A User-Guide for a PVS-Supported Temporal Verification System

Tamarah Arons
tamarah@wisdom.weizmann.ac.il

July 4, 2002

Contents

Chapter 1

Introduction

In this document we introduce our PVS (version 2.4.1) implementation of a linear temporal logic framework. This framework includes both a set of theories defining a temporal logic, and a number of strategies which aid in conducting the proofs. We use the atomic bakery algorithm as an example.

We assume that the user has a basic knowledge of PVS and will not be explaining the specification language or prover commands of PVS. An excellent tutorial, *A Tutorial Introduction to PVS* [COR+95][1] is highly recommended for anyone lacking experience with PVS. Alternatively, an extremely brief introduction to PVS is provided in Appendix A.

1.1 Setting Up the Framework

Accompanying this user guide are two PVS dump files and a file pvs-strategies of strategies. The first dump file, TemporalLogicPrelude.dump, is a dump of all the temporal logic theories which have been created. The second, BakeryExample.dump is a dump of the bakery example referred to in this guide.

To set up the framework proceed as follows:

1. Create directories for the temporal logic theories, and for the bakery example (it is preferable that they be separate). For this explanation we assume that the directories are called Prelude and Bakery.

2. Copy the two dump files to their respective directories. Copy the pvs-strategies file to the Prelude directory.

3. Go to the Prelude directory and open PVS.[2] (You will be prompted as to whether to create a new context - answer yes).

4. Undump the dump file TemporalLogicPrelude.dump (`M-x undump-pvs-files`) into this directory and open the file temporal_logic_prelude.pvs. (This file is not, itself, part

[1] This tutorial, along with the dump files of its examples can be found at ~verify/PVS2-4.1/Doc. The various user guides of PVS are also stored here.

[2] PVS executables can currently be accessed as ~verify/PVS2-4.1/pvs for Sun machines, ~verify/PVS2-4.1/pvs_linux for Linux machines.

of the temporal theories, its purpose is to include all the files which are in the prelude so that they can all be placed in the context conveniently.)

5. Typecheck the file (`M-x tc`). Prove all the theories in the directory using the (`M-x pri`) command. This takes a couple of minutes.

6. Change the context (`M-x change-context`) to directory Bakery. Undump BakeryExample.dump.

7. This example is dependent on the files in the temporal logic prelude. To add these files to the context, use the command `M-x load-prelude-library` and give Prelude as the directory.

8. Open the file bakery_example.pvs and typecheck it.

Whenever one wants to access the temporal logic theories from a new directory it is necessary to load them as a prelude library. However, the files in the directory do not have to be reproved. It is also necessary to copy the pvs-strategies file to the new directory (unless it has been copied to the home directory).

1.2 A List of Theories

We give a list of the theories included in the dump files, and a brief explanation of each file.

The Temporal Logic Prelude

Theories defining the basic temporal logic system	
tempoperators	Definitions of state sequences and temporal operators.
PFS	Definition of finite discrete systems and some invariance rules.
Theories defining rules for proving response properties	
CHAIN	The chain rule.
WELL	Rule WELL.
WELL_simple	A non-parameterized version of the WELL rule.
DIST_RANK	The distributed rank rule. Derived from WELL.
DIST_RANK_simple	A non-parameterized version of the DIST-RANK rule.
DIST_RANK_infinite	A version of DIST_RANK for infinite systems.
PREORDER	Combines distributed ranks with the use of a pre-order.
Miscellaneous	
TRANS_DOMAIN	Defines the often used transition domain TRANS_DOMAIN : *(location, processor id)*, and proves that it is finite.
remove_impl	Lemmas converting implications and iff statements to if-then-else structures. See strategies `remove-impl` and `remove-iff` in Section 5.3.
more_nat_types	Defines often used subsets of positive natural numbers.
comparator_defs	Defines properties of comparators, including well-foundedness.
min_max	Defines the element of a set which is minimum / maximum with respect to an evaluation function.
finite_domain	Lemmas for deducing the finiteness of a domain.
nat_well_founded	The natural numbers are well-founded.
set_well_founded	The strict subset relation is well-founded over sets of natural numbers.
two_well_founded	The composition of two well-founded domains is well-founded.
repeated_well_founded	The repeated concatenation of a well-founded domain is well-founded.
equivalent_well_founded	Gives conditions under which the well-foundedness of one operator can be deduced from that of another.

The Bakery Example

bakery_definition	Definition of the bakery system
bakery_safety	Proof that bakery implements mutual exclusion
Proofs of accessibility of the critical region	
bakery_dist_rank	Proof using rule DIST_RANK
bakery_well	Proof using rule WELL
bakery_preorder	Proof using rule PREORDER

Chapter 2

Temporal Logic Theories

In this Chapter we present our PVS-model of a linear temporal logic and a fair discrete system. Our system is based on the LTL definitions given in [MP95].

2.1 State Sequences, Temporal Operators and Assertions

Let S be a system containing a set V of discrete variables.

A *state* s of S is a type-consistent interpretation of the system variables V.

A *state sequence* is an infinite sequence of states, represented in our model as a mapping from time (the natural numbers) to states:

```
STATE_SEQ: TYPE = [TIME → STATE]
```

Assertions are properties that are defined on individual states, without reference to their position in the state sequence, i.e.

```
ASSERTION: TYPE = [STATE → bool]
```

Disjunction, conjunction, negation and implication over assertions are defined using the operators of Fig. 2.1.

A *temporal property*, `TP`, is a property that is interpreted over the state sequence.

```
TP: TYPE = [STATE_SEQ, TIME → boolean]
```

Temporal properties are composed of assertions connected together using boolean connectives or the temporal operators. In this theory we have defined the temporal operators henceforth ($\Box$), eventually ($\Diamond$), next ($\bigcirc$) and until ($\mathcal{U}$). The temporal operators defined in this theory are listed in Fig. 2.1.

Most of the operations have more than one associated operator, and the user may use whichever he or she prefers. All of these operators, excluding `G`, `F`, `X`, `U` and `[]` are overloadings of pre-defined PVS operators. Overloaded operators are case insensitive, and overloaded binary operators can be used in an infix manner (e.g. `a implies b`). In contrast, `G`, `F`, `X` and `U` are case sensitive and `U` is a prefix operator only (i.e. the user must specify `U(a, b)` and not `a U b` or `u(a, b)`).

Operation	Operators
Disjunction	`OR`, \/
Conjunction	`AND`, /\, &
Negation	`NOT`, ~
Implication	`IMPLIES`
Henceforth ($\Box$)	`G`, `[]`
Eventually ($\Diamond$)	`F`, `<>`
Next ($\bigcirc$)	`X`, `O`
Until ($\mathcal{U}$)	`U`

Figure 2.1: Boolean connectives and temporal operators

However, when using the overloaded infix operators one must beware that sometimes PVS does not use the precedence rules we would expect in temporal logic. In general, using brackets, or the prefix notation, overcomes this problem. For example, the lemma

```
Conjunction_distributive: LEMMA G(a AND b) = G(a) AND G(b)
```

does not typecheck, but the following two versions do:

```
Conjunction_distributive2: LEMMA G(a AND b) = AND(G(a), G(b))

Conjunction_distributive3: LEMMA G(a AND b) = (G(a) AND G(b))
```

2.2 Parameterized Fair Systems

A *parameterized fair system* (PFS) $S = \langle V, \Theta, \rho, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$ consists of

- V : A non-empty set of typed *system variables*. A state s of the system S provides a type-consistent interpretation of the system variables V. For a state s and a system variable $v \in V$, we denote by $s.v$ the value assigned to v by the state s. Let Σ denote the set of states over V.

- Θ : The *initial condition*. An assertion characterizing the initial states.

- $\rho(V, V')$: The *transition relation*. A formula relating the values V of the variables in state $s \in \Sigma$ to the values V' in an S-successor state $s' \in \Sigma$.

- $\mathcal{T}$: The non-empty *transition domain*. This is a domain from which we will map justice and compassion conditions.

- $\mathcal{J}$: The *justice conditions*. This is a mapping from $\mathcal{T}$ to assertions ($\mathcal{J}$: [$\mathcal{T} \mapsto$ assertion]). It characterizes the just states. For every $t \in \mathcal{T}$, a computation must contain infinitely many $\mathcal{J}[t]$-states.

- $\mathcal{C}$: The *compassion conditions*. It is a mapping from $\mathcal{T}$ to pairs of assertions : $[\mathcal{T} \mapsto \langle p, q\rangle])$ where p and q are assertions. If $\mathcal{C}[t] = \langle p, q\rangle$, we will refer to p and q as $\mathcal{C}[t].p$ and $\mathcal{C}[t].q$, respectively. Each such pair $\langle p, q\rangle$ characterizes sets of enabled and taken states. For every $t \in \mathcal{T}$, a computation containing infinitely many $\mathcal{C}[t].p$-states must also contain infinitely many $\mathcal{C}[t].q$-states.

A PFS typically has an associated *parameterized domain*, $\mathcal{P}$. This is the domain with respect to which the system is parameterized. It may be finite, e.g. $[1 \ldots N]$, or infinite, e.g. the set $\mathbb{N}$ of natural numbers.

Generally, $\mathcal{P}$ is the set of process identifiers in the system.

We formulate a PFS in the PVS specification language as follows :

```
PFS: TYPE =
  [# initial: ASSERTION,
     rho: BI_ASSERTION,
     justice: JUSTICE_TYPE,
     compassion: COMPASSION_TYPE #]
```

where

```
BI_ASSERTION: TYPE = [STATE, STATE → boolean]

JUSTICE_TYPE: TYPE = [TRANSITION_DOMAIN → ASSERTION]

COMPASSION_PAIR: TYPE = [# p: ASSERTION, q: ASSERTION #]

COMPASSION_TYPE: TYPE = [TRANSITION_DOMAIN → COMPASSION_PAIR]
```

We note that this differs from the definition given above in that it does not have a system variables component. When importing the PFS theory, the user must instantiate it with `STATE` and `TRANSITION_DOMAIN` parameters. Intuitively, the system variables are the structures found in the state definition.

Let S be a PFS for which the above components have been identified.

The transition relation $\rho(V, V')$ identifies state s' as an *S-successor* of s if $\langle s, s'\rangle \models \rho(V, V')$, where $\langle s, s'\rangle$ is the joint interpretation which interprets $x \in V$ as $s[x]$ and x' as $s'[x]$.

We define a *run* of $\mathcal{D}$ to be an infinite sequence of states $\sigma : s_0, s_1, s_2, \ldots$ satisfying the requirement of

- *Consecution*: For each $j = 0, 1, \ldots$, the state s_{j+1} is an S-successor of the state s_j.

A run $\sigma : s_0, s_1, s_2, \ldots$ of S is *initialized* if it satisfies

- *Initiality*: s_0 is initial, i.e. $s_0 \models \theta$.

A run σ of S is *fair* if it satisfies the following requirements:

- *Justice*: For every $J \in \mathcal{J}$ there are infinitely many J-states in σ.

- *Compassion*: For every $\langle p, q\rangle \in \mathcal{C}$, if σ contains infinitely many p-states then it must also contain infinitely many q-states.

An initialized fair run is called a *computation*.

In the PVS specification language we define a `run` , `initialized_run` and `consecution` as expected. We then define the `INITIALIZED_RUN` and `COMPUTATION` types as follows:

```
just(seq, pfs): bool = ∀ f: ∀ t: ∃ k: k > t ∧ pfs`justice(f)(seq(k))

compassionate(seq, pfs): bool =
    ∀ (f: TRANSITION_DOMAIN):
      (∀ t: (∃ j: j ≥ t ∧ p(pfs`compassion(f))(seq(j)))) ⊃
       (∀ k: ∃ l: l ≥ k ∧ q(pfs`compassion(f))(seq(l)))

computation(seq, pfs): bool =
    pfs`initial(seq(0)) ∧
     consecution(seq, pfs) ∧ just(seq, pfs) ∧ compassionate(seq, pfs)

INITIALIZED_RUN(pfs): TYPE = {seq | initialized_run(seq, pfs)}

COMPUTATION(pfs): TYPE = {seq | computation(seq, pfs)}
```

A temporal property *tp* is said to be *valid* if *tp* holds in the first state of every model, i.e. $\models tp$.

Property *tp* is *P-valid* if it holds in the first state of every computation of program P, i.e. $P \models tp$. It is *P-reachable valid* if it holds in the first state of every initialized run of program P.

An assertion p is *P-state valid* if it holds in every state of every computation of program P.

In our system we define:

```
is_valid(tp : TP): bool = ∀ seq: tp(seq, 0)

is_P_valid(tp, pfs): bool = ∀ seq: computation(seq, pfs) ⊃ tp(seq, 0)

is_P_reachable_valid(tp, pfs): bool =
    ∀ seq: initialized_run(seq, pfs) ⊃ tp(seq, 0)

is_P_state_valid(a: ASSERTION, pfs): bool =
    ∀ seq, t: computation(seq, pfs) ⊃ a(seq(t))
```

We note that P-reachable validity is a stronger property than P-validity. Similarly, all computations are initialized runs. Formally, we prove that :

```
P_reachable_valid_implies_P_valid: LEMMA
  is_P_reachable_valid(tp, pfs) ⊃ is_P_valid(tp, pfs)

computation_is_run: LEMMA
  ∀ seq: computation(seq, pfs) ⊃ initialized_run(seq, pfs)
```

These lemmas can be invoked in proofs in order to convert to the preferred notation.

