUND))
                         ("2" (DCLEMMA "pr_diamond_pr")
                              (GROUND))))
                   ("2" (DCLEMMA "pr_diamond_pr")
                        (GROUND))))))
      ("2" (ASSOC -4 :DIR RL)
           (MONO-CHOP! "T" -4)
           (("1" (DCLEMMA "lem1_2_4")
                 (GROUND))
             ("2" (EXPAND "Triple")
                  (ASSERT)
                  (PROPAX))))))
```

gasburner.lem1_2: PROVED

```
("" (SKOLEM 1 ("X!1"))
    (DCINIT)
    (FLATTEN)
    (DCLEMMA "lem1_2_1")
    (SPLIT)
```

```
    (("1" (DCLEMMA "lem1_2_2")
          (ASSERT)
          (PROPAX))
      ("2" (DCLEMMA "lem1_2_3")
           (ASSERT)
           (PROPAX))
      ("3" (DCLEMMA "lem1_2_5")
           (ASSERT)
           (PROPAX))
      ("4" (PROPAX))
      ("5" (PROPAX))
      ("6" (PROPAX))
      ("7" (PROPAX))))
```

```
gasburner.lem1_3_1: PROVED

("" (SKOLEM 1 ("X!1"))
    (DCINIT)
    (FLATTEN)
    (MONO-CHOP "Triple")
    (("1" (EXPAND "Des_1")
          (EXPAND "Des_2")
          (BOX-RIGHT)
          (BOX-LEFT)
          (SPLIT)
          (("1" (PROPAX)) ("2" (PROPAX))
                          ("3" (PROPAX))
                          ("4" (PROPAX))))
      ("2" (HIDE -4)
           (EXPAND "Triple")
           (DISJ-CHOP)
           (ASSERT)
           (DISJ-CHOP))))
```

```
gasburner.lem1_3_2: PROVED

("" (DCINIT)
    (EXPAND "Triple")
    (EXPAND "T")
    (FLATTEN)
    (HIDE 2 3)
    (EXPAND "len")
    (EXISTS-INTRO "tt")
    (EXISTS-INTRO "Leak")
    (INTRO-EXISTS "tt")
    (DC-CASE "dur(Leak) = n(r!2);dur(Leak) = n(0)")
    (("1" (DC-CASE "dur(tt) = n(r!1);dur(tt) = n(r!3)")
          (("1" (AXIOM4)
                (AXIOM4)
                (DC-CASE "(n(30)*n(r!2)<=n(r!1));(n(r!3)>n(0))")
                (("1" (FREE-CHOP)
                      (FLATTEN)
                      (CHOP-FREE)
                      (FLATTEN)
                      (ASSERT))
                  ("2" (HIDE -1 -2)
                       (MONO!)
                       (("1" (ASSERT))
                         ("2" (EXPAND "pr")
                              (EXPAND "len")
                              (GROUND))))))
            ("2" (HIDE -1) (MONO!))))
```

```
      ("2" (MONO!)
           (DCLEMMA "Thm3a" ("p" "not Leak"))
           (ASSERT)
           (PROPAX))))


gasburner.lem1_3_3: PROVED

("" (DCINIT)
    (FLATTEN)
    (EXPAND "Triple")
    (FLATTEN)
    (PROPAX))


gasburner.lem1_3_4: PROVED

("" (DCINIT)
    (FLATTEN)
    (ASSOC)
    (CHOP-MONO! "'not Leak'")
    (("1" (DCLEMMA "Thm6")
          (ASSERT)
          (PROPAX))
      ("2" (EXPAND "Triple")
           (ASSERT)
           (PROPAX))))


gasburner.lem1_3: PROVED

("" (SKOLEM 1 ("X!1"))
    (DCINIT)
    (FLATTEN)
    (DCLEMMA "lem1_3_1")
    (SPLIT)
    (("1" (DCLEMMA "lem1_3_2")
          (ASSERT)
          (PROPAX))
      ("2" (DCLEMMA "lem1_3_3")
           (ASSERT)
           (PROPAX))
      ("3" (DCLEMMA "lem1_3_4")
           (ASSERT)
           (PROPAX))
      ("4" (PROPAX))
      ("5" (PROPAX))
      ("6" (PROPAX))
      ("7" (PROPAX))))


gasburner.lem1: PROVED

("" (APPLY (THEN* (DCINIT) (FLATTEN)))
    (CHOP-INDUCT "LAMBDA(f:IForm):Des_1/\Des_2/\f => Triple"
                 "Leak")
    (("1" (APPLY (THEN* (BOX-LEFT)
                        (ASSERT)
                        (PROPAX))))
      ("2" (BOX-RIGHT)
           (APPLY
             (THEN* (EXPAND "empty")
                    (EXPAND "Triple")
                    (EXPAND "T")
```

```
                   (FLATTEN)))
          (APPLY (THEN* (DCLEMMA "Thm1b")
                        (ASSERT))))
     ("3" (SPLIT)
          (("1" (APPLY (THEN* (BOX-LEFT) (GROUND))))
            ("2" (APPLY (THEN* (DCLEMMA "lem1_2")
                               (GROUND))))
            ("3" (APPLY (THEN* (DCLEMMA "lem1_3")
                               (GROUND)))))))))
```

