PC/DC Users Guide

Søren T. Heilmann

October 2, 1998

Contents

Chapter 1

Introduction

PC/DC is the Proof Checker for Duration Calculus, a software tool used to capture and verify properties of systems specified in the real time logic Duration Calculus (DC). As a tool it builds heavily on the powerful proof system PVS [1]. Duration Calculus was originally developed by Zhou, Hoare and Ravn [1]. A concise reference to the foundation, development and properties of the logic is [2].

PC/DC was originally developed by Jens Ulrik Skakkebæk for version 1 of PVS and described in his PhD-thesis [4]. The advent of later versions of PVS (PVS2α+ and PVS2.1) has necessitated a port of the tool to this new platform during which small improvements of the tool has been done. The basic ideas and implementation however is unchanged and serves as an example of the work necessary to extend PVS with convenient support for a specific logic like DC.

This document serves multiple purposes. In chapter 2 it will explain how to obtain and install the software.

Knowledge of PVS is a must for any user of PC/DC and the reader is referred to the PVS documentation ([3]). However a small guide for getting started with using the tool is included in chapter 3.

As an extension of PVS the tool offers a DC specific syntax for formulas and theories as well as a library of definitions. These are explained in chapter 4.

A major part of the extension is the prover commands available in PC/DC. The bulk of this document is a description of the syntax and effect of these commands. This can be found in chapter 5.

1.1 Changes for PVS 2.2

To ease future maintenance for PC/DC the special DC parser/unparser originally build into PC/DC has been removed form the tool as part of the port to PVS 2.2. The information in the following chapter has *not* been updated to reflect this change. In stead the follow shortly describes the changes:

Installation PC/DC now builds on PVS 2.2 and you must have this version of PVS installed before installing PC/DC. Otherwise the installation procedure is the same.

Getting Started Any mention of `.dc` files is obsolete. DC theories are now specified as normal PVS theories and stored in `.pvs` files. A DC theory should however import the `dc` library stored in *pvs-path*`/dc/lib/dc` and import the theory `dc@dc_top`. Also note that a DC proof should be started with `(DC-INIT)` as the first proof command.

[1] `http://www.csl.sri.com/pvs.html`

Concrete Syntax The section on concrete syntax no longer apply. This means that the second column in the tables in this section is obsolete. Always use the definitions in the third column.

Prover Commands In the description of the prover commands `;` is used as chop operator, this is obsolete use `^` in stead. The syntax `'P'` for state predicates is also obsolete, use `pr(P)`. The use of `SOME` and `ALL` as quantifiers is obsolete, use `ex !` and `all !` in stead, respectively.

Chapter 2

Installation

2.1 Getting the PC/DC software

You can obtain the software either via anonymous ftp at `ftp.it.dtu.dk/pub/sh` or on the WWW at `www.it.dtu.dk/~sh`. The Web address is also the place to look for any news regarding the software, new releases, bug fixes etc. The software is stored as a gzip compressed tar file `pcdc2_xxxxxx.tgz`. `xxxxxx` is the release date. As a rule get the latest release available.

2.2 Installing the PC/DC software

PC/DC builds on PVS2.1 so the requirement is that you already have PVS2.1 installed. Furthermore you should have access permission to make changes to the PVS2.1 installation as PC/DC installs into the PVS2.1 directory tree. If you do not have such permissions look at section 2.4 for a way around this.

Assuming you have obtained the software as described above you install the software by following these few steps:

1. Find out the complete file path of your PVS 2 installation. This path is denoted *pvs-path* in the following. In a networked installation you should choose a path accessible from the network.

2. Change directory to *pvs-path*.

3. Make a directory called `dc` for the PC/DC software in the *pvs-path* directory.

4. Copy the file `pcdc2_xxxxxx.tgz` to the *pvs-path*/`dc` directory.

5. Change directory to *pvs-path*/`dc`.

6. Unpack the archive. Do: `gunzip -c pcdc2_xxxxxx.tgz | tar xvf -` . You should see a list of files scroll across the screen. If you do not have `gunzip` on your system try using `unzip` instead.

7. Merge the file `load-dc` with the `.pvsemacs` file in your home directory. In case you do not have such a file simply create it by copy `load-dc` into it. The `load-dc` file defines the Emacs commands used to start PC/DC from within PVS. You can not do without it.

8. PC/DC is now installed. The next section describes how to run the software.

2.3 Running the PC/DC software

Once installed you start PC/DC from within PVS by issuing the command `M-x load-dc`. When ready it will respond with the message `PC/DC ready`.

You will find the user documentation for PC/DC in *pvs-path*`/dc/doc` as Postscript files. Print them out.

Should you have any problems installing and running the software or any comments in general do not hesitate to write me at `sh@it.dtu.dk`. I will need to know which version you are using. Do `M-x dc-version` to display the PC/DC release date.

2.4 Creating a shadow installation of PVS

In case you do not have access permission to do the changes in the PVS installation required in 2.2 here is a little trick that might help you. This may also be useful if you want to have different version of PC/DC installed at the same time.

The idea is simply to create a directory containing links to all the topmost files and directories of the PVS installation and then to install PC/DC in this directory. Do the following steps:

1. Decide on a directory to put the shadow installation into. This should of cause be somewhere in the directory tree which you have permission to access and change. Call the directory *my-pvs-path* in the following.
2. Create the *my-pvs-path* directory.
3. Change directory to *my-pvs-path*.
4. Create links: `ln -s` *pvs-path*`/* .`
5. Copy `pvs` executable. `cp -f` *pvs-path*`/pvs .`
6. Make it edit-able. `chmod u+w pvs`
7. Edit `pvs` to change the `PVSPATH` variable to point to *my-pvs-path* instead of *pvs-path*.
8. Install PC/DC as describe in section 2.2 but in the *my-pvs-path* directory.

Chapter 3

Getting Started

Chances are that you already know PVS quite well. If not you will have to acquaint yourself with the PVS documentation [3]. This chapter will only supply you with the most rudimentary information needed to use PC/DC. If you know PVS there will only be few surprises in the use of PC/DC but you still have to get to know the different PC/DC proof commands.

Specifications are held in DC theory files (with suffix `.dc`) placed in the context directory along side the different kinds of PVS files. DC theory files are manipulated using all the Emacs commands known from PVS.