We note that there is automatic conversion from `ASSERTION` to `TP` , so it is syntactically correct to write `is_P_valid(a, pfs)` for assertion a.

2.3 Defining the BAKERY Algorithm

In this section we define a parameterized fair system (PFS) for the BAKERY algorithm presented in Fig. 2.2.

$$
\begin{array}{lll}
\textbf{in} & N & : \ \textbf{integer where}\, N > 1 \\
\textbf{local} & y & : \ \textbf{array}\,[1..N]\,\textbf{of natural where}\, y = 0 \\
\multicolumn{2}{l}{\overset{N}{\underset{p=1}{\Big\|}}\ P[p] ::} & \left[\begin{array}{l}
\textbf{loop forever do} \\
\quad \left[\begin{array}{ll}
l_0: & \textbf{NonCritical} \\
l_1: & y[p] := \textbf{choose}\, m\, \textbf{such that}\, \forall q : (m > y[q]) \\
l_2: & \textbf{await}\, \forall q : (y[q] = 0 \vee y[p] < y[q]) \\
l_3: & \textbf{Critical} \\
l_4: & y[p] := 0
\end{array}\right]
\end{array}\right]
\end{array}
$$

Figure 2.2: Parameterized mutual exclusion algorithm BAKERY

The system state defines the current location and y-values for each of the processors. In Fig. 2.3 we define BAKERY in the PVS specification language.

We noted before that the transition domain often is comprised of a location and processor identifier field. We have therefore defined a `TRANS_DOMAIN` type, which defines a transition domain with these types. Importing `TRANS_DOMAIN[`*progSize*`,` N`]` creates and imports the following definitions:

```
LOCATION: TYPE = upto[progSize − 1]

PROC_ID: TYPE = upto_nz[N]

TRANS_DOMAIN: TYPE = [# loc: LOCATION, pid: PROC_ID #]
```

In addition, it provides lemmas proving that these three types are finite – these lemmas are useful in liveness proofs.

The user is not, however, obliged to use the `TRANS_DOMAIN` theory to define a transition domain and may define a transition domain manually.

The intuition for the justice set is that a just transition which is continually enabled should be eventually taken. For a just transition at line l_i of processor p, we create a justice requirement stating that processor p must infinitely often be in a state in which the transition at line l_i is not enabled.

Justice requirements for BAKERY are given in Fig. 2.4. There is no justice requirement associated with location l_0 as there is no restriction that any processor leave the non-critical section. We note that the justice requirements associated with locations 3 and 4 simply assert that the processor p is not at location l_3 or l_4, respectively. In contrast, the requirement associated with location 2 asserts that either the processor is not at l_2 or it cannot progress as it must "wait" for its y-value to become minimal.

The condition associated with location 1 asserts that either the processor p is not at location l_1, or there is no natural number m greater than all the current y-values. The

```
bakery_definition[N: posnat]: THEORY
 BEGIN

  ASSUMING N_greater_1: ASSUMPTION N > 1 ENDASSUMING

  IMPORTING TRANS_DOMAIN[5, N]

  STATE: TYPE = [# y: [PROC_ID → nat], loc: [PROC_ID → LOCATION] #]

  IMPORTING PFS[STATE, TRANS_DOMAIN]

  p, q: VAR PROC_ID

  rho: BI_ASSERTION =
      (λ (current, next: STATE):
         next = current ∨
         (∃ p:
              loc(current)(p) = 0 ∧
                y(next) = y(current) ∧ loc(next) = loc(current) WITH [(p) := 1]
            ∨ loc(current)(p) = 1 ∧
               (∃ (m: nat): (∀ q: y(current)(q) < m) ∧
                                  y(next) = y(current) WITH [(p) := m])
                ∧ loc(next) = loc(current) WITH [(p) := 2]
            ∨ loc(current)(p) = 2 ∧
                (∀ q: q ≠ p → y(current)(q) = 0 ∨ y(current)(p) ≤ y(current)(q))
                ∧ y(next) = y(current)
                ∧ loc(next) = loc(current) WITH [(p) := 3]
            ∨ loc(current)(p) = 3
                ∧ y(next) = y(current)
                ∧ loc(next) = loc(current) WITH [(p) := 4]
            ∨ loc(current)(p) = 4
                ∧ y(next) = y(current) WITH [(p) := 0]
                ∧ loc(next) = loc(current) WITH [(p) := 0]))
 END bakery_definition
```

Figure 2.3: PVS formulation of BAKERY algorithm

second conditions is always false. The justice conditions must include all conditions in which the transition cannot be taken, and syntactically (if not semantically) the case of there being no satisfying m is one of them. The requirement that there be such an m is no different from the requirement that all non-zero y-values be smaller than that of p, in j_2; it is only our knowledge of the properties of natural numbers that assures us that this is a trivial requirement. However, PVS needs to have such properties formally proved to it and so it is necessary to state this condition.

The compassion conditions are left empty (using an empty compassion definition defined in the PFS theory).

The completed PFS is presented in Fig. 2.4.

```
st: VAR STATE

t: VAR TRANS_DOMAIN

justice: [TRANS_DOMAIN → ASSERTION] =
    (λ t:
       (λ st:
          IF loc(t) = 1
            THEN loc(st)(pid(t)) ≠ 1 ∨ ¬ (∃ (m: nat): ∀ p: y(st)(p) < m)
          ELSIF loc(t) = 2
                THEN loc(st)(pid(t)) ≠ 2 ∨
                      ¬ (∀ q: q ≠ pid(t) ⊃ y(st)(q) = 0 ∨ y(st)(pid(t)) ≤ y(st)(q))
          ELSIF loc(t) = 3 ∨ loc(t) = 4 THEN loc(st)(pid(t)) ≠ loc(t)
          ELSE TRUE
          ENDIF))

pfs: PFS =
    (# initial := {st | ∀ p: y(st)(p) = 0 ∧ loc(st)(p) = 0},
       rho := rho,
       justice := justice,
       compassion := empty_compassion #)
```

Figure 2.4: A PFS for the BAKERY algorithm

Chapter 3

Proving Safety Properties

In this Chapter we examine methods for proving safety properties, that is properties of the form `is_P_valid` $(\mathtt{G}(a))(P \models \Box a$ in the syntax of [MP95]), or `is_P_reachable_valid` $(\mathtt{G}(a))$, for any assertion a.

We have implemented a number of invariance rules to aid in this verification. The rules have associated strategies which bring in the lemma, and, where possible, perform some instantiation and simplification. Most of the rules are taken from [MP95].

- `BINV` : This is the basic invariance rule.
  It can be invoked using strategy `binv` with optional parameters $inv1, \ldots, inv4$. Parameters $inv1, \ldots, inv4$ are the names of structures to be expanded.

- `INV` : The general invariance rule.
  Invoked using strategy `inv` with parameters *weaker* and *stronger* where *weaker* is the property whose invariance is to be proved, and *stronger* is an inductive property implying *weaker*. (In Fig. 3.1, *weaker* corresponds to a, *stronger* to b.)

- `MONI` : Monotonicity of invariants rule.
  Invoked using strategy `moni`.

- `CONI` : Conjunction of invariance rule.
  The `CONI` rule is invoked using the strategy `coni` with required parameters $subst1$, $subst2$ indicating the line numbers of the formulas whose conjunction is to be generated. If the optional flag : *hide*? is set to t, then the formulas are hidden and only the conjunction is shown. By default the formulas are not hidden.

- `SV_PSV` : Conversion from state validities to P-state validities

- `state_valid_to_valid` and `valid_to_state_valid` : Conversion of state-valid formulas to valid formulas and vice versa.
  The strategies `state-valid-to-valid` and `valid-to-state-valid` invoke these two rules, respectively. The first optional parameter, $fnum$, (default -) is the line number of the formula to be converted. If the second parameter : *hide*? (default f) is set to t the formula at line $fnum$ is hidden provided that it was given as a negative integer.

```
BINV: LEMMA
  ∀ fds: ∀ (a: ASSERTION):
    (∀ seq:
      initialized_run(seq, fds) →
        (a(seq(0)) ∧
          (∀ t: fds‘ρ(seq(t), seq(t+1)) ∧ a(seq(t)) → a(seq(t+1)))))
     → is_P_reachable_valid(G(a), fds)

INV: LEMMA
  ∀ fds: ∀ (a, b: ASSERTION):
    (∀ seq: initialized_run
      (seq, fds) →
        ((∀ t: b(seq(t)) → a(seq(t))) ∧
          b(seq(0)) ∧
           (∀ t: fds‘ρ(seq(t), seq(t+1)) ∧ b(seq(t)) → b(seq(t+1)))))
     → is_P_reachable_valid(G(a), fds)

MONI: LEMMA
  is_P_state_valid((a → b), fds) ∧ is_P_valid(G(a), fds) → is_P_valid(G(b), fds)

CONI: LEMMA
  is_P_valid(G(a), fds) & is_P_valid(G(b), fds) → is_P_valid(G(a & b), fds)

SV_PSV: LEMMA
  is_P_valid(G(a), fds) ∧ is_P_state_valid((a & b) → c, fds) →
   is_P_state_valid(b → c, fds)

valid_to_state_valid: LEMMA
  is_P_valid(G(a), fds) → is_P_state_valid(a, fds)

state_valid_to_valid: LEMMA
  is_P_state_valid(a, fds) → is_P_valid(G(a), fds)
```

Figure 3.1: Invariance rules

We note that rules `BINV` and `INV` prove that the property is P-reachable valid, and not just P-valid. It is easy to show that a property that is P-reachable valid is P-valid (one can invoke the lemma `P_reachable_valid_implies_P_valid`).

3.1 Proving that the BAKERY Algorithm Implements Mutual Exclusion

We return to the BAKERY algorithm of Figures 2.2 and 2.3 and show how system invariants can be proved using the rules the previous section. The reader is invited to open the file `bakery_safety.pvs` in PVS and to work through the proofs with us. It is possible to step through the proofs by using `M-x step-proof`, or to view them using `M-x edit-proof`.

It is easy to see that a processor p has y-value 0 exactly when it is at locations 0 and 1:

```
yZero: ASSERTION =
    λ st:
      LET y = y(st), loc = loc(st) IN
        ∀ (i: PROC_ID): (y(i) = 0 IFF (loc(i) = 0 ∨ loc(i) = 1))

yZero: LEMMA is_P_reachable_valid(G(yZero), fds)
```

The proof script below shows how one might prove this theory. First we apply strategy `binv`. PVS automatically proves the base case of this inductive argument, and returns to us with the induction step (showing that property *yZero* is invariant over the transition relation *rho*). Note that the strategy also did some labeling – in general referring to lines by their labels is far more robust than using the line numbers. As a convention, the current state, $seq(t)$, is referred to as `current!1`, and its successor state, $seq(t+1)$, as `next!1`. Formulas stating these equivalences are labeled *current* and *next* and are hidden. (They can be revealed using the `reveal` command.)

We expand *yZero*. Thereafter we apply the strategy `split-rho` which splits the transition relation rho, returning with five subgoals (the case in which the system idles, `next!1 = current!1`, is solved automatically).