```
gasburner.lem2_1_TCC1: PROVED

("" (AUTO-REWRITE-EXPLICIT)
    (GROUND))
```

```
gasburner.lem2_1: PROVED

("" (DCINIT)
    (FLATTEN)
    (EXISTS-INTRO "Leak")
    (INTRO-EXISTS "Leak")
    (EXPAND "len")
    (EXISTS-INTRO "tt")
    (EXPAND "Safe")
    (EXPAND "len")
    (INTRO-EXISTS "tt")
    (DC-CASE "dur(tt) = n(r!3);dur(tt) = n(r!4)")
    (("1" (DC-CASE "dur(Leak) = n(r!1);dur(Leak) = n(r!2)")
          (("1" (AXIOM4)
                (AXIOM4)
                (DC-CASE "(n(30)*n(r!1)<=n(r!3));(n(r!4)>=n(0)/\n(r!2)<=n(1))")
                (("1" (FREE-CHOP)
                      (FLATTEN)
                      (CHOP-FREE)
                      (FLATTEN)
                      (ASSERT))
                  ("2" (HIDE -1 -2)
                       (MONO!)
                       (("1" (ASSERT)) ("2" (ASSERT))))))
            ("2" (HIDE -1) (MONO!))))
     ("2" (MONO!))))
```

```
gasburner.lem2_2: PROVED

("" (DCINIT)
    (FLATTEN)
    (MONO-CHOP "(dur(Leak)<=len/n(30))")
    (("1" (BOX-RIGHT)
          (EXPAND "T")
          (ASSERT))
     ("2" (HIDE -3)
          (CHOP-MONO "dur(Leak)<=n(1)")
          (("1" (EXPAND "Des_1")
                (BOX-RIGHT)
                (BOX-LEFT)
                (ASSERT)
                (EXPAND "len")
                (DCLEMMA "axiom3")
                (QUANT * ("Leak" "not Leak"))
                (CASE "(Leak /\ (not Leak)) = ff")
                (("1" (CASE "(Leak \/ (not Leak)) = tt")
```

```
                        (("1" (REPLACE -1)
                              (REPLACE -2)
                              (REWRITE "axiom1")
                              (EXPAND "pr")
                              (EXPAND "len")
                              (GROUND))
                          ("2" (APPLY-EXTENSIONALITY))))
                  ("2" (APPLY-EXTENSIONALITY))))
            ("2" (DCLEMMA "lem2_1")
                 (ASSERT)
                 (PROPAX))))))
```

gasburner.lem2_3: PROVED

```
("" (DCINIT)
    (FLATTEN)
    (ASSOC)
    (CHOP-MONO "dur(Leak) <= n(1)")
    (("1" (EXPAND "Des_1")
          (BOX-RIGHT)
          (DC-CASE "('Leak'/\len<=n(1));dur(Leak)=n(0)")
          (("1" (HIDE -2 -3)
                (EXPAND "pr")
                (EXPAND "len")
                (EXISTS-INTRO "tt")
                (INTRO-EXISTS "tt")
                (DC-CASE "dur(Leak) = n(r!1);dur(Leak) = n(0)")
                (("1" (DC-CASE "dur(tt) = n(r!1);dur(tt) = n(r!2)")
                      (("1" (AXIOM4)
                            (AXIOM4)
                            (DC-CASE "(n(r!1)<=n(1));TRUE")
                            (("1" (FREE-CHOP)
                                  (FLATTEN)
                                  (ASSERT))
                              ("2" (MONO!) (ASSERT))))
                        ("2" (HIDE -1) (MONO!))))
                  ("2" (MONO!) (ASSERT))))
            ("2" (MONO)
                 (("1" (BOX-RIGHT)
                       (BOX-LEFT)
                       (GROUND))
                   ("2" (BOX-RIGHT)
                        (DCLEMMA "Thm3a"
                                 ("p" "not Leak"))
                        (ASSERT)
                        (PROPAX))))))
      ("2" (EXPAND "T")
           (DCLEMMA "lem2_1")
           (GROUND)
           (MONO! -4)
           (ASSERT))))
```

gasburner.lem2: PROVED

```
("" (DCINIT)
    (FLATTEN)
    (EXPAND "Triple")
    (SPLIT)
    (("1" (EXPAND "Safe")
          (EXPAND "T")
          (ASSERT))
      ("2" (DCLEMMA "lem2_2")
```

```
          (ASSERT)
          (PROPAX))
     ("3" (DCLEMMA "lem2_3")
          (ASSERT)
          (PROPAX))))


gasburner.Gasburner: PROVED

("" (DCINIT)
    (FLATTEN)
    (DCLEMMA "lem1")
    (DCLEMMA "lem2")
    (ASSERT)
    (PROPAX))
```

C.4 Proof Chain Analysis

Below follows the result of doing a proof chain analysis on the `Gasburner` lemma. The output is obtained by positioning the cursor at the lemma and typing `M-x status-proofchain`. The listing contains the gasburner lemmas as well as PC/DC inference rules and the definitions from the encoding of DC in PVS. Also definitions and axioms from the PVS prelude file are shown.