To get an idea of how PC/DC works follow the script below for a quick tour.

Start PVS in a directory where you have write permissions and load PC/DC (`M^x load-dc`). Create a new DC theory using the `M^x new-pvs-file` command. Call it `example.dc`. Up comes a buffer with a skeleton for a DC theory. Make it look like the following either by editing or by including the file *pvs-path*`/examples/user-guide-examples/example.dc`.

```
example : DCTHEORY

  BEGIN

    p: VAR State
    r,s: VAR nonneg_real

    gt_chop_lemma :
        LEMMA  ((dur(p) > n(r));(dur(p) > n(s))) => (dur(p) > n(r)+ n(s))

  END example
```

As you can see a DC theory is not much different from a normal PVS theory except for the DCTHEORY keyword. This particular theory declares a couple of variables and the lemma `gt_chop_lemma`. PVS cannot do much with a theory until it has been type checked. Try this using the command `M^x typecheck`[1]. Having type checked the theory we can go on to proving the lemma. Place the cursor somewhere in the `gt_chop_lemma` name and issue the command `M^x prove`.

The screen splits into two displaying the `*pvs*` buffer. Interactive proofs are performed in this buffer using, among others, the prover commands found in chapter 5. The `*pvs*` buffer should look something like this:

[1]There is a number of short cut keys available for the most frequently used commands. See the PVS documentation or use the command `M^x help-pvs`.

```
gt_chop_lemma :

{-1}    ((dur(p!1) > r!1);(dur(p!1) > s!1))
  |-------
{1}    dur(p!1) > r!1 + s!1

Rule?
```

This is what a proof goal sequent looks like, a collection of numbered formulas with a turnstyle symbol separation the antecedent from the consequent formulas. The formula number is used to designate the formulas when issuing prover commands.

As you can see the global variables `r`, `s` and `p` have been skolemized to the constants `r!1`, `s!1` and `p!1`. Furthermore the implication has been flattened moving the premise to the antecedent. This is done automatically as part of setting up the lemma up for proving.

The cursor is placed to the right of the prover command prompt. We will proceed by binding the duration of `p!1` to a couple of constants `r!3` and `r!3`.

```
Rule? (EXISTS-INTRO "p!1")
Introducing left existential binding of duration measure of p!1,
this simplifies to:
gt_chop_lemma :

{-1}    (dur(p!1) = r!2 /\ (dur(p!1) > r!1));(dur(p!1) > s!1)
  |-------
[1]    dur(p!1) > r!1 + s!1

Rule? (INTRO-EXISTS "p!1")
Introducing right existential binding of duration measure of p!1,
this simplifies to:
gt_chop_lemma :

{-1}    (dur(p!1) = r!2 /\ (dur(p!1) > r!1));(dur(p!1) = r!3 /\ (dur(p!1) > s!1))
  |-------
[1]    dur(p!1) > r!1 + s!1
```

We want to utilize that durations accumulate across chop. For this we first add a simplified antecedent formula using a case split. This simplified form is required by the `AXIOM4` rule.

```
Rule? (DC-CASE "(dur(p!1) = n(r!2));(dur(p!1) = n(r!3))")

DC case splitting on
(dur(p!1) = n(r!2));(dur(p!1) = n(r!3)),
this yields  2 subgoals:
gt_chop_lemma.1 :

{-1}    ((dur(p!1) = r!2);(dur(p!1) = r!3))
[-2]    (dur(p!1) = r!2 /\ (dur(p!1) > r!1));(dur(p!1) = r!3 /\ (dur(p!1) > s!1))
  |-------
[1]    dur(p!1) > r!1 + s!1

Rule? (AXIOM4)
Rewriting using Axiom 4,
this simplifies to:
gt_chop_lemma.1 :
```

```
{-1}    (dur((p!1)) = (r!2) + (r!3))
[-2]    (dur(p!1) = r!2 /\ (dur(p!1) > r!1));(dur(p!1) = r!3 /\ (dur(p!1) > s!1))
  |-------
[1]    dur(p!1) > r!1 + s!1
```

Now we have established that the duration of `p` equals `r!2+r!3` on the entire interval. Next we want to propagate the relations `r!2 > r!1` and `r!3> s!1` to the top level. We can do this because the relations are context free i.e. independent of the interval. Again we setup a simplified formula using a case split. Then we propagate the relations to the top level and finish this subgoal using the `ASSERT` command.

```
Rule? (DC-CASE "(n(r!2) > n(r!1));(n(r!3)> n(s!1))")

DC case splitting on
(n(r!2) > n(r!1));(n(r!3)> n(s!1)),
this yields  2 subgoals:
gt_chop_lemma.1.1 :

{-1}    ((r!2 > r!1);(r!3 > s!1))
[-2]    (dur((p!1)) = (r!2) + (r!3))
[-3]    (dur(p!1) = r!2 /\ (dur(p!1) > r!1));(dur(p!1) = r!3 /\ (dur(p!1) > s!1))
  |-------
[1]    dur(p!1) > r!1 + s!1

Rule? (FREE-CHOP)
Propagating left context free expression through chop,
this simplifies to:
gt_chop_lemma.1.1 :

{-1}    (r!2 > r!1 /\ (TRUE;(r!3 > s!1)))
[-2]    (dur((p!1)) = (r!2) + (r!3))
[-3]    (dur(p!1) = r!2 /\ (dur(p!1) > r!1));(dur(p!1) = r!3 /\ (dur(p!1) > s!1))
  |-------
[1]    dur(p!1) > r!1 + s!1

Rule? (FLATTEN)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
gt_chop_lemma.1.1 :

{-1}    r!2 > r!1
{-2}    (TRUE;(r!3 > s!1))
[-3]    (dur((p!1)) = (r!2) + (r!3))
[-4]    (dur(p!1) = r!2 /\ (dur(p!1) > r!1));(dur(p!1) = r!3 /\ (dur(p!1) > s!1))
  |-------
[1]    dur(p!1) > r!1 + s!1

Rule? (CHOP-FREE)
Propagating right context free expression through chop,
this simplifies to:
gt_chop_lemma.1.1 :

[-1]    r!2 > r!1
{-2}    (r!3 > s!1 /\ (TRUE;TRUE))
```

```
[-3]    (dur((p!1)) = (r!2) + (r!3))
[-4]    (dur(p!1) = r!2 /\ (dur(p!1) > r!1));(dur(p!1) = r!3 /\ (dur(p!1) > s!1))
  |-------
[1]    dur(p!1) > r!1 + s!1

Rule? (FLATTEN)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
gt_chop_lemma.1.1 :

[-1]    r!2 > r!1
{-2}    r!3 > s!1
{-3}    (TRUE;TRUE)
[-4]    (dur((p!1)) = (r!2) + (r!3))
[-5]    (dur(p!1) = r!2 /\ (dur(p!1) > r!1));(dur(p!1) = r!3 /\ (dur(p!1) > s!1))
  |-------
[1]    dur(p!1) > r!1 + s!1

Rule? (ASSERT)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of gt_chop_lemma.1.1.
```