We tackle the first of the remaining subgoals. We first call strategy `split-all` which simplifies and case splits, generating three subgoals. We call `inst?` to instantiate the first one, and then `simp` (which calls the inbuilt PVS simplify, assert and flatten commands) to simplify and complete the proof. The same can be done on the second and third branch generated by `split-all`.

```
yZero :

  |-------
{1}   is_P_valid(G(assertion_to_TP(yZero)), fds)

Rule? (binv)
yZero rewrites yZero(current!1)
  to FORALL (i: PROC_ID):
        (y(current!1)(i) = 0 IFF
          (loc(current!1)(i) = 0 OR loc(current!1)(i) = 1))
lemma BINV,
this simplifies to:
yZero :
;;;The inductive step of the BINV rule

{-1,(rho)}
      loc(current!1)(p!1) = 0 AND
       y(next!1) = y(current!1) AND
        loc(next!1) = loc(current!1) WITH [(p!1) := 1]
       OR
       loc(current!1)(p!1) = 1 AND
        (EXISTS (m: nat):
           (FORALL (q: PROC_ID[N]): y(current!1)(q) < m) AND
            y(next!1) = y(current!1) WITH [(p!1) := m])
         AND loc(next!1) = loc(current!1) WITH [(p!1) := 2]
        OR
```

```
        loc(current!1)(p!1) = 2 AND
         (FORALL (q: PROC_ID[N]):
            q /= p!1 IMPLIES
             y(current!1)(q) = 0 OR y(current!1)(p!1) <= y(current!1)(q))
          AND
          y(next!1) = y(current!1) AND
           loc(next!1) = loc(current!1) WITH [(p!1) := 3]
         OR
         loc(current!1)(p!1) = 3 AND
          y(next!1) = y(current!1) AND
           loc(next!1) = loc(current!1) WITH [(p!1) := 4]
          OR
          loc(current!1)(p!1) = 4 AND
           y(next!1) = y(current!1) WITH [(p!1) := 0] AND
            loc(next!1) = loc(current!1) WITH [(p!1) := 0]
{-2,(invariant)}
      yZero(current!1)
  |-------
{1,(rtp)}
      yZero(next!1)

Rule? (expand "yZero")
Expanding the definition of yZero,
this simplifies to:
yZero :
;;;The inductive step of the BINV rule

[-1,(rho)]
      loc(current!1)(p!1) = 0 AND
       y(next!1) = y(current!1) AND
        loc(next!1) = loc(current!1) WITH [(p!1) := 1]
       OR
       loc(current!1)(p!1) = 1 AND
        (EXISTS (m: nat):
           (FORALL (q: PROC_ID[N]): y(current!1)(q) < m) AND
            y(next!1) = y(current!1) WITH [(p!1) := m])
         AND loc(next!1) = loc(current!1) WITH [(p!1) := 2]
        OR
        loc(current!1)(p!1) = 2 AND
         (FORALL (q: PROC_ID[N]):
            q /= p!1 IMPLIES
             y(current!1)(q) = 0 OR y(current!1)(p!1) <= y(current!1)(q))
          AND
          y(next!1) = y(current!1) AND
           loc(next!1) = loc(current!1) WITH [(p!1) := 3]
         OR
         loc(current!1)(p!1) = 3 AND
          y(next!1) = y(current!1) AND
           loc(next!1) = loc(current!1) WITH [(p!1) := 4]
          OR
          loc(current!1)(p!1) = 4 AND
           y(next!1) = y(current!1) WITH [(p!1) := 0] AND
            loc(next!1) = loc(current!1) WITH [(p!1) := 0]
```

```
{-2,(invariant)}
      FORALL (i: PROC_ID):
        (y(current!1)(i) = 0 IFF
          (loc(current!1)(i) = 0 OR loc(current!1)(i) = 1))
  |-------
{1,(rtp)}
      FORALL (i: PROC_ID):
        (y(next!1)(i) = 0 IFF (loc(next!1)(i) = 0 OR loc(next!1)(i) = 1))

Rule? (split-rho)
Splits rho, replaces and simps.
If remove-impl is t (the default), then strategy remove-impl is applied.,
this yields  5 subgoals:
yZero.1 :
;;;The inductive step of the BINV rule

{-1,(rho)}
      loc(current!1)(p!1) = 0
{-2,(rho)}
      y(next!1) = y(current!1)
{-3,(rho)}
      loc(next!1) = loc(current!1) WITH [(p!1) := 1]
{-4,(invariant)}
      FORALL (i: PROC_ID):
        IF y(current!1)(i) = 0
          THEN (loc(current!1)(i) = 0 OR loc(current!1)(i) = 1)
        ELSE NOT (loc(current!1)(i) = 0 OR loc(current!1)(i) = 1)
        ENDIF
  |-------
{1,(rtp)}
      IF y(current!1)(i!1) = 0
        THEN (loc(current!1) WITH [(p!1) := 1](i!1) = 0 OR
               loc(current!1) WITH [(p!1) := 1](i!1) = 1)
      ELSE NOT (loc(current!1) WITH [(p!1) := 1](i!1) = 0 OR
                 loc(current!1) WITH [(p!1) := 1](i!1) = 1)
      ENDIF

Rule? (split-all)
Split-all if-then-else consequents,
this yields  3 subgoals:
yZero.1.1 :
;;;The inductive step of the BINV rule

{-1,(rtp)}
      p!1 = i!1
{-2,(rho)}
      loc(current!1)(i!1) = 0
[-3,(rho)]
      y(next!1) = y(current!1)
{-4,(rho)}
      loc(next!1) = loc(current!1) WITH [(i!1) := 1]
[-5,(invariant)]
      FORALL (i: PROC_ID):
```

```
        IF y(current!1)(i) = 0
          THEN (loc(current!1)(i) = 0 OR loc(current!1)(i) = 1)
        ELSE NOT (loc(current!1)(i) = 0 OR loc(current!1)(i) = 1)
        ENDIF
  |-------
{1,(rtp)}
      y(current!1)(i!1) = 0

Rule? (inst?)
Found substitution:
i: PROC_ID gets i!1,
Using template: y(current!1)(i) = 0
Instantiating quantified variables,
this simplifies to:
yZero.1.1 :
;;;The inductive step of the BINV rule

[-1,(rtp)]
      p!1 = i!1
[-2,(rho)]
      loc(current!1)(i!1) = 0
[-3,(rho)]
      y(next!1) = y(current!1)
[-4,(rho)]
      loc(next!1) = loc(current!1) WITH [(i!1) := 1]
{-5,(invariant)}
      IF y(current!1)(i!1) = 0
        THEN (loc(current!1)(i!1) = 0 OR loc(current!1)(i!1) = 1)
      ELSE NOT (loc(current!1)(i!1) = 0 OR loc(current!1)(i!1) = 1)
      ENDIF
  |-------
[1,(rtp)]
      y(current!1)(i!1) = 0

Rule? (simp)
Flatten, assert, simplify,

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

We consider now the second branch generated by `split-rho`. Unsurprisingly, this branch is similar to its previous sibling, and so it is likely that using `split-all`, instantiating, and simplifying will complete the proof. We use the strategy `(split-all-inst ("i!1"))` which performs just these tasks – it performs the splitting and simplification of `split-all` while also instantiating with all the values in its argument list. This command closes the branch, and can likewise be used to complete the three remaining branches, concluding the proof.

The complete proof is thus:

```
(BINV)
(EXPAND "yZero")
(SPLIT-RHO)
(("1" (SPLIT-ALL)
    (("1" (INST?) (SIMP)) ("2" (INST?) (SIMP)) ("3" (INST?) (SIMP))))
```

```
("2" (SPLIT-ALL-INST ("i!1")))
("3" (SPLIT-ALL-INST ("i!1")))
("4" (SPLIT-ALL-INST ("i!1")))
("5" (SPLIT-ALL-INST ("i!1")))))
```

As a second example, we consider the property that if the y-values of two processors are non-zero then they are unique:

yUnique: ASSERTION =
    λ st:
      LET y = y(st) IN
        $\forall$ (i: PROC_ID): ($\forall$ (q: PROC_ID): $q \neq i \wedge y(i) \neq 0 \rightarrow y(q) \neq y(i)$)

yUnique: LEMMA
  is_P_reachable_valid(G(yUnique), fds)

This proof is very similar to that for `yZero`. We use it to illustrate how strategies can be used to simplify proofs. Instead of typing `binv` and then expanding `yUnique`, we can simply pass `binv` the parameter yUnique, and it does this expansion for us. Up to four parameters can be given in this way to `binv` for the purpose of expansion. Similarly, one often splits a proof using `split-rho` and then applies `split-all` or `split-all-inst` to all the subgoals. The strategy `split-rho-all` does exactly that. In order to limit splitting, it automatically undoes the `split-all-inst` if it generates more than one subgoal. (Parameters can be used to override this feature, or modify the strategy's functionality in other ways. Details can be found in the `split-rho-all` entry in Section 5.4.) In our case `(split-rho-all ("i!1" "q!1"))` completes the proof, as illustrated below.

The list of values given `split-rho-all` should be the list of all values of the quantified data type with which we want to instantiate. In this case `i!1` and `q!1` are the only two processor identity values in the proof, and it is necessary to instantiate with both. In general, however, it often suffices, and is more efficient, to instantiate with only a subset of the possible values.

```
yUnique :

  |-------
{1}   is_P_valid(G(assertion_to_TP(yUnique)), fds)

Rule? (binv "yUnique")
lemma BINV,
this simplifies to:
yUnique :
;;;The inductive step of the BINV rule

{-1,(rho)}
          loc(current!1)(p!1) = 0 AND
           y(next!1) = y(current!1) AND
            loc(next!1) = loc(current!1) WITH [(p!1) := 1]
       OR loc(current!1)(p!1) = 1 AND
           (EXISTS (m: nat):
              (FORALL (q: PROC_ID[5, N]): y(current!1)(q) < m) AND
```

```
              y(next!1) = y(current!1) WITH [(p!1) := m])
           AND loc(next!1) = loc(current!1) WITH [(p!1) := 2]
      OR loc(current!1)(p!1) = 2 AND
          (FORALL (q: PROC_ID[5, N]):
             q /= p!1 IMPLIES
              y(current!1)(q) = 0 OR y(current!1)(p!1) <= y(current!1)(q))
           AND
           y(next!1) = y(current!1) AND
            loc(next!1) = loc(current!1) WITH [(p!1) := 3]
      OR loc(current!1)(p!1) = 3 AND
          y(next!1) = y(current!1) AND
           loc(next!1) = loc(current!1) WITH [(p!1) := 4]
      OR loc(current!1)(p!1) = 4 AND
          y(next!1) = y(current!1) WITH [(p!1) := 0] AND
           loc(next!1) = loc(current!1) WITH [(p!1) := 0]
{-2,(yUnique invariant)}
      FORALL (i: PROC_ID):
        FORALL (q: PROC_ID):
          q /= i AND y(current!1)(i) /= 0 IMPLIES
           y(current!1)(q) /= y(current!1)(i)
{-3,(binv)}
      y(next!1)(q!1) = y(next!1)(i!1)
  |-------
{1,(rtp)}
      q!1 = i!1
{2,(binv)}
      y(next!1)(i!1) = 0

Rule? (split-rho-all ("i!1" "q!1"))
Tries to complete a proof using split-rho and then split-all-inst.
If this fails does simple simplification and returns goal to user,
Q.E.D.
```

We now prove a reachability invariant, which defines the set of states which are reachable. In general, most verification efforts include such an invariant, Our system expects the formula to be called `reachable`, and the lemma proving its invariance `reachableInv`. Following this naming convention allows this reachability invariant to be easily included in other, future, proofs.

reachable: ASSERTION =
$(\lambda$ st:
LET $y = y$(st), loc = loc(st) IN
$\forall$ (i: PROC_ID):
$(y(i) = 0 \equiv (\text{loc}(i) = 0 \vee \text{loc}(i) = 1)) \wedge$
$\forall$ (q: PROC_ID):$q \neq i \wedge y(i) \neq 0 \rightarrow y(q) \neq y(i)) \wedge$
$(\text{loc(st)}(i) \geq 3 \rightarrow$
$y(i) > 0 \wedge$
$(\forall$ (q: PROC_ID): $q \neq i \rightarrow \text{loc(st)}(q) < 3 \wedge (y(q) = 0 \vee y(i) < y(q)))))$

reachableInv: LEMMA
is_P_reachable_valid(G(reachable), fds)

We give three different proofs for this lemma, and the reader is encouraged to run PVS alongside, tracing through the proofs with us. The three proofs can be found as lemmas reachableInv, reachableInv2, and reachableInv3 in the file bakery_safety.pvs.

Proof 1 As in the previous proofs we start by applying `binv`. The consequent is composed of three conjunctions, the first and second of which are the invariants `yZero` and `yUnique`, respectively. We decide to split the consequent in order to prove each one separately, and thus call `split-up`. However, this generates five subgoals as the first subgoal, an if-and-only-if statement is split into two branches, and the last also generates two subgoals. The first two subgoals are the two directions of the if-and-only-if of invariant `yZero`. To discharge them we bring in the invariant using the strategy `(use-invariant "yZero" "next")`. This strategy invokes the lemma `yZero`, expands it out, and instantiates it in accordance with formula *next*. (We recall that this hidden formula was created by `binv`, and contains the equivalence *seq!1(t!1+1) = next!1*.) Typically, the desired instantiation is according to the formula *current*. If no formula is stated (i.e. only one parameter is given) then the instantiation is in accordance to formula *current*. See Section 5.4.2 for more details.

The property `yZero` must be correctly instantiated, after which simplification `simp` completes the proof. On the third branch we similarly bring in invariant `yUnique`. The fourth branch is easily completed using `(split-rho-all ("i!1"))`. The fifth branch can also be completed using `(split-rho-all ("q!1" "i!1"))`, however this will be quite time consuming, and since there is little feedback to the screen one might suspect that the system is hanging, or has encountered other problems. For this reason, we prefer to use the command `(then (split-rho)(split-all-inst ("q!1" "i!1")))`. The first command given to a `then` statement is applied, and the list of subsequent commands is applied to each of the subgoals generated by the first command. So, in this case, `split-all-inst` is applied to the six subgoals generated by `split-rho`. The advantage of using this construct over `(split-rho-all ("q!1" "i!1"))` is that one can see how the proof is progressing. The proof is presented below:

```
(BINV "reachable")
(SPLIT-UP)
(("1" (USE-INVARIANT "yZero" "next") (INST? -) (SIMP))
 ("2" (USE-INVARIANT "yZero" "next") (INST? -) (SIMP))
 ("3" (USE-INVARIANT "yUnique" "next") (INST - "i!1" "q!1") (SIMP))
 ("4" (SPLIT-RHO-ALL ("i!1")))
 ("5"
  (COMMENT "(THEN (SPLIT-RHO)(SPLIT-ALL-INST ('q!1' 'i!1')))")
  (SPLIT-RHO)
  (("1" (SPLIT-ALL-INST ("q!1" "i!1")))
   ("2" (SPLIT-ALL-INST ("q!1" "i!1")))
   ("3" (SPLIT-ALL-INST ("q!1" "i!1")))
   ("4" (SPLIT-ALL-INST ("q!1" "i!1")))
   ("5" (SPLIT-ALL-INST ("q!1" "i!1"))))))
```

This proof can be viewed using `M-x edit-proof`. We note that PVS does not store the command `(then (split-rho)(split-all-inst ("q!1" "i!1")))` as a single command. Rather, it is stored as a single application of `(split-rho)`, followed by separate applications of `(split-all-inst ("q!1" "i!1"))` on every branch. The proof stored is thus syntactically different from that typed in.

Proof 2 It seems wasteful to use the yZero invariant twice, almost identically, on two branches because of the splitting which occurred. Alternatively, we could have used `split-and` before calling `split-up`. This would split the proof into two subgoals, on the first we are asked to prove the property yZero, and on the second yZero is assumed. This has a further advantage that the proof that $loc(next!1)(i!1) >= 3$ *implies* $y(next!1)(i!1) > 0$ (the fourth branch of the previous proof) is now automatically resolved as it can be concluded directly from yZero. We thus, in fact, prove three goals rather than five:

```
(BINV "reachable")
(SPLIT-AND)
(("1" (SPLIT-RHO-ALL ("i!1")))
 ("2" (SPSI)
  (("1" (SPLIT-RHO-ALL ("q!1" "i!1")))
   ("2" (SPLIT-RHO-ALL ("q!1" "i!1"))))))
```

In this proof we also chose not to use the previously proved invariants in order to illustrate different possibilities; alternatively, the first 2 goals of this proof could have been proved by using `use-invariant`. We have also used `spsi` instead of `split-up`. Strategy `spsi` first SPlits once and then SImplifies. In contrast, `split-up` will split repeatedly until it finds nothing more to split on the specified line number (default '1' when not specified). In this case, the simpler `spsi` is sufficient.

Proof 3 In the second proof `split-rho-all` was applied on all three of the branches after the splitting occurs. In fact `split-rho-all` has an optional parameter *split-up*, defaulting to 1, which performs the strategy `split-up` (on the specified formula, *split-up*). So, if we call `(split-rho-all ("q!1" "i!1"))` consequent 1 is split up, as desired. Thus the very brief proof

```
(BINV "reachable") (SPLIT-RHO-ALL ("q!1" "i!1"))
```

suffices.

In comparing the three proofs we note that the time that they run is inversely related to their textual length. On a 670 MHz Celeron PC, the three proofs took 207, 350 and 354 seconds, respectively. Given that proof 1 used the lemmas `yZero` and `yUnique`, rather than reproving these properties, it is unsurprising that it is faster. In general, there is often a trade-off between automation and speed: the more the user directs the proof, the more efficient it can be made.

The processing time between commands in the second and third proofs may also be problematic for a user who may fear that the prover is hanging.

Proving mutual exclusion We can now prove the desired mutual exclusion property:

```
mutex: ASSERTION =
    (λ st:
       ∀ (i, q: PROC_ID):
         loc(st)(i) = 3 ∧ loc(st)(q) = 3 → i = q)

mutualExclusion: LEMMA
  is_P_reachable_valid(G(mutex), fds)
```

This property is a simple corollary of the reachability invariant. To prove it we bring in the reachability invariant using `use-invariant`, expand expressions, instantiate and split:

```
(USE-INVARIANT "reachableInv")
(INST?)
(EXPAND* "reachable" "mutex")
(INST?)
(SPLIT-ALL-INST ("q!1" "i!1"))
```

The rule `expand*` takes a list of expressions to be expanded. Alternatively, we could have used the two commands `(expand "reachable") (expand "mutex")`.

Chapter 4

Proving Response Properties

In this chapter we consider rules for proving simple response properties. These are properties of the form `is_P_valid(G(p IMPLIES F(q)))` , for assertions p and q. In the syntax of [MP95], these are formulas of the form $P \models p \Longrightarrow \Diamond q$.

For example, the formula

`is_P_valid(`G`((`λ` st: loc(st)(z) = 1) → `F`(`λ` st: loc(st)(z) = 3)), pfs)`

asserts the *accessibility* property for program BAKERY. This property ensures that whenever process z wishes to enter the critical section, as is evident by observing it at location 1, it will eventually enter the critical section at location 3.

We have formulated a number of proof rules for accessibility. The general principle is that the rule traces the progress of computations from an arbitrary p-state to an unavoidable q-state. This is done by defining a ranking function over a well-founded domain and showing that as long as a q-state is not reached, the rank will never increase and will eventually decrease. Thus we cannot have an infinite sequence of p-states without a q-state, or the rank would decrease infinitely often.

Along with the proof rules we have also constructed theories to prove domain well-foundedness (Section 4.1.4)

In Section 4.2 we illustrate the use of proof rules by proving accessibility for the bakery algorithm.

4.1 Proof Rules

In this Section we detail the proof rules that we have implemented in our system, each of which has been proved to be sound. We will not discuss the proofs of the soundness of these rules, but only give a brief intuition. The user who is interested in the soundness proofs can examine the PVS files.

The rules have associated strategies, which, similarly to the strategies of Chapter 3, invoke the rules and perform some simplification.

Before looking at individual rules we will consider some definitions that are used in this section.

Well-foundedness

As we mentioned earlier, our rules use a ranking over a well-founded domain. We define *well-founded* as follows:

Let $\prec$ be a partial order over domain $\mathcal{A}$. A subset $\mathcal{B} \subseteq \mathcal{A}$ is called *directed* if for every $a, b \in B$, $a = b$ or $a \prec b$ or $b \prec a$.

Domain $(\mathcal{A}, \prec)$ is *well-founded* if every directed subset of $\mathcal{A}$ has a minimal element.

Transition Domain

The *transition domain* $\mathcal{T}$ is a non-empty domain used to divide the transition relation ρ into transitions, one transition for each element of the transition domain.

As an example we consider BAKERY, for which we use as the transition domain the pair (*location, processor_id*). Consider a state s in BAKERY in which processors P_1 is at location 0 and processor P_2 at location 4. Domain element $(0, 1) \in \mathcal{T}$ signifies the transition by which P_1 advances from location 0 to location 1, element $(4, 1)$ signifies processor P_2 taking a step from location 4 to location 0. The transition corresponding to element $(1, 1)$ is not enabled – it will be enabled only when processor P_1 reaches location 1. In this example, the idling transition is not covered by the transition domain, showing that it need not cover all behaviors in the transition relation.

It is not necessarily the case that the transition domain is defined by the processor identity and location. However, this is often the most practical definition, and the theory `TRANS_DOMAIN.pvs` contains such a definition, as well as useful properties of this domain, such as it being finite. Alternatively, the user may define any domain he or she prefers. The theory `finite_domain.pvs` contains lemmas that may help in proving the domain finite.

Infinite Parameterized Domains

We noted, in Section 2.2, that the parameterized domain may be finite or infinite. While the WELL rule presented first is applicable in the same form to systems with finite and infinite systems, for the distributed rank rule of Section 4.1.2, we have generated versions for finite and infinite rank domains.

4.1.1 Rule WELL

Rule WELL is variation of the rule WELL-P of [SPBA00].

We have two versions of this rule, a general rule, WELL(Fig. 4.2), for proving parameterized properties of the form $\forall (i \in \mathcal{P}) : p_i \Longrightarrow \Diamond q_i$, and a simpler version, WELL-SIMPLE, without the parameterization (Fig. 4.1). We present and explain the simpler version first.

The WELL-SIMPLE rule traces the progress of a computation from an arbitrary p-state to an unavoidable q-state. With each *helpful* assertion h_t of the rule we associate the justice requirement $\mathcal{J}[t]$. Intuitively, the helpful assertion defines a set of states in which a just transition is enabled. When this just transition is taken, and ceases to be helpful, the rank decreases. Thus, the helpful set indicates a transition it would be "helpful" to take in order to decrease the rank.

Rule WELL-SIMPLE

For a PFS with

transition domain	$\mathcal{T}$,
state space	Σ,
and justice conditions	$\mathcal{J}$;
Given	
initial and goal assertions	p, q
helpful assertions	$\{h_t : t \in \mathcal{T}\}$,
a well-founded relation	$\succ$ over $\mathcal{A}$,
and ranking functions	$\{\delta_t : \Sigma \mapsto \mathcal{A} \mid t \in \mathcal{T}\}$

$$\text{W1.} \quad p \rightarrow q \vee \bigvee_{t \in \mathcal{T}} h_t$$

$$\text{W2.} \quad h_t \wedge \rho \rightarrow \left.\begin{array}{l} q' \vee \left(\bigvee_{u \in \mathcal{T}} (\delta_t \succ \delta'_u \wedge h'_u)\right) \\ \vee \left(h'_t \wedge \delta_t = \delta'_t \wedge \neg\mathcal{J}'[t]\right) \end{array}\right\} \text{For } t \in \mathcal{T}$$

$$\overline{\qquad p \Longrightarrow \Diamond q \qquad}$$

Figure 4.1: Rule WELL-SIMPLE

Premise W2 assures that the application of a transition to a state satisfying assertion h_t will never cause the rank of the state to increase. Furthermore, as long as the rank does not decrease h_t will continue to hold and $\mathcal{J}[t]$ will not. Since the system is just (weakly fair), $\mathcal{J}[t]$ must hold eventually and so the rank must eventually decrease. Due to the well-foundedness of the ranking functions, the rank cannot decrease infinitely often. Thus, we cannot have an infinite fair computation which avoids reaching a q-state.

The parameterized WELL rule, Fig. 4.2, is an extended version of the WELL-SIMPLE rule allowing one to prove parameterized properties. For example, we might want to prove in the BAKERY algorithm that for every process i, if i wants to enter the critical section then it will, i.e. $\forall i : waiting(i) \Longrightarrow \Diamond(critical(i))$.

Applying rules WELL and WELL-SIMPLE in infinite systems

There is no requirement that $\mathcal{P}$, $\mathcal{T}$ or $\mathcal{A}$ be finite.

When using the rules in infinite systems care should be taken in insuring that the ranking function $\succ$ is well-founded over $\mathcal{A}$. Ranking functions which are well-founded over parameterized rank elements $\mathcal{A}$, are not necessarily well-founded when $\mathcal{A}$ is an infinite domain. Intuitively, in a finite or parameterized system one can allow the rank to decrease finitely often for every process. In a system with an infinite number of processes, this cannot be done.

Rule WELL

For a PFS with
transition domain $\mathcal{T}$,
parameterized domain $\mathcal{P}$,
state space Σ,
and justice conditions $\mathcal{J}$;
Given
initial and goal assertions $p_i, q_i : i \in \mathcal{P}$,
helpful assertions $\{h_{t,i} : t \in \mathcal{T}, i \in \mathcal{P}\}$,
a well-founded relation $\succ$ over $\mathcal{A}$,
and ranking functions $\{\delta_{t,i} : \Sigma \mapsto \mathcal{A} \mid t \in \mathcal{T}, i \in \mathcal{P}\}$

$$\forall i \in \mathcal{P}:$$

$$\text{W1.} \quad p_i \quad \rightarrow \quad q_i \ \vee \ \bigvee_{t \in \mathcal{T}} h_{t,i}$$

$$\left.\begin{array}{lll} \text{W2.} \quad h_{t,i} \ \wedge \ \rho & \rightarrow & q_i' \ \vee \ \left[\bigvee_{u \in \mathcal{T}} (\delta_{t,i} \succ \delta_{u,i}' \ \wedge \ h_{u,i}') \right] \\ & & \vee \ \left(h_{t,i}' \ \wedge \ \delta_{t,i} = \delta_{t,i}' \ \wedge \ \neg \mathcal{J}'[t] \right) \end{array}\right\} \text{For } t \in \mathcal{T}$$

$$\overline{\qquad\qquad\qquad\qquad\qquad\qquad}$$

$$p_i \Longrightarrow \Diamond q_i$$

Figure 4.2: Parameterized well rule WELL

Using rule WELL

In order to use Rule WELL one should import the theory `WELL` (in file `WELL.pvs`) with the following parameters:

Σ	The state space
$\mathcal{A}$	The domain over which the rank values range
$\mathcal{T}$	The transition domain
$\mathcal{P}$	The parameterized domain
p	The parameterized initial assertion in the form $[\mathcal{P} \mapsto \text{Assertion}]$
q	The parameterized goal assertion in the form $[\mathcal{P} \mapsto \text{Assertion}]$
$hset$	The set of helpful assertions $h_{t,i}$ in the form $[\mathcal{T}, \mathcal{P} \mapsto \text{Assertion}]$
δ	A ranking function $\delta : [\mathcal{T}, \mathcal{P} \mapsto (\Sigma \mapsto \mathcal{A})]$
$\succ$	A well founded relation over $\mathcal{A}$

Having imported the rule, it can be applied by using the strategy `well` which invokes the lemma, expands out the definitions, and tries to prove some of the premises. (E.g. the strategy can often prove W1 automatically, leaving a simplified form of W2 for the user).

Rule WELL-SIMPLE can be found as theory `WELL_simple` in file `WELL_simple.pvs` , and can be invoked using strategy `well_simple` .

4.1.2 Rule DIST-RANK

Rule DIST-RANK

For a PFS with
transition domain $\mathcal{T}$,
parameterized domain $\mathcal{P}$,
state space Σ,
and justice conditions $\mathcal{J}$;
Given
initial and goal assertions $p_i, q_i : i \in \mathcal{P}$,
helpful assertions $\{h_{t,i} : t \in \mathcal{T}, i \in \mathcal{P}\}$,
finite rank domain $\mathcal{R}$
and ranking functions $\{\delta_{r,i} : \Sigma \mapsto \mathbb{N} \mid r \in \mathcal{R}, i \in \mathcal{P}\}$

$$\forall i \in \mathcal{P} :$$

$$\text{D1.} \quad p_i \rightarrow q_i \vee \bigvee_{t \in \mathcal{T}}^{m} h_{t,i}$$

$$\left.\begin{array}{ll} \text{D2.} & h_{t,i} \wedge \rho \rightarrow q'_i \vee h'_{t,i} \vee \left(\bigvee_{r \in \mathcal{R}} (\delta_{r,i} > \delta'_{r,i}) \wedge \bigvee_{u \in \mathcal{T}} h'_{u,i} \right) \\ \text{D3.} & h_{t,i} \wedge \rho \rightarrow q'_i \vee \bigwedge_{r \in \mathcal{R}} \delta_{r,i} \geq \delta'_{r,i} \\ \text{D4.} & \text{reachable\_valid}(\Box(h_{t,i} \rightarrow \neg \mathcal{J}[t])) \end{array}\right\} \text{For } t \in \mathcal{T}$$

$$\overline{\qquad p_i \Longrightarrow \Diamond q_i \qquad}$$

Figure 4.3: Rule DIST-RANK for systems with finite rank domain

The distributed rank rule, DIST-RANK, presented in Fig. 4.3 is a restricted version of the general WELL rule where the ranking functions assume a very special form. We note:

- The indices of the ranking function, the rank domain $\mathcal{R}$, is not necessarily the transition domain, $\mathcal{T}$, as was the case in the WELL rule.
- The rank domain $\mathcal{R}$ must be a *finite type.* (There is no such restriction on $\mathcal{T}$.)
- The rank values are natural numbers.
- Premise D4 requires that the property be P-reachable valid, and not just P-valid.

Intuitively, as long as a q_i-state is not reached $h_{t,i}$ will hold for some t. The ranking functions $\delta_{r,i}$ never increase for any r (D3). At every step either at least one δ_r decreases, or the same h_t holds in the next step as did in the current step (D2). Since $h_{t,i}$ implies $\neg\mathcal{J}[t]$ (D4), no $h_{t,i}$ can hold forever in a fair computation, and so eventually the rank value of the rank function must decrease in some domain. Since the range of the ranking functions is $\mathbb{N}$, they are well-founded and cannot decrease infinitely often. Thus, eventually a q_i-state must be reached.

In Section 4.2 we demonstrate how this rule can be applied to the BAKERY example.

Rule DIST-RANK-SIMPLE: a non-parameterized version of rule DIST-RANK

Just as we provided a non-parameterized version, WELL-SIMPLE, of rule WELL, we provide a non-parameterized version, DIST-RANK-SIMPLE, of rule DIST-RANK.

Rule DIST-RANK-INFINITE: allowing infinite rank domains

If the rank domain, $\mathcal{R}$, is infinite then the above rule is not sound. Intuitively, though $\delta_{r,i}$ may be finite, and monotonically decreasing for each r, if there are infinitely many r then δ can decrease infinitely often and is not well-founded.

Typically, the rank domain is the parameterized domain, $\mathcal{P}$, which can be infinite. We have therefore devised an infinite version of the DIST-RANK rule, DIST-RANK-INFINITE. This rule is more unwieldy to work with and though it is applicable to finite systems, we suggest that it not be used when the simpler, finitary, version suffices.

We found it necessary in premises D2 and D6 to use *reachable valid* rather than the weaker *valid*. Intuitively, this is because the primed state in D3 – D5 is not necessarily feasible, i.e. it is possible that it is never part of a computation. We assume that this is an *e*xtendible PFS, that is a PFS in which any prefix of a run satisfying initiality and consecution, can be extended into an infinite initialized run. Any system with an idling transition is extendible. (The concept of extendibility is fully defined as property `P_extendible` in theory `PFS`). Since the system is extendible, the primed state is part of an initialized run, and so properties which are reachable valid hold in it.

Typically, in proving safety properties one does not use fairness constraints, and so proves them reachable valid rather than valid. The use of reachable validity in premises D2 and D6, though notationally inelegant, is not particularly difficult to use.

Whereas in finite systems we can allow the ranks of each element in the rank domain $\mathcal{R}$ to start high and then decrease, this may violate the well-founded requirement in infinite systems: If $\mathcal{R}$ is infinite and all elements start with non-zero values $\delta_{r,i}$ can be decremented infinitely often, by decrementing finitely often for each $r \in \mathcal{R}$.

Therefore, we restrict ourselves to ranking functions which, for any given state, assign a positive rank to only finitely many elements of the rank domain. The predicate "is_finite" in D2 is defined as $\exists N, (f : [(S) \mapsto [0..N-1]]) : injective?(f)$.