```
Gasburner has been PROVED.

The proof chain for Gasburner is COMPLETE.

Gasburner depends on the following proved theorems:
   dcp_thms.Thm1a
   dcp_cf_reverse.cfc4
   dcp_also_rewrite.IForm_Impl
   dcp_rules.chop_false
   dcp_also_rewrite.and_not
   dcp_rules.chop_free_true_dc
   gasburner.lem1_3_4
   dcp_thms.pr_diamond_pr
   dcp_rules.mono_bang
   dcp_also_rewrite.n1
   dcp_also_rewrite.DiamondFalse
   gasburner.lem1_3
   dcp_rules.disj_chop_dc
   gasburner.lem2_1
   dcp_rules.dur_preserve
   dcp_thms.pred_box
   gasburner.lem1
   dcp_rules.chop_mono
   gasburner.lem2
   dcp_thms.lowbnd
   dcp_rules.box_left
   dcp_rules.dur_lemma
   dcp_rules.axiom3_dc
   dcp_also_rewrite.BoxTrue
   dcp_thms.dur_add_gt_ge
   gasburner.lem2_3
```

```
   dcp_thms.Thm3a
   dcp_rules.mono
   dcp_context_free.cfc3
   dcp_also_rewrite.int_or_drop
   dcp_rules.box_right
   dcp_thms.Thm1b
   dcp_thms.Thm6_len
   dcp_also_rewrite.LemFalse
   dcp_cf_reverse.cfc1
   dcp_rules.intro_exists
   dcp_also_rewrite.LemTrue
   dcp_context_free.cfb1
   gasburner.lem1_2
   dcp_rules.dur_preserve_TCC2
   dcp_also_rewrite.IForm_Iff
   gasburner.lem1_1
   dcp_rules.mono_chop
   gasburner.lem1_2_1
   dcp_context_free.cf1
   dcp_also_rewrite.n3
   gasburner.lem1_2_3
   dcp_rules.mono_chop_bang
   gasburner.lem2_2
   dcp_context_free.cfc4
   dcp_rules.chop_mono_bang
   gasburner.lem1_3_1
   gasburner.lem1_3_2
   dcp_rules.axiom4_dc
   dcp_also_rewrite.not_not
   dcp_rules.free_chop_true_dc
   gasburner.lem1_3_3
   dcp_rules.assoc
   dcp_rules.mono_chop_mono
   dcp_rules.intro_exists_intro
   dcp_also_rewrite.n2
   dcp_thms.Thm6a
   dcp_context_free.cfc1
   dcp_rules.false_chop
   dcp_also_rewrite.or_not
   dcp_thms.dur_add_ge_gt
   dcp_cf_reverse.cfc3
   dcp_rules.exists_intro
   gasburner.lem1_2_5
   dcp_also_rewrite.BoxConj
   reals.divides_TCC1
   dcp_also_rewrite.int_and_true
   dcp_thms.Thm6
   reals.closed_divides_TCC1
   dcp_also_rewrite.n4_TCC1
   gasburner.lem2_1_TCC1
   dcp_rules.assoc_dc
   gasburner.lem1_2_4
   dcp_rules.dur_preserve_TCC1
   reals.times_TCC1
   gasburner.lem1_2_2
   reals.plus_TCC1
   reals.minus_TCC1
   dcp_cf_reverse.cf1
   dcp_also_rewrite.n4

Gasburner depends on the following axioms:
   dcp_rules.axiom1
   dcp_rules.axiom4
   dcp_rules.axiom3
```

```
   number_axioms.trichotomy
   dcp_rules.axiom2
   dcp_rules.chop_induct

Gasburner depends on the following definitions:
   dcp_not_rewrite.<>
   dcp_defs.<
   dcp_defs.*
   dcp_not_rewrite.[]
   reals./
   dcp_defs.cf
   reals.*
   dcp_defs.\/
   dcp_equality.=
   dcp_defs.cf
   dcp_rules.*
   dcp_not_rewrite.^
   gasburner.Triple
   dcp_defs.<=
   dcp_defs.tt
   gasburner.Safe
   dcp_not_rewrite.FALSE
   gasburner.T
   dcp_rules.rule
   dcp_thms.m
   dcp_not_rewrite.TRUE
   dcp_rules.rule
   dcp_not_rewrite.=>
   dcp_not_rewrite.empty
   dcp_quant.ex
   gasburner.Des_2
   dcp_not_rewrite.<=>
   dcp_defs.\/
   dcp_not_rewrite.len
   dcp_defs.>=
   dcp_defs.+
   dcp_defs.ff
   dcp_defs./\
   dcp_defs.valid
   dcp_rules.+
   numbers.>
   dcp_not_rewrite.pr
   dcp_rules.rule
   dcp_defs.NOT
   dcp_defs.NOT
   dcp_defs.>
   numbers.<=
   numbers.>=
   reals.+
   dcp_defs.n
   dcp_defs./\
   dcp_defs./
   notequal./=
   gasburner.Des_1
   dcp_defs.-
   reals.-
   dcp_rules.seq
```