We are left with two subgoals from the two case splits we did above. But these are simple consequences of the monotonicity of chop. This concludes the proof in the following way:

```
gt_chop_lemma.1.2 :

[-1]    (dur((p!1)) = (r!2) + (r!3))
[-2]    (dur(p!1) = r!2 /\ (dur(p!1) > r!1));(dur(p!1) = r!3 /\ (dur(p!1) > s!1))
  |-------
{1}    ((r!2 > r!1);(r!3 > s!1))
[2]    dur(p!1) > r!1 + s!1

Rule? (MONO!)
Applying monotonicity,

This completes the proof of gt_chop_lemma.1.2.

This completes the proof of gt_chop_lemma.1.

gt_chop_lemma.2 :

[-1]    (dur(p!1) = r!2 /\ (dur(p!1) > r!1));(dur(p!1) = r!3 /\ (dur(p!1) > s!1))
  |-------
{1}    ((dur(p!1) = r!2);(dur(p!1) = r!3))
[2]    dur(p!1) > r!1 + s!1

Rule? (MONO!)
Applying monotonicity,

This completes the proof of gt_chop_lemma.2.

Q.E.D.
```

Chapter 4

Specification Language

The Duration Calculus is encoded in the logic of PVS. The encoding is a *shallow* semantic encoding. This means that the semantics or meaning of DC formulas, terms and states are specified in PVS as types and functions. A *deep* encoding would specify also the syntax of DC and concept of DC proofs.

Having no specification of the syntax of DC means that the concept of a DC theorem in the traditional sense ([2]) is also not captured.

4.1 Encoding

The encoding is contained in a PVS library called `dc` located in *pvs-path*`/dc/lib/dc`. It is a good idea to acquaint yourself with the contents of this library. The following presents some of the main definitions.

Semantic Domains The basic domains or types of a semantic model of DC are those of time and interval. Modeling these domains is easy in PVS:

```
Time : TYPE = real

b, e : VAR Time
Interval : TYPE = { (b, e) | b <= e}
```

The meaning of terms are functions of intervals, especially real terms are real-valued functions and formulas are boolean functions:

```
T : TYPE+
Term : TYPE = [Interval -> T]

IReal : TYPE = Term[real]
IForm : TYPE = Term[bool]
```

The meaning of state expressions and state variables are finitely variable functions of time. The property of finite variability is explicitly defined.

```
T : TYPE+
f : VAR [Time -> T]

State : State = { f | Finite_Variability(f)}

Finite_Variability(f) = ...
```

Logical Connectives The logical connectives (or rather their meaning) is encoded explicitly. There are connectives for both formulas and state expressions.

```
i : VAR Interval
A, B : VAR IForm

/\(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);

t : VAR Time
ta, tb : VAR [Time -> bool]

/\(ta,tb)(t) : bool = ta(t) AND tb(t);
\/(ta,tb)(t) : bool = ta(t) OR tb(t);
not(ta)(t) : bool = NOT ta(t);

tt(t) : bool = TRUE;
ff(t) : bool = FALSE;
```

Real-valued Operators The most prominent real valued operator is of course the duration operator $\int$. Notice that this is encoded *implicitely* as a non-negative (`NN`) term. The lenght of an interval is then easily defined:

```
dur(s) : INNReal

len : INNReal = dur(tt)
```

Other real valued operators are the arithmetic operators `+`, `-`, `*` and `/` and the function `n` to lift a real constant to a DC term.

Boolean Operators Apart from the logical connectives the boolean operators include equality (`=`) and the relational operators on reals (`>`, `>=`, `<` and `<=`). And also the constants `true` and `false`.

State predicates are encoded as abbreviations. Notice the explicite lifting of the real literal `0` to a DC term using `n`.

```
pr(s) : IForm = dur(s) = len /\ (len > n(0));
point : IForm = (len = n(0));
```

Modalities The modalities are encoded as follows:

```
^(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
```

Quantifiers The DC quantifiers are encoded as functions in the following way:

```
T : TYPE+

i : VAR Interval
A : VAR [T -> IForm]
```

```
fall(A)(i) : bool = (FORALL (r:T): A(r)(i));
ex(A)(i) : bool = (EXISTS (r:T) : A(r)(i));
```

The meaning of a formula like $\exists r \bullet l = r$ is thus represented by `ex(LAMBDA (r:real): len=n(r))`. The awkwardness of this notation is alleviated through as special DC parser.

Validity

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

4.2 Concrete Syntax

PVS and the `dc` library are the essentials needed to reason about DC specifications. What is lacking, however, is a convinient notation for DC specifications to avoid the cumbersome PVS formulations for DC quantification and state predicates. This is why a dedicated DC parser has been build into PC/DC.

The DC parser parses DC theories. The concrete syntax of DC theories is so close to that of PVS theories that a complete grammar will not be presented. Rather we highlight the few differences.

First of all the DCTHEORY keyword replaces the THEORY keyword to indicate a DC theory. Secondly a few constructs is added to the PVS expression syntax. These are presented in the table below. Thirdly, due to the use of `;` as the chop operator the semicolon used to seperate declarations in PVS theories is replaced by a double semicolon (`;;`).