This requirement introduces problems, however. Typically, one does not know at the outset of the run which processes will "participate", and should be given non-zero rank. This is generally determined later, once process i, for which accessibility is being proved, reaches a certain point in the program. To overcome this problem we use a rank changing assertion, ra_i, which can change at most once, from true to false, before a q-state is reached (D3). Intuitively, this assertion changes value exactly when process i reaches the point at which it can be determined which processes will participate. Ranks $\delta_{r,i}$ are allowed to increase at this point.

Rule DIST-RANK-INFINITE

For an extendible PFS with	
transition domain	$\mathcal{T}$,
parameterized domain	$\mathcal{P}$,
state space	Σ,
and justice conditions	$\mathcal{J}$;
Given	
initial and goal assertions	$p_i, q_i : i \in \mathcal{P}$,
helpful assertions	$\{h_{t,i} : t \in \mathcal{T}, i \in \mathcal{P}\}$,
rank domain	$\mathcal{R}$[a],
ranking functions	$\{\delta_{r,i} : \Sigma \mapsto \mathbb{N} \mid r \in \mathcal{R}, i \in \mathcal{P}\}$
and rank changing assertion	$ra_i : i \in \mathcal{P}$

$$
\begin{array}{l}
\forall i \in \mathcal{P} : \\
\begin{array}{lll}
\text{D1.} & p_i \quad \rightarrow & q_i \ \vee \ \bigvee\limits_{t \in \mathcal{T}}^{m} h_{t,i} \\
\text{D2.} & \multicolumn{2}{l}{\text{reachable\_valid}(\Box(\text{is\_finite}\{\mathrm{r} \in \mathcal{R} | \delta_{\mathrm{r,i}} > 0\}))}
\end{array} \\
\left.\begin{array}{lll}
\text{D3.} & h_{t,i} \wedge \rho \quad \rightarrow & q' \ \vee \ (\neg ra_i \rightarrow \neg ra'_i) \\
\text{D4.} & h_{t,i} \wedge \rho \quad \rightarrow & q' \ \vee \ h'_{t,i} \ \vee \\
 & & \left((ra_i \ \wedge \ \neg ra'_i \ \vee \ \bigvee\limits_{r \in \mathcal{R}} (\delta_{r,i} > \delta'_{r,i})) \ \wedge \ \bigvee\limits_{u \in \mathcal{T}} h'_{u,i} \right) \\
\text{D5.} & h_{t,i} \wedge \rho \quad \rightarrow & q' \ \vee \ ra_i \ \wedge \ \neg ra'_i \ \vee \ \bigwedge\limits_{r \in \mathcal{R}} \delta_{r,i} \geq \delta'_{r,i} \\
\text{D6.} & \multicolumn{2}{l}{\text{reachable\_valid}(\Box(h_{t,i} \rightarrow \neg \mathcal{J}[t]))}
\end{array}\right\} \text{For } t \in \mathcal{T} \\
\hline
\qquad p_i \Longrightarrow \Diamond q_i
\end{array}
$$

[a] There is an assumption that $\exists (f : [\mathcal{R} \mapsto \mathbb{N}]) : \mathit{injective?}(f)$.

Figure 4.4: Rule DIST-RANK-INFINITE for infinite systems

Using rules DIST-RANK, DIST-RANK-SIMPLE and DIST-RANK-INFINITE

Rule DIST-RANK can be imported as theory `DIST_RANK` (in the file `DIST_RANK.pvs`), rule DIST-RANK-SIMPLE as theory `DIST_RANK_simple` (in the file `DIST_RANK_simple.pvs`), rule DIST-RANK-INFINITE as theory `DIST_RANK_infinite` (in the file `DIST_RANK_infinite.pvs`). Theories `DIST_RANK` and rule `DIST_RANK_infinite` take the following parameters (the parameter ra is not needed `DIST_RANK`):

Σ	The state space
$\mathcal{T}$	The transition domain
$\mathcal{R}$	The rank domain
$\mathcal{P}$	The parameterized domain
p	The parameterized initial assertion in the form $[\mathcal{P} \mapsto \text{Assertion}]$
q	The parameterized goal assertion in the form $[\mathcal{P} \mapsto \text{Assertion}]$
$hset$	The set of helpful assertions $h_{t,i}$ in the form $[\mathcal{T}, \mathcal{P} \mapsto \text{Assertion}]$
ra	Then rank changing assertion in the form $[\mathcal{P} \mapsto \text{Assertion}]$
δ	A ranking function $\delta : [\mathcal{T}, \mathcal{P} \mapsto (\Sigma \mapsto \mathbb{N})]$

Theory `DIST_RANK_simple` has the same parameters as theory `DIST_RANK` , except that the parameterized domain $\mathcal{P}$, is omitted.

Having imported the rule, it can be applied by using the strategy `dist_rank`, `dist_rank_simple` or `dist_rank_infinite`, respectively. This strategy will invoke the lemma, expand out the definitions, and try to prove some of the premises.

The importing generates some type-checks. In the case of rules `dist_rank` and `dist_rank_simple`, one must prove that the rank domain is finite. When using rule `dist_rank_infinite` one must prove that the system is extendible, and that there is a mapping from $\mathcal{R}$ to $\mathbb{N}$.

4.1.3 Rule PRE-ORDER

Rule PRE-ORDER (Fig. 4.5) is a special case of rule DIST-RANK in which the assertions in the helpful set *hset*, for any state, can be ordered by a *total pre-order*. It does not include parameterization by the parameterized domain.

We recall that a relation is a pre-order if it satisfies the properties of:

Q1. *Reflexivity*

$$\forall i : [1..m] : i \preceq i$$

Q2. *Transitivity*

$$\forall i, j, k : [1..m] : i \preceq j \ \wedge \ j \preceq k \quad \rightarrow \quad i \preceq k$$

A *total* (or *dichotomous*) pre-order satisfies, in addition,

Q3. $\quad \forall i, j : [1..m] : i \preceq j \ \vee \ j \preceq i$

Element $t \in \mathcal{T}$ is *minimal* if for every $u \in \mathcal{T}$, $t \preceq u$.

For every total pre-order defined over a finite, non-empty domain $\mathcal{T}$ there always exist one or more minimal elements, as stated by the following claim (and formally verified in the PVS theory for rule PRE-ORDER):

Rule PRE-ORDER

For an extendible PFS with
finite transition domain $\mathcal{T}$,
state space Σ,
and justice conditions $\mathcal{J}$;
Given
initial and goal assertions p, q
assertion φ,
helpful assertions $\{h_t : t \in \mathcal{T}\}$,
finite rank domain $\mathcal{R}$,
pre-order $\preceq : \Sigma \mapsto 2^{\mathcal{T}\times\mathcal{T}}$,
and ranking functions $\{\delta_r : \Sigma \mapsto \mathbb{N} \mid r \in \mathcal{R}\}$

$$
\begin{array}{lll}
\text{P1.} & p & \rightarrow \; q \;\vee\; \varphi \\
\text{P2.} & h_t \;\wedge\; \rho & \rightarrow \; q' \;\vee\; h_t' \;\vee\; (\varphi' \;\wedge\; \bigvee_{r\in\mathcal{R}} \delta_r > \delta_r') \\
\text{P3.} & h_t \;\wedge\; \rho & \rightarrow \; q' \;\vee\; \bigwedge_{r\in\mathcal{R}} \delta_r \geq \delta_r' \\
\text{P4.} & \multicolumn{2}{l}{\text{reachable\_valid}(\Box(h_t \;\rightarrow \neg\mathcal{J}[t]))}
\end{array}
\left.\vphantom{\begin{array}{l}a\\a\\a\\a\end{array}}\right\} \text{For } t \in \mathcal{T}
$$

$$
\text{P5.} \quad \text{reachable\_valid}(\Box(\varphi \;\wedge\; \bigwedge_{t\in\mathcal{T}} m \preceq t \rightarrow h_m))
$$

$$
\overline{\qquad\qquad p \Longrightarrow \Diamond q \qquad\qquad}
$$

Figure 4.5: Rule PRE-ORDER

Claim 1 *If $\preceq$ is a total pre-order defined over finite, non-empty domain $\mathcal{T}$, then there always exists a $t \in \mathcal{T}$, such that*

$$\forall u \in \mathcal{T} : t \preceq u.$$

The pre-order we use in the rule is formally defined as

$$\preceq : \Sigma \mapsto 2^{\mathcal{T}\times\mathcal{T}}$$

For readability, we will write $t \preceq_s u$ instead of $(t, u) \in \preceq (s)$. When state s is understood from the context we write simply $t \preceq u$.

The rule requires that if $t \in \mathcal{T}$ is minimal for state s then assertion $h_t \in hset$ holds at state s. By the claim, every state has at least one minimal t, and so it has at least one helpful transition, h_t.

The advantage of this rule over rule DIST-RANK is that there is no need to prove a premise of the form of premise D2 of DIST-RANK:

$$D2. \quad h_{t,i} \;\wedge\; \rho \rightarrow q_i' \;\vee\; h_{t,i} \;\vee\; [\bigvee_{r\in\mathcal{R}} (\delta_{r,i} > \delta_{r,i}') \;\wedge\; \bigvee_{t\in\mathcal{T}} h_{t,i}']$$

When proving that a premise of the form $\bigvee_{t \in \mathcal{T}} h'_{t,i}$ holds in the system, one typically has to "give" the system a suitable index t for every state s. This often necessitates significant user interaction with the theorem prover. In this rule, one proves instead that for every reachable state s, for every m such that m is a minimum for s (P5), h_m is in the helpful set of s, thus proving that a helpful assertion holds at every reachable state.

The rule ensures that the sequence of states between the initial p-state and the desired q-state will all be φ-states. Intuitively, this may be a characterization of the states which it is possible to visit between a p and a q-state. This property may not be true of all reachable states. For example, if x is a counter that is monotonically incremented we may want to prove that $x = 2 \Longrightarrow F(x = 4)$. An appropriate φ might state that $x = 2 \ \vee \ x = 3$. Other states (e.g. $x = 6$) may be reachable from the initial state, however they will not be found on a path to a state where $x = 4$. If such a characterization is not useful one can set φ to `true`.

Using rule PRE-ORDER

Rule PRE-ORDER can be imported as theory `PREORDER` (in the file `PREORDER.pvs`). The importing generates type-checks requiring the user to prove that $\mathcal{R}$ and $\mathcal{T}$ are finite, non-empty types, that $\preceq$ is a total pre-order, and that the PFS is extendible. Theory `PREORDER` requires the following parameters:

Σ	The state space
$\mathcal{T}$	The transition domain
$\mathcal{R}$	The rank domain
p	The initial assertion
q	The goal assertion
φ	An assertion characterizing states that can occur between a p and q state
$hset$	The set of helpful assertions h_t in the form $[\mathcal{T} \mapsto \text{Assertion}]$
$\preceq$	The pre-order, in the form : $[\Sigma \mapsto ([\mathcal{T}, \mathcal{T}] \mapsto \text{Boolean})]$
δ	A ranking function $\delta : [\mathcal{T} \mapsto (\Sigma \mapsto \mathbb{N})]$

Having imported the rule, it can be applied by using the strategy `preorder`. This strategy will invoke the lemma, expand out the definitions, and try to prove some of the premises.

4.1.4 Proving Well-foundedness

Rule WELL required that the comparator $\succ$ of the ranking values $\mathcal{A}$ be well-founded. However, proving well-foundedness is often a non-trivial exercise in itself, even in the case of obviously well-founded comparators such as the greater-than operator $>$ over the natural numbers. We have therefore proved the well-foundedness of comparators over frequently used comparator domains (natural numbers, sets of natural numbers) and rules for combining well-founded domains into a compound well-founded domain.

Natural Numbers

The theory nat_well_founded contains the lemma nat_wellfounded which proves that the greater than operator is a well-founded order over the natural numbers. nat_well_founded This theory also has a lemma upto_n_wellfounded which proves that the greater than operator defines a well-founded order over the natural numbers in the domain $[0 \dots N]$.

The parameter N must be given when importing the theory. This parameter is the value of the largest element in the domain when using lemma upto_n_wellfounded. Its value is ignored when using lemma nat_wellfounded.

Sets

The theory set_well_founded proves the well-foundedness of the strict superset operator over sets of elements, where each element is a natural number in the domain $[0 \dots N]$.

This theory can be invoked as lemma set_wellfounded after invoking the theory with parameter N.

Composition of Well-founded Domains

It is often useful to combine different domain elements into one compound structure, whose elements are compared lexicographically, using the comparator operators of each domain. When each comparator is well-founded over its domain, this evaluation is also well-founded. In order to allow the user to build and use such domains we have devised two theories that can be used (singly, together or possibly repeatedly) to prove the well-foundedness of such an operator.

Let D_1 and D_2 be non-empty domains with operators gt_1 and gt_2 such that gt_1 is well founded over D_1, and gt_2 over D_2.

We define a compound domain DOUBLE_RANK_ELT as a structure with two fields, first of type D_1 and second of type D_2. Let d and e be two elements of DOUBLE_RANK_ELT. The operator d_gt is defined over DOUBLE_RANK_ELT as follows:

$$d\_gt(d, e) := gt_1(first(d), first(e)) \ \lor \ first(d) = first(e) \land gt_2(second(d), second(e))$$

The well-foundedness of this operator is proved in lemma compound_gt_wellfounded in theory two_well_founded. The theory takes four parameters:

D_1, D_2	Non-empty domains
gt_1, gt_2	Well-founded operators over D_1 and D_2, respectively

We now consider the case in which the same domain is repeatedly arbitrarily many times to obtain the compound domain: COMPOUND_RANK_ELT is an array of N fields of domain D_1. Let g, h be any two elements of COMPOUND_RANK_ELT. The operator c_gt over COMPOUND_RANK_ELT is defined as:

$$c\_gt(g, h) = \exists i : gt1(g(i), h(i)) \ \land \ \forall j : j < i \rightarrow g(j) = h(j)$$

where i and j are natural numbers in the domain $[0, \dots, N]$.

The well-foundedness of `c_gt` over `COMPOUND_RANK_ELT` is proved in lemma `compound_gt_wellfounded`. The theory takes 3 parameters:

The theory takes four parameters:

D_1	Non-empty domain
N	Positive natural number
gt_1	Well-founded operator over D_1

Renaming

Theory `two_well_founded` proves the well-foundedness of operators over a very particular structure `[first, second]`. The user, however, may not want to maintain this naming convention, and so we have proved that the renaming of fields is irrelevant. In fact, we have proved a stronger property.

Let D_1 and D_2 be non-empty domains with operators gt_1 and gt_2.

Let d_1 and e_1 be elements of D_1, and d_2, e_2 be elements of D_2.

Let m be any mapping such that $gt_1(d_1, e_1) \iff gt_2(m(d_2), m(e_2))$

Lemma `equivalent` proves that the well-foundedness of gt_1 over D_1 implies the well-foundedness of gt_2 over D_2.

These renaming properties are found in theory **equivalent_wf_functions**, which takes as parameters

D_1, D_2	Non-empty domains
gt_1, gt_2	Binary comparators over D_1 and D_2, respectively

4.2 Proving Accessibility in BAKERY

In this Section we take a look at how the DIST-RANK rule (Section 4.1.2) can be used to prove accessibility of the critical region in BAKERY. This proof can be found in the `bakery_dist_rank` theory in the dump file accompanying this tutorial. The dump file also contains theories proving accessibility using the WELL-SIMPLE and PRE-ORDER rules, `bakery_well` and `bakery_preorder`, respectively. These files are commented, both the the definitions and the proof scripts, and the user may browse through them if he so desires. However, for reasons of space we will explain only the proof using the DIST-RANK rule.

The property that we would like to prove is that if a processor is waiting to enter the critical region (i.e. is at location 1) it will eventually do so (i.e. it will reach location 3). Our assertions characterizing the initial and goal states (p and q in the explanation of rule DIST-RANK) are:

```
waiting(i): ASSERTION = (λ st: loc(st)(i) = 1)
critical(i): ASSERTION = (λ st: loc(st)(i) = 3)
```

and the formula we want to prove is:

```
accessibility: LEMMA ∀ z: is_P_valid(G(waiting(z) ⊃ F(critical(z))), pfs)
```

We must define a helpful set and ranking function. Intuitively, at every step we want to define some step, by some processor as helpful inasmuch as it will bring z closer to entering the critical section. Taking any such helpful transition should also decrease the rank.

When processor z is at location 1, there are no conditions preventing it from progressing to location 2, and this is the most helpful step that can occur in the system. However, once z is at location 2, it cannot progress if there is any other processor in the system with smaller y-value. All processors with smaller y-values must exit the critical section and set their y-value to 0 before z can continue. That is, they must reach location 0. Only the processor with the smallest y-value is actually able to progress, any other processors with non-zero y-values must be at location 2, and, like z, are unable to progress. The most helpful transition is for the processor with the smallest y-value to take a step. More formally, a transition $d \in \mathcal{T}$ is *helpful* for z at state $st \in \Sigma$ under the following conditions:

```
IF loc(d) = 1
   THEN pid(d) = z ∧ loc(st)(z) = 1
ELSE loc(st)(z) = 2 ∧ loc(d) > 1 ∧ small(st) = pid(d) ∧ loc(st)(small(st)) = loc(d)
ENDIF
```

where `small` returns the processor id of the processor with the smallest non-zero y-value at a given state, if such a processor exists.

We now consider the ranking function. The rank must decrease whenever a helpful transition is taken. When z is at location 1, the only helpful transition is for it to more to 2, and this is easy to incorporate into a rank function. When z is at location 2 we have the rank decrease whenever a processor moves along the path from location 2 to location 0. Such movement is helpful as the processor must be the processor with smallest y-value (processor `small`) in order to be able to progress from location 2. On moving from location 4 to location 0, the processor's y-value is set to 0, it is no longer the `small` processor, and its actions are no longer helpful. The processor may continue from location 0 to 2, but this time it will have a larger y-value than z and will not be able to progress further before z has entered the critical region. In order to ensure that ranks are monotonically decreasing, we differentiate, at location 2, between processors with rank greater than that of z, who have no future helpful transitions, and those with smaller y-values, with a future of helpful transitions leading them back to location 0. We formulate a ranking function:

```
stage(st, p, z): upto[5] =
    COND loc(st)(p) = 2 & y(st)(p) < y(st)(z) → 5,
         loc(st)(p) = 3 → 4,
         loc(st)(p) = 4 → 3,
         loc(st)(p) = 0 → 2,
         loc(st)(p) = 1 → 1,
         loc(st)(p) = 2 & y(st)(p) ≥ y(st)(z) → 0
    ENDCOND

rank: [PROC_ID, PROC_ID → [STATE → nat]] =
    λ p, z: λ st: IF loc(st)(z) = 2 THEN stage(st, p, z) ELSE 5 ENDIF
```

As is often the case, the rank domain, $\mathcal{R}$, equals the parameterized domain, $\mathcal{P}$, which is the processor identifier (`PROC_ID`).

We note that in the above function, the rank of a processor p different from z decreases as p moves from location 0 (rank 2) to location 2 (rank 0). This decrease is not strictly necessary, as these transitions are not helpful. As it is designed, when processor z is at location 2, the rank will decrease whenever any processor takes a transition. This simplifies the proof by obviating the need to prove that a transition which does not decrease the rank is not helpful.

Having formulated the necessary definitions, we now try to prove accessibility.

In order to fully understand our explanations the reader should open, and step through, (using the `M-x step-proof` command) the proof while reading the explanation. We begin by applying strategy `dist_rank(waiting, critical)`. This strategy invokes the distributed rank rule, while expanding out the expressions `waiting` and `critical`. It returns 4 subgoals, one for each of the four premises of rule DIST-RANK(Fig. 4.3).

The first subgoal

```
;;;Proving D1: is_P_state_valid(START_ELT implies (GOAL_ELT or disj_set(hset)), pfs)