Math	ASCII	Definition
$;$	`;`	`^`
$\forall$	`ALL`	`fall`
$\exists$	`SOME`	`ex`
$\lceil\bullet\rceil$	`'•'`	`pr`
$\longrightarrow$	`-->`	`fby`
$\xrightarrow{\bullet}$	`-:•:->`	`lto`
$\xrightarrow{\leq\bullet}$	`-:<=•:->`	`upto`

For the reader familiar with the DC notation used in papers and reports on DC we offer the following conversion table to the ASCII syntax:

Math	ASCII	Definition
$\wedge$	`/\`	`/\`
$\vee$	`\/`	`\/`
$\neg$	`NOT`	`not`
$\Rightarrow$	`=>`	`=>`
$\Leftrightarrow$	`<=>`	`<=>`
$;$	`;`	`^`
$\Box$	`[]`	`[]`
$\Diamond$	`<>`	`<>`
$\forall$	`ALL`	`fall`
$\exists$	`SOME`	`ex`
$\int$	`dur`	`dur`
$\lceil\rceil$	`point`	`point`
$\lceil\bullet\rceil$	`'•'`	`pr`
$\longrightarrow$	`-->`	`fby`
$\xrightarrow{\bullet}$	`-:•:->`	`lto`
$\xrightarrow{\leq\bullet}$	`-:<=•:->`	`upto`

Chapter 5

Prover Commands

This chapter describes the prover commands specific to PC/DC. Descriptions of other commands can be found in the PVS documentation.

The descriptions are automatically generated[1] from the help texts available in PVS using the command `help-pvs-prover`. This ensures consistency (but unfortunately, not correctness).

5.1 Command Parameters

The prover commands share a small number of different parameter types. These are explained here.

`fnums`, `fnumsl`, `fnumsr`

Used to specify the sequent formulas in which to search for a formula of a specified kind. Searching is from the top down and stops at the first formula satisfying the search criteria. Possible values are:

*	Search in all sequent formulas
-	Search in all antecedent formulas
+	Search in all consequent formulas
integer	Search only in the formula designated by the integer
integer list	ie. (integer integer ... integer). Search only among formulas designated by integers in the list.

`cnum`, `cnuml`, `cnumr`

A natural number designating a chop operator in a formula. The designated chop is the `cnum`'th chop counting from the left as examplified below:

`cnum`	Designated chop
1	$A\underline{;}B;C$
3	$A;B;C\underline{;}D$
3	$A;(B;C)\underline{;}D$
1	$((A_1;A_2 \wedge A_3))\underline{;}B;C$

`dir`

Specifies the direction of rewriting. Possible values are:

[1] Hopefully the typographical layout and type setting resulting from this generation will improve over time.

`lr`	left to right
`rl`	right to left

`count`

A natural number. Often used to give the number of times some rewrting shall be repeated.

`dcfmla`

String parsed as a DC formula using the DC parser.

`state`

String parsed as a DC state expression using the DC parser.

`type`

String parsed as a PVS type expression.

`i`

String parsed as a PVS expression designating the context interval to use when applying a DC rule.

5.2 Miscellaneous

Command	*Function*
`dc-sem`	Toggle hiding of semantics
`dc-init`	Initialize DC goal
`dc-expand`	Expand DC definition
`dc-lemma`	Introduce DC lemma
`dc-case`	Case split on DC formula
`dc-decide`	Apply DC decision procedure

`dc-sem`: Toggle hiding of semantics

syntax: `(dc-sem)`

effect: `Toggle hiding of semantics on/off.`

`dc-init`: Initialize DC goal

syntax: `(dc-init)`

arguments:

```
Initializes the prover for proving DC theorems. Installs
automatic rewriting of DC operators etc.
```

effect:
```
A customized definition of this proof command may be kept
in the file pvs-strategies in the current context directory.
Such a definition will override this one.
```

`dc-expand:` Expand DC definition

syntax: `(dc-expand name &optional (fnums *))`

effect: `Expands definitions in PC/DC and does rewriting.`

`dc-lemma:` Introduce DC lemma

syntax: `(dc-lemma name &optional subst i)`

effect: `Adds DC lemma named <name> to the premiss`

`dc-case:` Case split on DC formula

syntax: `(dc-case dcfmla &optional i)`

arguments:
`<dcfmla> is a DC formula.`
`<i> is the context interval.`

effect: `Does a case split on the DC formula <dcfmla> producing two subgoals. One with <dcfmla> added to the antecedent and one with <dcfmla> added to the consequent.`

`dc-decide:` Apply DC decision procedure

syntax: `(dc-decide &optional (fnums *))`

arguments:
`<fnums> designates the formulas on with to run the decision procedure.`

effect: `Applies the DC decision precedure on the formulas <fnums>.`

5.3 Basic Properties of Chop

Command	*Function*
`false-chop`	Propagate left false through chop
`chop-false`	Propagate right false through chop
`chop-false-chop`	Propagate center false through chop
`assoc`	Apply associativity of chop
`left-assoc`	Left associate chops
`right-assoc`	Right associate chops
`top-assoc`	Lift chop to top
`center-assoc`	Lift chops to top
`disj-chop`	Distribute chop over left disjuction
`chop-disj`	Distribute chop over right disjuction
`chop-disj-chop`	Distribute chop over center disjuction
`chop-and`	Distribute conjunction over chop
`and-chop`	Distribute conjunction over chop

`false-chop:` Propagate left false through chop

syntax: `(false-chop &optional (fnumsl -) i)`

arguments:
`<fnumsl> designates a formula in the antecedent of the form FALSE;A.`
`<i> is the context interval.`

effect: Propagates FALSE to the left of the chopped formula <fnumsl> through to the antecedent thus proving the goal.

chop-false: Propagate right false through chop

syntax: (chop-false &optional (fnumsl -) i)

arguments:
<fnumsl> designates a formula in the antecedent of the form A;FALSE.
<i> is the context interval.

effect: Propagates FALSE to the right of the chopped formula <fnumsl> through to the antecedent thus proving the goal.

chop-false-chop: Propagate center false through chop

syntax: (chop-false-chop &optional (fnumsl *) i)

arguments:
<fnumsl> designates a formula in the antecedent of the form A;FALSE;B.
<i> is the context interval.

effect: Propagates FALSE in the center of the chopped formula <fnumsl> through to the antecedent thus proving the goal.

assoc: Apply associativity of chop

syntax: (assoc &optional (fnums *) (dir lr))

arguments:
<fnums> designates a formula of the form A;B;C or A;(B;C) depending of <dir>.
<dir> is the direction of rewriting. LR: left to right. RL: right to left.

effect: Applies associativity of chop to rewrite formula <fnums>.
Rewrites A;B;C to A;(B;C) when <dir> is LR.
Rewrites A;(B;C) to A;B;C when <dir> is RL.

left-assoc: Left associate chops

syntax: (left-assoc &optional (fnums *) (count))

arguments:
<fnums> designates a formula.
<count> is a positive integer.

effect: Rewrites chopped formula <fnums> in to left associative form using the equivalence A;B;C <=> A;(B;C). Rewrites right to left, outside in. To rewrite left to right use right-assoc.

If <count> is given rewrite <count> number of times else rewrite repeatedly.

`right-assoc:` Right associate chops

syntax: `(right-assoc &optional (fnum *) (count))`

arguments:

```
<fnums> designates a formula.
<count> is a positive integer.
```

effect: Rewrites chopped formula <fnums> in to right associative form using the equivalence A;B;C <=> A;(B;C). Rewrites left to right, outside in. To rewrite right to left use left-assoc.

If <count> is given rewrite <count> number of times else rewrite repeatedly.

`top-assoc:` Lift chop to top

syntax: `(top-assoc cnum &optional (fnums *))`

arguments:

```
<cnum> designates a chop in <fnums>
<fnums> designates a chopped formula.
```

effect: Applies associativity of chop to lift the chop designates by <cnum> in the formula <fnums> to the top. Resulting for example in a formula (A;B;C);(D;E;F;G) for <cnum> equal 3.
The left and right side of <cnum> are rewritten into left associative form.

`center-assoc:` Lift chops to top

syntax: `(center-assoc cnuml cnumr &optional (fnum *))`

arguments:

```
<cnuml> designates a left chop in <fnums>
<cnumr> designates a right chop in <fnums>
<fnums> designates a chopped formula.
```

effect: Applies associativity of chop to lift the chops designates by <cnuml> and <cnumsr> in the formula <fnums> to the top.
Resulting for example in a formula
(A;B);(C;D);(E;F;G) for <cnuml> equal 2 and <cnumr> equal 4.
The left, right and center parts of <fnums> are rewritten into left associative form.

`disj-chop:` Distribute chop over left disjuction

syntax: `(disj-chop &optional (fnums *) (dir lr))`

arguments:

```
<fnums> designates a formula of the form (A\/B);C or A;C \/ B;C
  depending of <dir>.
<dir> is the direction of rewriting.
```

effect: Distributes chop over left disjunction in <fnums> using the equivalence:
  (A\/B);C <=> A;C \/ B;C.
<dir> gives the direction of rewriting.

`chop-disj`: Distribute chop over right disjuction

syntax: `(chop-disj &optional (fnums *) (dir lr))`

arguments:

```
<fnums> designates a formula of the form A;(B\/C) or A;B \/ A;C
  depending of <dir>.
<dir> is the direction of rewriting.
```

effect: `Distributes chop over right disjunction in <fnums> using the equivalence:`

```
   A;(B\/C) <=> A;B \/ A;C
<dir> gives the direction of rewriting.
```

`chop-disj-chop`: Distribute chop over center disjuction

syntax: `(chop-disj-chop &optional (fnums *) (dir lr))`

arguments:

```
<fnums> designates a formula of the form A;(B\/C);D or A;B;D \/ A;C;D
  depending of <dir>.
<dir> is the direction of rewriting.
```

effect: `Distributes chop over center disjunction in <fnums> using the equivalence:`

```
   A;(B\/C);D <=> A;B;D \/ A;C;D
<dir> gives the direction of rewriting.
```

`chop-and`: Distribute conjunction over chop

syntax: `(chop-and &optional (fnumsl *) (fnumsr *) (dir rl) i)`

arguments:

```
<fnumsl> designates a formula of the form A;(len=r /\ B)
<fnumsl> designates a formula of the form C;(len=r /\ D)
<dir> is the direction of rewriting.
<i> is the context interval.
```

effect: `Introduces the equivalence:`

```
  A;(len=r /\ B) /\ C;(len=r /\ D) <=> (A /\ C);(len=r /\ B /\ D)
into the antecedent and does propositional simplification using ground.

Direction LR is not implemented.
```

`and-chop`: Distribute conjunction over chop

syntax: `(and-chop &optional (fnumsl *) (fnumsr *) (dir rl) i)`

arguments:

```
<fnumsl> designates a formula of the form (len=r /\ A);B
<fnumsl> designates a formula of the form (len=r /\ C);D
<dir> is the direction of rewriting.
<i> is the context interval.
```

effect: `Introduces the equivalence:`

```
  (len=r /\ A);B /\ (len=r /\ C);D <=> (len=r /\ A /\ C);(B /\ D)
into the antecedent and does propositional simplification using ground.

Direction LR is not implemented.
```

5.4 Modal Proof Commands

Command	*Function*
box-right	Remove consequent box
box-left	Remove antecedent box
diamond-left	Remove antecedent diamond
diamond-right	Remove consequent diamond
box-fby	Box followed by
box-lto	Box leads to
box-upto	Box upto

box-right: Remove consequent box

syntax: `(box-right &optional (fnumsr +) i)`

arguments:

```
<fnumsr> designates a boxed formula in the consequent of the form []A.
<i> is the context interval.
```

effect: `Removes box from formula <fnumsr>.`

```
Unboxed formulas of the antecedent are removed.
Undiamonded formulas of the consequent are removed.

Use box-fby, box-lto and box-upto to retain formulas of the
antecedent using these operators.
```

box-left: Remove antecedent box

syntax: `(box-left &optional (fnumsl -) i)`

arguments:

```
<fnumsl> designates a boxed formula in the antecedent of the form []A.
<i> is the context interval.
```

effect: `Removes box from formula <fnumsl>.`

diamond-left: Remove antecedent diamond

syntax: `(diamond-left &optional (fnumsl -) i)`

arguments:

```
<fnumsl> designates a diamonded formula in the antecedent of the form <>A.
<i> is the context interval.
```

effect: `Removes diamond from formula <fnumsl>.`

```
Unboxed formulas of the antecedent are removed.
Undiamonded formulas of the consequent are removed.

Use box-fby, box-lto and box-upto to retain formulas of the
antecedent using these operators.
```

diamond-right: Remove consequent diamond

syntax: `(diamond-right &optional (fnumsr +) i)`

arguments:

<fnumsr> designates a diamonded formula in the consequent of the form <>A.
<i> is the context interval.

effect: Removes diamond from formula <fnumsr>.

`box-fby`: Box followed by

syntax: (box-fby &optional (fnums *) (dir lr) i)

arguments:

<fnums> is a formula of the form A-->B or [](A-->B) depending of <dir>.
<dir> is the direction of rewriting.
<i> is the context interval.

effect: Rewrites <fnums> using the equivalence:

A-->B <=> [](A-->B)

If <dir> is LR boxes <fnums>.
If <dir> is RL unboxes <fnums>.

`box-lto`: Box leads to

syntax: (box-lto &optional (fnums *) (dir lr) i)

arguments:

<fnums> is a formula of the form A-:t:->B or [](A-:t:->B) depending of <dir>.
<dir> is the direction of rewriting.
<i> is the context interval.

effect: Rewrites <fnums> using the equivalence:

A-:t:->B <=> [](A-:t:->B)

If <dir> is LR boxes <fnums>.
If <dir> is RL unboxes <fnums>.

`box-upto`: Box upto

syntax: (box-upto &optional (fnums *) (dir lr) i)

arguments:

<fnums> is a formula of the form A-<=:t:->B or [](A-<=:t:->B) depending of <dir>.
<dir> is the direction of rewriting.
<i> is the context interval.

effect: Rewrites <fnums> using the equivalence:

A-<=:t:->B <=> [](A-<=:t:->B)

If <dir> is LR boxes <fnums>.
If <dir> is RL unboxes <fnums>.

5.5 Monotonicity

Command	*Function*
`mono`	Apply monotonicity of chop
`mono!`	Apply monotonicity of chop
`mono-chop`	Apply left monotonicity of chop
`mono-chop!`	Apply left monotonicity of chop
`chop-mono`	Apply right monotonicity of chop
`chop-mono!`	Apply right monotonicity of chop
`chop-mono-chop`	Apply center monotonicity of chop
`chop-mono-chop!`	Apply center monotonicity of chop

`mono`: Apply monotonicity of chop

syntax: `(mono &optional (fnumsl -) (fnumsr +) i)`

arguments:

`<fnumsl> designates a chopped formula A;B of the antecedent.`
`<fnumsr> designates a chopped formula C;D of the consequent.`
`<i> is the context interval.`

effect: `Applies monotonicity of chop leaving formulas [](A=>C) and [](B=>D) to be proved. The sequent is otherwise preserved.`

`mono!`: Apply monotonicity of chop

syntax: `(mono! &optional (fnumsl -) (fnumsr +) i)`

arguments:

`<fnumsl> designates a chopped formula A;B of the antecedent.`
`<fnumsr> designates a chopped formula C;D of the consequent.`
`<i> is the context interval.`

effect: `Applies monotonicity of chop leaving two subgoals to prove formulas A=>C and B=>D. Any other DC formulas of the sequent are removed.`

`mono-chop`: Apply left monotonicity of chop

syntax: `(mono-chop &optional (dcfmla "") (fnumsl -) cnum i)`

arguments:

`<dcfmla> is a DC formula F.`
`<fnumsl> designates a chopped formula A;B of the antecedent.`
`<i> is the context interval.`

effect: `Applies monotonicity of chop on the left formula A leaving 2 subgoals. One introducing F;B into the antecedent, the other proving [](A=>F). The DC formulas of the current goal is maintained in both subgoals.`

`mono-chop!`: Apply left monotonicity of chop

syntax: `(mono-chop! &optional (dcfmla "") (fnumsl -) cnum i)`

arguments:

`<dcfmla> is a DC formula F.`
`<fnumsl> designates a chopped formula A;B of the antecedent.`
`<i> is the context interval.`

effect: Applies monotonicity of chop on the left formula A leaving 2 subgoals.
One proving A=>F the other introducing F;B into the antecedent.
The DC formulas of the current goal is maintained only in the second subgoal.

chop-mono: Apply right monotonicity of chop

syntax: (chop-mono &optional (dcfmla "") (fnumsl -) cnum i)

arguments:
<dcfmla> is a DC formula F.
<fnumsl> designates a chopped formula A;B of the antecedent.
<i> is the context interval.

effect: Applies monotonicity of chop on the right formula B leaving 2 subgoals.
One introducing A;F into the antecedent, the other proving [](B=>F).
The DC formulas of the current goal is maintained in both subgoals.

chop-mono!: Apply right monotonicity of chop

syntax: (chop-mono! &optional (dcfmla "") (fnumsl -) cnum i)

arguments:
<dcfmla> is a DC formula F.
<fnumsl> designates a chopped formula A;B of the antecedent.
<i> is the context interval.

effect: Applies monotonicity of chop on the right formula B leaving 2 subgoals.
One proving B=>F the other introducing A;F into the antecedent.
The DC formulas of the current goal is maintained only in the second subgoal.

chop-mono-chop: Apply center monotonicity of chop

syntax: (chop-mono-chop &optional (dcfmla "") (fnumsl -) cnuml cnumr i)

arguments:
<dcfmla> is a DC formula F.
<fnumsl> designates a chopped formula A;B;C of the antecedent.
<i> is the context interval.

effect: Applies monotonicity of chop on the center interval B leaving 2 subgoals.
One introducing A;F;C into the antecedent, the other proving [](B=>F).
The DC formulas of the current goal is maintained in both subgoals.

This proof command was previously called mono-chop-mono.

chop-mono-chop!: Apply center monotonicity of chop

syntax: (chop-mono-chop! &optional (dcfmla "") (fnumsl -) cnuml cnumr i)

arguments:
<dcfmla> is a DC formula F.
<fnumsl> designates a chopped formula A;B;C of the antecedent.
<i> is the context interval.

effect: Applies monotonicity of chop on the center formula B leaving 2 subgoals.
One proving B=>F the other introducing A;F;C into the antecedent.
The DC formulas of the current goal is maintained only in the second subgoal.

This proof command was previously called mono-chop-mono!.

5.6 Quantifiers

Command	*Function*
exists-intro	Introduce left binding of duration
intro-exists	Introduce right binding of duration
intro-exists-intro	Introduce center binding of duration
len-intro	Introduce left binding of length
intro-len	Introduce right binding of length
intro-len-intro	Introduce center binding of length
exists-chop	Move left ∃ out of chop
chop-exists	Move right ∃ out of chop
chop-exists-chop	Move center ∃ out of chop

exists-intro: Introduce left binding of duration

syntax: (exists-intro state &optional (fnums *) cnum (dir lr))

arguments:

```
<state> is a boolean state expression.
<fnums> designates a chopped formula A;B or (SOME r: dur(state)=r /\ A);B
  depending on <dir>.
<cnum> designates a chop in the formula <fnums>.
<dir> is the direction of rewriting.
```

effect: If <dir> is LR introduces binding of duration of <state> in the left formula A producing (SOME r: dur(state) = r /\ A);B.
If <cnum> is given then initially this chop is lifted using top-assoc.

If <dir> is RL removes binding of duration of <state> in the left formula. Producing A;B.

intro-exists: Introduce right binding of duration

syntax: (intro-exists state &optional (fnums *) cnum (dir lr))

arguments:

```
<state> is a boolean state expression.
<fnums> designates a chopped formula A;B or A;(SOME r: dur(state)=r /\ B)
  depending on <dir>.
<cnum> designates a chop in the formula <fnums>.
<dir> is the direction of rewriting.
```

effect: If <dir> is LR introduces binding of duration of <state> in the right formula B producing A;(SOME r: dur(state) = r /\ B).
If <cnum> is given then initially this chop is lifted using top-assoc.

If <dir> is RL removes binding of duration of <state> in the right formula. Producing A;B.

intro-exists-intro: Introduce center binding of duration

syntax: (intro-exists-intro state &optional (fnums *) cnuml cnumr (dir lr))

arguments:

<state> is a boolean state expression.
<fnums> designates a chopped formula A;B;C or A;(SOME r: dur(state)=r /\ B);C
depending on <dir>.
<cnuml> designates a left chop in the formula <fnums>.
<cnumr> designates a right chop in the formula <fnums>.
<dir> is the direction of rewriting.

effect: If <dir> is LR introduces binding of duration of <state> in the center
formula B producing A;(SOME r: dur(state) = r /\ B);C.
If <cnuml> and <cnumr> are given then initially these chops are lifted using coneter-chop.

If <dir> is RL removes binding of duration of <state> in the center
formula. Producing A;B;C.

len-intro: Introduce left binding of length

syntax: (len-intro &optional (fnums *) cnum (dir lr))

arguments:

<fnums> designates a chopped formula A;B or (SOME r: len=r /\ A);B
depending on <dir>.
<cnum> designates a chop in the formula <fnums>.
<dir> is the direction of rewriting.

effect: If <dir> is LR introduces binding of length in the left
formula A producing (SOME r: len=r /\ A);B.
If <cnum> is given then initially this chop is lifted using top-assoc.

If <dir> is RL removes binding of length in the left formula. Producing A;B.

intro-len: Introduce right binding of length

syntax: (intro-len &optional (fnums *) cnum (dir lr))

arguments:

<fnums> designates a chopped formula A;B or A;(SOME r: len=r /\ B)
depending on <dir>.
<cnum> designates a chop in the formula <fnums>.
<dir> is the direction of rewriting.

effect: If <dir> is LR introduces binding of length in the right formula B
producing A;(SOME r: len=r /\ B).
If <cnum> is given then initially this chop is lifted using top-assoc.

If <dir> is RL removes binding of length in the right formula. Producing A;B.

intro-len-intro: Introduce center binding of length

syntax: (intro-len-intro &optional (fnums *) cnuml cnumr (dir lr))

arguments:
<fnums> designates a chopped formula A;B;C or A;(SOME r: len=r /\ B);C depending on <dir>.
<cnuml> designates a left chop in the formula <fnums>.
<cnumr> designates a right chop in the formula <fnums>.
<dir> is the direction of rewriting.

effect: If <dir> is LR introduces binding of length in the center formula B producing A;(SOME r: len=r /\ B);C.
If <cnuml> and <cnumr> are given then initially these chops are lifted using center-assoc.

If <dir> is RL removes binding of length in the center formula.
Producing A;B;C.

`exists-chop`: Move left ∃ out of chop

syntax: (exists-chop &optional (fnums *) (dir lr) type)

arguments:
<fnums> is a chopped formula for the form (SOME x:A);B or (SOME x: A;B) depending of <dir>.
<dir> is the direction of rewriting.
<type> is the type of x.

effect: Moves existential quantifier in or out of left hand side of chopped formula <fnums>.
If <dir> is LR the quantifier is moved out producing (SOME x: A;B).
If <dir> is RL the quantifier is moved in producing (SOME x: A);B.
It may be necessary to supply the type of x in order for PVS to instantiate the rewrite rule correctly or otherwise no rewriting will occur.

`chop-exists`: Move right ∃ out of chop

syntax: (chop-exists &optional (fnums *) (dir lr) type)

arguments:
<fnums> is a chopped formula for the form A;(SOME x:B) or (SOME x: A;B) depending of <dir>.
<dir> is the direction of rewriting.
<type> is the type of x.

effect: Moves existential quantifier in or out of right hand side of chopped formula <fnums>.
If <dir> is LR the quantifier is moved out producing (SOME x: A;B).
If <dir> is RL the quantifier is moved in producing A;(SOME x: B).
It may be necessary to supply the type of x in order for PVS to instantiate the rewrite rule correctly or otherwise no rewriting will occur.

`chop-exists-chop`: Move center ∃ out of chop

syntax: (chop-exists-chop &optional (fnums *) (dir lr) type)

arguments:

<fnums> is a chopped formula for the form `A;(SOME x:B);C` or `(SOME x: A;B;C)` depending of <dir>.
<dir> is the direction of rewriting.
<type> is the type of x.

effect: Moves existential quantifier in or out of center formula of chopped formula <fnums>.
If <dir> is LR the quantifier is moved out producing `(SOME x: A;B;C)`.
If <dir> is RL the quantifier is moved in producing `A;(SOME x: B);C`.
It may be necessary to supply the type of x in order for PVS to instantiate the rewrite rule correctly or otherwise no rewriting will occur.

5.7 Induction

Command	*Function*
`bool-induct-chop`	Left induct on bool state
`bool-chop-induct`	Right induct on bool state
`induct-chop`	Left induct
`chop-induct`	Right induct

`bool-induct-chop`: Left induct on bool state

syntax: `(bool-induct-chop r state &optional i)`

arguments:

<R> is a DC formula schema of type [IForm -> IForm].
<state> is a boolean state expression.
<i> is the context interval.

effect: Uses left DC induction of boolean states to produce three subgoal.
One introducing `[]R(TRUE)` in to the antecedent.
The other to prove `[]R(point)`. The third to prove
`[]([]H(X) => R(X \/ 'state';X \/ 'NOT state';X))` for all DC formulas X.

`bool-chop-induct`: Right induct on bool state

syntax: `(bool-chop-induct r state &optional i)`

arguments:

<R> is a DC formula schema of type [IForm -> IForm].
<state> is a boolean state expression.
<i> is the context interval.

effect: Uses right DC induction of boolean states to produce three subgoal.
One introducing `[]R(TRUE)` in to the antecedent.
The other to prove `[]R(point)`. The third to prove
`[]([]H(X) => R(X \/ X;'state' \/ X;'NOT state'))` for all DC formulas X.

`induct-chop`: Left induct

syntax: `(induct-chop r state &optional i)`

arguments:

<R> is a DC formula schema of type [IForm -> IForm].
<state> is a state expression.
<i> is the context interval.

effect: Uses left DC induction on states to produce three subgoal.
One introducing []R(TRUE) in to the antecedent.
The other to prove []R(point). The third to prove
[]([]H(X) => R(X \/ (SOME x: ’state=m(x)’;X))) for all DC formulas X.

chop-induct: Right induct

syntax: (chop-induct r state &optional i)

arguments:

<R> is a DC formula schema of type [IForm -> IForm].
<state> is a state expression.
<i> is the context interval.

effect: Uses right DC induction on states to produce three subgoal.
One introducing []R(TRUE) in to the antecedent.
The other to prove []R(point). The third to prove
[]([]H(X) => R(X \/ (SOME x: X;’state=m(x)’))) for all DC formulas X.

5.8 Context Free Formulas

Command	*Function*
free-chop	Propagate left context free expression through chop
chop-free	Propagate right context free expression through chop
chop-free-chop	Propagate center context free expression through chop

free-chop: Propagate left context free expression through chop

syntax: (free-chop &optional (fnums *) (dir lr))

arguments:

<fnums> is a formula of the form CF /\ A;B or CF /\ (A;B) depending of <dir>.
<dir> is the direction of rewriting.

effect: If CF is a context free formula propagates CF in or out of chopped formula
using rewriting on the equivalence:
CF /\ A;B <=> CF /\ (A;B)
<dir> is the direction of rewriting.

chop-free: Propagate right context free expression through chop

syntax: (chop-free &optional (fnums *) (dir lr))

arguments:

<fnums> is a formula of the form A;B /\ CF or CF /\ (A;B) depending of <dir>.
<dir> is the direction of rewriting.

effect: If CF is a context free formula propagates CF in or out of chopped formula
using rewriting on the equivalence:
A;B /\ CF <=> CF /\ (A;B)
<dir> is the direction of rewriting.

chop-free-chop: Propagate center context free expression through chop

syntax: (chop-free-chop &optional (fnumsl *) (dir lr))

arguments:
<fnums> is a formula of the form A;(B /\ CF);C or CF /\ (A;B;C) depending of <dir>.
<dir> is the direction of rewriting.

effect: If CF is a context free formula propagates CF in or out of chopped formula
using rewriting on the equivalence:
A;(B /\ CF);C <=> CF /\ (A;B;C)
<dir> is the direction of rewriting.

5.9 Duration Properties

Command	*Function*
axiom4	Apply axiom 4
dur-preserve	Apply duration preservation over chop
assertion	Assert equality on durations
p-monotone	Assert state predicate implication

axiom4: Apply axiom 4

syntax: (axiom4 &optional (fnums *) (dir lr) i)

arguments:
<fnums> designates a formula of the form (dur(p)=n(r));(dur(p)=n(s)) or
dur(p)=n(r+s) depending of <dir>.
<dir> is the direction of rewriting.
<i> is the context interval.

effect: Rewrites the formula <fnums> using axiom 4 of DC:
(dur(p)=n(r));(dur(p)=n(s)) <=> dur(p)=n(r+s)

dur-preserve: Apply duration preservation over chop

syntax: (dur-preserve &optional (fnumsl -) i)

arguments:
<fnumsl> designates a formula in the antecedent of the form
(dur(p)*dur(q));(dur(p)*dur(q)).
<i> is the context interval.

effect: Applies duration preservation of predicate of durations over chop.
If * is an additive predicate (like =, >, >= etc.) the formula
dur(p)*dur(q) in the antecedent. <fnumsl> is removed.

assertion: Assert equality on durations

syntax: (assertion statep stateq i)

arguments:
<statep> is a boolean state expression.
<stateq> is a boolean state expression.
<i> is the context interval.

effect: Uses assertion on <statep> and <stateq>. Produces two subgoals.
One with dur(statep)=dur(stateq) added to the antecedent.
The other proving (statep<=>stateq)(t) for all times t.

p-monotone: Assert state predicate implication

syntax: (p-monotone &optional (fnumsl -) (fnumsr +) i)

arguments:
<fnumsl> designates a formula in the antecedent of the form 'p'.
<fnumsl> designates a formula in the consequent of the form 'q'.
<i> is the context interval.

effect: Applies monotonicity of state predicates to derive 'p'=>'q' from
proving (p=>q)(t) for all times t.
All DC formulas of the sequent are removed.

Bibliography

[1] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1992.

[2] Michael R. Hansen and Zhou Chaochen. Duration calculus, logical foundations. *Formal Aspects of Computing*, 9:283–303, 1997.

[3] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System*. Computer Science Laboratory, SRI International, Menlo Park CA 94025, USA, March 1993.

[4] Jens Ulrik Skakkebæk. A verification assistant for a real-time logic. Technical Report 150, Dept. of Computer Science, Technical University of Denmark, 1994. Ph.D. thesis.