{-1,(dist_rank)}
      loc(seq!1(t!1))(z!1) = 1
  |-------
{1,(dist_rank)}
      1 = 3
{2,(dist_rank)}
      EXISTS (d: TRANS_DOMAIN[5, N]):
        IF loc(d) = 1 THEN pid(d) = z!1 ELSE FALSE ENDIF
```

can be discharged by supplying the system with the transition which is helpful in the starting state, the transition of processor z from location 1 to location 2 i.e.
`(inst + "(# loc := 1, pid := z!1 #)")`.

For reasons of space, the second subgoal will not be copied here, however, the user can view it in the PVS-prover.

We first note that the strategy has labelled the sequents in the subgoal, and compare these labels with the expressions in premise D2 of rule DIST-RANK. The reachability invariant has been introduced and been labeled `reachable`, $h_{t,z}$, an assertion that there is a helpful transition, is labeled `hset` and the transition relation, ρ, is labeled `rho`. Considering the consequents, sequent `same` asserts that the same transition is helpful in the next state ($h'_{t,z}$), sequent `dec` asserts that the rank decreases ($\bigvee_{r\in\mathcal{R}}(\delta_{r,z} > \delta'_{r,z}) \;\wedge\; \bigvee_{t\in\mathcal{T}} h'_{t,z}$). The sequents `rtp` asserts that the goal assertion, q'_z holds.

We noted before that if z is in location 2, then any non-idling transition will cause the rank to decrease. In fact, moving from location 1 to 2 also causes the rank to decrease. The only case in which the rank can be constant is if the z remains in location 1. We split off this simple case, which can be completed by `(split-rho-all)`.

We now look at the remaining case. Since we know that the rank is going to decrease, we are going to focus on sequent `dec`. After expanding out the definition of 'stage', we divide the proof into two subgoals - proving that there is some helpful transition in the next stage, and then that the rank decreases.

We know that the transition by the next state's `small` processor will be helpful in the next state, and so can instantiate this as the helpful transition. Sequent `same` seems superfluous and so we hide it to simplify the proof. Examining `dec` shows it to be dependent on the properties of `small`: if `small(next)` is at location 1 (`loc(next!1)(small(next!1)) = 1`) then it has y-value of 0 and by the definition of `small`, this can only occur if there are no processors with non-zero y-value. Since z has non-zero y-value (it is not at location 0 or 1), we have an immediate contradiction. However, these properties of `small` do not appear in the sequent - nowhere do we see it explicitly stated that process `small` is the process with the smallest non-zero y-value.

Pertinent properties of the `small` function are defined in lemma `smallSmall`, which asserts that as long as there is some processor q with non-zero y-value, then `small` is indeed the processor with the smallest value. Our definition and usage of the `small` function is perpendicular to the temporal rules we are discussing. We will therefore use these definitions with little explanation: the user familiar with PVS and its epsilon expressions should have no difficulty understanding these definitions; we believe that other users will be able to suffice with our brief explanation.

After introducing lemma `smallSmall`, the proof can be concluded with the strategy `(split-rho-all ("small(next!1)" "z!1"))`. (The strategy instantiates all quantified formulas with the values `small(next!1)` and `z!1`, splits the transition relation, rho, and simplifies.)

We now consider the second branch, showing that there is some processor for which the ranking function has decreased. If this was an idling transition, then, of course, this is not the case and we instead show that the same transition will remain helpful. We split the transition relation to consider the idling transition `(spsi "rho")`. We then hide sequent `dec`, as we know it will not be used, and call `(split-all)`.

We next consider the non-idling case. We skolemize, and in the transition relation, `rho`, the processor that took a step is allocated the skolem variable `p!1`. This is the processor whose rank decreased, and so we instantiated `dec` with `p!1`. We hide `same`. Applying `(split-rho-all)` returns two goals. In the first we see immediately that by instantiating sequent -7 with `z` we can conclude the proof by generating a contradiction with sequent -1. (Intuitively, if we return to our PVS formulation of bakery, Fig. 2.3, at location 1 a processor chooses a value, m, larger than the y-values of the other processors. The maximality of m is represented in sequent -7, while in -1 we have a false assertion that m is less that the y-value of z.) In the second branch we are at a situation where processor `p!1` took a transition from location 2. However, sequent 3, asserts that the y-value of `p!1` is less that that of z, a contradiction. The sequent is conclude with `(split-all-inst ("p!1" "z!1"))`.

The third subgoal in our proof requires that we show that the rank of processors never increases. We hide the sequent `hset`, after which the goal can be completed with the command `(split-rho-all ("z!1" "r!1")`.

The forth subgoal in our proof, corresponding to premise D4 of the rule, is to prove that if a transition is helpful at a state, then it is not just there. We use `(split-all -)` to split out all the nested if-then-else statements. The first subgoal requires that we show that there is always some natural number m greater than the y-values of all processors in the system. This purely mathematical proof is imported as lemma `mExists`, and the proof easily completed. The second subgoal can be completed by importing lemma `smallSmall`, completing the proof.

Chapter 5

Strategies

In order to work more efficiently with PVS we have developed a number of strategies which perform tasks which either are frequently repeated in proofs (e.g. `simp`) or perform actions which would otherwise be long or complicated to input manually (e.g. the recursive `inst-upto` strategy).

Some strategies perform functions which have proved useful over many proofs and are not specific for our implementation of temporal logic (e.g. `split-all`) while others are specific to this domain (e.g `well` which introduces the WELL rule).

In Section 5.1 we explain the use of strategy parameters. Section 5.2 discusses our general approach to sequent simplification, and in Section 5.3 we list the strategies that have been created to facilitate this approach. In Section 5.4 we detail strategies tailored to working with our temporal logic theories. (While many of the strategies in Section 5.3 were developed while working with temporal logic theories, they are potentially applicable in other contexts too.)

Some strategies are seldom used, or are called almost exclusively by other strategies. At the end of Sections 5.3 and 5.4 are comments on which of the strategies are most often used directly when building a proof.

5.1 Formal and Actual Parameters of Strategies

Strategies may have required and / or optional parameters. Generally there is no need to set the optional parameters, and they can be left at their default values. (In this chapter, and in the the PVS help command, optional parameters are bracketed with their default values.)

When invoking a strategy actuals are associated with formal parameters according to the order in which they are given. If there are fewer actuals than formals, then those parameters for which no actual was provided are bound with their default values. To give parameter values out of order, or without specifying a previous parameter in the list, one must name the formal when giving the actual.

For example, consider the strategy `split-all-inst` with parameters values, (inst-into -), (split-up 0), (split-all -), (remove-impl? t), (hide? t)

Parameter values is required, all other parameters are optional, and are bracketed with their default values.

If one enters `(split-all-inst ("a") 2 5)` then ("a") is bound to values, 2 to inst-into and 5 to split-up, while split-all, remove-impl? and hide? retain their default values of -, t and t, respectively.

To enter parameters out of order, one needs to specify the name of the parameter, precede by a colon. E.g., to specify a split-up value while leaving inst-into at its default one can enter `(split-all-inst ("a") :split-up 5)` or even `(split-all-inst :split-up 5 :values ("a") )`.

When using the PVS prover, one can see the list of parameters, and their default values, but using the command `(help` *strategy-name*`)`. Help information from the strategy file is also provided.

Calling `(help split-all)` gives the information
`(SPLIT-ALL-INST/ VALUES &OPTIONAL (INST-INTO -) (SPLIT-UP 0) (SPLIT-ALL -) (REMOVE-IMPL? T) (HIDE? T))` as well as a description of the use of the parameters.

5.2 Approach to Sequent Simplification

From experience we have become aware of a number of typical properties of sequents:

1. It is generally more efficient to split consequents than antecedents. This is because the antecedents are typically far more numerous, often including a number of properties which are not needed on every branch of the proof. These may include invariants and properties of the transition relation. Splitting on these often creates many subgoals with identical proofs.

2. Antecedents of the form *if a then b else false* or *if a then false else b* can be split without generating extra subgoals. Similarly for consequents of the form *if a then b else true* and *if a then true else b*.

   Splitting up these formulas may simplify the proof – it definitely cannot complicate it.

3. Useless formulas such as antecedent formulas *a or true* and *false implies a* or consequent formula *a and false* are often generated. Removing them improves readability.

4. If one uses the `split` command to split on an antecedent of the form *a implies b* one obtains two subgoals, one with *not a* and the other with *b*. However, it is typically preferable to have *a and b* on the second branch.

   Similarly, when splitting on *a or b* it is often desirable to obtain the subgoals *a* and *not a and b* rather than simply *a* and *b*.

5.3 General Strategies

`case-split` Parameters: variable (max 10)
Case splits variable on the values in the range [0, max]. E.g. `(case-split "i!1" 1)` will generate three subgoals, one for each case of *i!1 = 0*, *i!1 = 1* and $i!1 > 1$. If the domain of *i!1* is $\{0, 1\}$ this last goal is automatically discharged.

This is useful for splitting on the values of finite domain variables, however it is very inefficient and slow running.

`clean-up` Parameters: none
Hides all antecedents of the form *false implies a, a implies true, true or a* and *a or true*, as well as consequents of the form *a and false* or *false and a.*

`expandm` Parameters: values, (fnum *), (repeat? t)
This strategy applies (expand “a” fnum) to every the expression “a” in the list values (which must be bracketed). That is, all occurrences of the expression occurring in line(s) fnum are expanded. When fnum is left at its default * value, all occurrences of the values are expanded. If repeat? is set to t, then these expansions are performed repeatedly. If it is set to f, each expression is expanded once only. The result of this is that if the expansion of expression “b” reveals an expression “a” where “a” precedes “b” in the values list, this instance of “a” will not be expanded.

Note: This strategy is very similar to the inbuilt PVS rule `expand*`. `(expand* values)` is equivalent to `(expandm (values) :repeat? f)`. That is, `expand*` allows one to give to list of formulas without brackets, but does not repeat the expansions. It also does not allow one to limit the formulas where the expansion occurs.

`inst-from-upto` Parameters: min, max, (fnum +)
Instantiates the formula fnum with all the natural numbers n in the range $[min, max]$. e.g. `(inst-from-upto 2 4 -2)` will perform `(inst-cp -2 "2") (inst-cp -2 "3") (inst -2 "4")`.

`inst-set` Parameters: fnum, values, (hide? t)
Instantiates all values in the list value into the formula fnum e.g. `(inst-set 2 ("i!1" "p!1"))` will create 2 instantiations of formula 2, one with instantiation `i!1`, one with `p!1`.

If hide? is t (default), then the uninstantiated version of formula fnum is labeled ‘_inst_p” and hidden.

`inst-upto` Parameters: max, (fnum +)
Instantiates the formula fnum with all the natural numbers n in the range $[0, max]$.

`instbest` Parameters: (fnum *)
This is simply `(inst? fnum :if-match best)`. That is, `instbest` uses an instantiation that does not generate type-checks, if such an instantiation exists. Often generates better

instantiations than simply `inst?`, but is considerably slower.

`lazygrind` Parameters: none
First performs `grind` with no instantiation, and then with instantiation. This often generates better instantiations than a simple `grind` would. (Strategy derived from PVS manual.)

`lemma-l` Parameters: lem, (lab lem)
Applies a lemma, with labeling. The first parameter, lem, gives the name of the lemma to be applied. The lemma is labeled with the label given in the second parameter, if any, otherwise with name of the lemma.

`name-replace-label` Parameters: new, old, (lab new), (hide? t)
Replaces formula old with formula new, labeling the formula 'old = new' with label lab. If lab is not given, then new is used as the label. This equality formula is hidden if hide? is t.

(If labeling is not required, then the inbuilt `name-replace` strategy can be used.)

`reduce-all` Parameters: (sko? t)
Repeatedly calls `reduce-if` and `simp` while also replacing any newly generated formulas. If sko? is true (the default) the strategy also performs skolemization.

Note: This strategy never generates more than one goal.

`reduce-if` Parameters: (toreplace nil), (exclude nil)
Splits formulas of the form *if a then b else c* where *b* or *c* are *false* in the case of antecedents, *true* in the case of consequents.

Formulas in the toreplace list are replaced after the splitting occurs, and those in the exclude list will not be chosen to split. These parameters are very seldom used.

This strategy is seldom used directly, but rather as a component of other strategies.

`remove-iff` Parameters: none
This is a more restricted form of `remove-impl`, which converts *a iff b* formulas into *if a then b else not b*.

This conversion is declared to be a rewrite rule, so that any further *iff* formulas that are brought in will also be converted.

This strategy is particularly useful in removing iff consequent formulas, so that they can be split using `split-all`.

This strategy can only be used if the theory `remove-impl` has been imported into the theory being worked on.

`remove-impl` Parameters: none
Converts *a implies b* formulas into the form *if a then b else true* and *a iff b* into *if a then b else not b*.

Furthermore, these conversions are declared to be rewrite rules, so that any further *implies* or *iff* formulas that are brought in will also be converted.

This strategy is often called before using `split-all` (and particularly `split-all -`) to change formulas into a form that `split-all` can simplify.

This strategy can only be used if the theory `remove-impl` has been imported into the theory being worked on.

Caution: This strategy can make the text of the sequents far longer and less readable.

`save` : Parameters fnum (lab nil)
"Saves" formula fnum by hiding and then revealing it. If fnum is a number rather than a label then a label should be provided as a second parameter. In this case the formula is labeled before being hidden. The purpose of this strategy is to ensure that the formula will be available when needed later in the proof. It sometimes happens that a formula "disappears" by reducing to true or through substitutions and is no longer used by the prover. One can deal with this problem by saving the formula before it disappears and then revealing it when it is needed.

`simp` Parameters: none
This strategy simply combines the `simplify`, `assert` and `flatten` PVS commands, which are often used one after the other.

`split-all` Parameters: (fnum nil), (sko? t)
The functionality of this strategy is the same as if `split-if`, `reduce-if`, `replace*` and `simp` were repeatedly called. All consequent *if-then-else* formulas are split, as well as any in *fnum*. That is, `split-all` will split only consequent *if-then-else* formulas (and those antecedents simplified by `reduce-if`) while `split-all -` splits all *if-then-else* formulas. Finally, strategy `clean-up` is called to remove extraneous formulas.

`split-all-inst` : Parameters values, (inst-into -), (split-up 0), (split-all -), (remove-impl? t), (hide? t)
Applies `split-up` to split-up, and then repeatedly tries `(split-all split-all)` (that is, splits all consequent if-then-else formulas, and any in parameter split-all) and `(inst-set inst-into values)` until no further simplification can be done, and no further quantifications appear in formulas inst-into.

If hide? is t (default), then the uninstantiated version of formulas which have been instantiated are labeled '_inst_p" and hidden.

By default, remove-impl? is set to t, and so `(remove-impl)` is called, converting implications and iff-statements to if-then-else statements that can be split by split-all.

For this rule to succeed all the quantifiers in the inst-into formulas must be of the same type.

This rule is typically useful when there is a small set of variables in one domain which are being used. It is often used after `split-rho` (below) to complete a proof branch. It tends to be time-consuming.

`split-and` Parameters: (fnum +)
Splits the consequent conjunction a *and* b at line fnum generating two proof branches, one with antecedent a and consequent a *and* b, the other with consequent a.

`split-if` Parameters: (fnum +)
Chooses an *if-then-else* formula to be split generating one or two subgoals. It also calls `replace*` and `simp` on the subgoals. If fnum is not set, the first consequent *if-then-else* is split.

`split-impl` Parameters: (fnum -)
Splits the antecedent implication *a implies b* at line fnum generating two proof branches, one with *a and b*, the other with *not a*.

`split-or` Parameters: (fnum -)
Splits the antecedent disjunction *a or b* at line fnum generating two proof branches, one with *a*, the other with *not a and b*.

`split-up` Parameters: fnum
Repeatedly applies `spsi` to the formula fnum. Parameter fnum should ideally be a label rather than a line number as during the splitting the position of the original formula, and the formulas it has been split into, may move.

`spsi` Parameters: (fnum +)
Formula at line number fnum is `split` and `simp` applied to all branches. If no fnum is specified, a consequent is split (fnum defaults to +).

`try-grind, try-lazygrind, try-split-all` : Parameters none
Call the `grind, lazygrind` and `split-all` strategies. However, if they do not close this goal the changes are undone.

Comments

Of these, by far the most useful are `simp, spsi`, `split-all`, `reduce-all` and `clean-up`. Strategy `split-all-inst` can also be very useful though its usage is limited when many domains are used.

The `split-all` strategy was designed to address points 1, 2 and 3 of subsection 5.2, while `reduce-all` and `clean-up` are specific to points 2 and 3 respectively.

In most of our proofs `split-all` is the strategy that does most of the "work". When the proof is near completion it is often preferable to use `ground` or `grind :if-match nil`, which are more efficient. However, when they do not close the proof branch, these strategies often give very numerous, very similar subgoals, while `split-all` which has more limited splitting conditions, gives fewer subgoals. The `grind` strategy also expands all definitions: while this can be very useful in very simple proofs, in proofs with complex data structures it can be counter-productive.

The issue of point 4 can be addressed by using the `split-impl` or `split-or` strategies. Calling `remove-impl` and then `split-all -` splits all implications.

5.4 Strategies Tailored to the Temporal Logic Theories

In this Section we define strategies that have been developed for the temporal logic framework, and which are not applicable in other frameworks. These strategies refer to structures and naming conventions used by the temporal logic theories, and are thus unusable in other frameworks.

We list a number of naming conventions:

1. The parameterized fair system, PFS, is *pfs*.
2. The transition relation is defined as a variable *rho* (it is also possible to omit giving it a name, defining it immediately as a field of the PFS).
3. The justice set in the PFS is named *justice* (or has no name).
4. The inductive invariant of a system is named *reachable* and the lemma *reachableInv* proves its invariance. (Should numerous inductive invariants be proved for one system, their conjunction should be named *reachable*).
5. The ranking function is termed *rank*.
6. The assertion sets and sets of justice requirements given to a rule should be names from the group *hset, dset, jset, help, just* and *phi*.
7. If the transition domain has processor id and location fields then these are called *pid* and *loc*, respectively.

In addition to these conventions, the strategies themselves introduce labeling and naming conventions:

- The current state is *current!1* and the next state *next!1*. These are renamings of elements in the state sequence, typically *seq(t)* and *seq(t+1)*. The sequents with the equivalences (*seq!1(t!1) = current!1*) are labeled as *current* and *next* and are hidden. If the state sequence formula is required, it can be revealed, e.g. `(reveal current)`.
- The reachability invariant reachable is labeled *reachable*.
- The transition relation is labeled *rho*.
- The fair computation requirements are labeled *computation* and are hidden.
- A formula stating that the rank in the next state equals that in the current state is labeled *same*, where applicable, while a formula stating that the rank decreases is typically labeled *dec*.
- Some strategies label the sequent which we are “required to prove” as *rtp*.

5.4.1 Strategies Invoking Proof Rules

Note: In every case the strategy cannot be used unless the corresponding PVS theories in the prelude have been imported. Most of the theories will have been imported with theory PFS, however, the theories for response rules need to be imported separately.

`binv` Parameters: (inv1, inv2, inv3, inv4: nil)
Applies lemma BINV. Tries to use lazygrind to show that the proposed invariant holds at the initial state. Does some simplification and labeling. The formulas inv1, inv2, inv3 and inv4 are all expanded.

`bwait` Parameters: (inv1, inv2, inv3, inv4: nil), (split-rho? f)
Applies lemma BWAIT. Does some simplification and labeling. The formulas inv1, inv2, inv3 and inv4 are all expanded. If split-rho? is t then strategy `split-rho-all` is applied.

`coni` Parameters: subst1 subst2 (hide? f)
Applies lemma CONI. The parameters subst1 and subst2 are the line number of the formulas whose conjunction is to be taken. These formulas are then hidden if hide? is set to t.

`coni-m` Parameters: number (hide? f)
Repeatedly applies lemma CONI to the first number antecedents, so that they form one conjunction. These formulas are then hidden if hide? is set to t.

`dist_rank` Parameters: (inv1, inv2, inv3, inv4: nil), (split-rho? f)
Applies rule DIST-RANK. Brings in the reachability invariant. Does some simplification and labeling. The formulas inv1, inv2, inv3 and inv4 are all expanded.

If split-rho? is set to true (t) then the transition relation is split (using `split-rho`, described below).

`dist_rank_infinite` Parameters: (inv1, inv2, inv3, inv4: nil), (split-rho? f)
Applies rule DIST-RANK-INFINITE. Brings in the reachability invariant. Does some simplification and labeling. The formulas inv1, inv2, inv3 and inv4 are all expanded. E.g. `(dist_rank "rank_func")` will expand "rank_func" and apply the DIST_RANK lemma.

If split-rho? is set to true (t) then the transition relation is split (using `split-rho`, described below).

`dist_rank_simple` Parameters: (inv1, inv2, inv3, inv4: nil), (split-rho? f)
Applies rule DIST-RANK-SIMPLE. Brings in the reachability invariant. Does some simplification and labeling. The formulas inv1, inv2, inv3 and inv4 are all expanded.

If split-rho? is set to true (t) then the transition relation is split (using `split-rho`, described below).

`inv` Parameters: weaker, stronger, (inv1, inv2 : nil)
Applies lemma INV in order to prove the property "weaker". The property "stronger" should

be an inductive property implying "weaker". Tries to use lazygrind to show that "stronger" holds at the base case and that "stronger" implies "weaker". Does some simplification and labeling. The formulas stronger, weaker, inv1 and inv2 are all expanded.

`moni` Parameters: none
Applies lemma MONI. Does some simplification and labeling.

`preorder` Parameters: (inv1, inv2, inv3, inv4: nil), (split-rho? f)
Applies rule PRE-ORDER. Brings in the reachability invariant. Does some simplification and labeling. The formula inv1, inv2, inv3 and inv4 are all expanded. If split-rho? is set to true (t) then the transition relation is split (using `split-rho`, described below).

It is assumed that the preorder is named *preorder*, *pre_order* or *preord*.

`state-valid-to-valid` Parameters: (fnum -) (hide? f)
Converts formulas of the form `is_P_state_valid`(a, `pfs`)to `is_P_valid`($G(a)$, `pfs`). The parameter fnum gives the line number of the formula to be converted. If hide? is t then the formula at line fnum is hidden provided that fnum was given as a negative integer.

`valid-to-state-valid` Parameters: (fnum -) (hide? f)
Converts formulas of the form `is_P_valid`($G(a)$, `pfs`) to `is_P_state_valid`(a, `pfs`). The parameter fnum gives the line number of the formula to be converted. If hide? is t then the formula at line fnum is hidden provided that fnum was given as a negative integer.

`well` Parameters: (inv1, inv2, inv3, inv4: nil), (split-rho? f)
Applies rule WELL. Brings in the reachability invariant. Does some simplification and labeling. The formula inv1, inv2, inv3 and inv4 are all expanded. E.g. `(well "waiting")` will expand "waiting" and apply the WELL lemma.

If split-rho? is set to true (t) then the transition relation is split (using `split-rho`, described below).

`well_simple` Parameters: (inv1, inv2, inv3, inv4: nil), (split-rho? f)
Applies rule WELL-SIMPLE. Brings in the reachability invariant. Does some simplification and labeling. The formula inv1, inv2, inv3 and inv4 are all expanded. If split-rho? is set to true (t) then the transition relation is split (using `split-rho`, described below).

5.4.2 Other Strategies

`exptemp` Parameters: none
Expands many of the definitions of the temporal logic theories, such as is_P_valid, F, X etc.

`reach` Parameters: none
Invokes the reachability invariant using lemma "reachableInv".

`split-rho` Parameters: (remove-impl? t)(split-up 0)
Splits rho, replaces and simplifies. If remove-impl? is t, then strategy `remove-impl` is applied. Strategy `split-up` is applied to formula split-up.

This is the simplest strategy for splitting the transition relation.

`split-rho-all` Parameters: (values nil), (split-up 1), (inst-into -), (force? f), (remove-impl? t), (no-sub? f), (split-all -) (hide? t)
Calls `split-rho` to split the transition relation. An attempt is then made to complete each of the subgoals generated by using `split-all` on the consequents and formulas split-all. If values is not empty then the values in values are substituted into formulas inst-into (using `(split-all-inst values inst-into)`).

If remove-impl? is t, then `remove-impl` is called to convert implications and iff-statements to if-then-else statements.

The number of subgoals generated by `split-all` (or `split-all-inst`) can be controlled using the force? and no-sub? parameters.

By default, if applying `split-all` (or `split-all-inst`) to a rho-subgoal generates more than one subgoal it is undone. If force? is set to t then the strategies are not undone irrespective of how many subgoals are generated. If no-sub? is t then the strategies are undone if even one subgoal remains – i.e. under this option if `split-all` (`split-all-inst`) does not complete the proof for this goal, it is undone.

If hide? is t (default), then the uninstantiated version of formulas which have been instantiated are labeled '_inst_p" and hidden.

This strategy is slow running, particularly when values is not nil. It can be speeded up a little by setting split-all to +, in which case it may return more subgoals.

In large proofs it is often preferable to apply `split-rho` and then solve each subgoal, possibly using `split-all` or `split-all-inst`.

`use-invariant` Parameters: inv (where current)(expand t)
Invokes lemma inv and applies `exptemp`. Formula where is revealed (if hidden) and the lemma is instantiated in accordance with it. The default is to instantiate so that the invariant applies to the current state. If expand is t then inv is expanded (assuming that it is also the name of a formula).

Comments

The suggested approach is to start by choosing the appropriate strategy invoking proof rules (Section 5.4.1).

The "most difficult" subgoal generated by a rule strategy almost always involves the transition relation (e.g proving that there is some helpful transition at the next state, or that the ranking function has decreased in some way). On this branch we advise invoking `split-rho` or `split-rho-all`. While `split-rho-all` will often complete a number of the branches generated by `split-rho`, it may take too long to run. The choice of whether to us `split-rho-all` or `split-rho` depends on the size of the formulas present, and the willingness of the user to wait for PVS to complete running a strategy.

If the consequents generated by `split-rho` or `split-rho-all` contain no existential quantifiers then the new subgoals can often be completed or simplified by using `split-all` or `split-all-inst`. Where existential quantifiers do exist in the consequents it often desirable to split any such consequent so that the quantifier is outermost. In many cases it is then possible to instantiate the consequent before running the simplification.

Where there are a number of complex consequents it speeds up the proof to analyze which will hold in which cases and to hide the others (to prevent unnecessary splitting on these consequents and the resulting increase in time and subgoals.) For example, after applying the WELL rule one might be able to easily determine in which cases the same transition will remain helpful versus the cases in which the rank decreases. By splitting the proof according to this condition and hiding the irrelevant option on each branch the proof effort is greatly speeded up. Such splitting can be done before `split-rho` or after it. The earlier the splitting occurs the greater the efficiency saving but the more analysis is required. We note that such analysis and splitting, while recommended, is not necessary to complete the proof.

Appendix A

A Very Brief Introduction to PVS

The PVS system is an extensive, well-documented, deductive verification system. It is impossible to summarize all its features in a short document. The purpose of this document is to briefly introduce the user to the PVS system and some of the more frequently used commands. The full PVS documentation is available at ~verify/PVS2-4.1/Doc/ .

In Section A.1 we give an overview of the sequent calculus which forms the theoretical basis for the PVS proof system. In Section A.2 we define some basic concepts in the PVS system, in Section A.3 overview the specification language, and in Section A.4 explain how to run PVS. Section A.5 details some useful prover features. Sections A.6 and A.7 summarize the most frequently used system and prover commands, respectively.

A.1 The Logic of PVS

In this section we examine the theoretical basis underlying the PVS logic.

The PVS prover maintains a *proof tree*, and it is the goal of the user to construct a proof tree which is complete, in the sense that all the leaves are recognized as true. Each node of the proof tree is a *proof goal*, from which its offspring nodes follow by a *proof step*.

Each proof goal is a *sequent* consisting of a sequence of *antecedent* formulas, and a sequence of *consequent* formulas. Together, the antecedents and the consequents are called the *sequent formulas*.

We will let Γ and Δ represent (finite) sequences of formulas, and p q, p_i and q_i, represent individual formulas.

Sequents are represented in the form $p_1, p_2, \ldots, p_n \vdash q_1, q_2, \ldots, q_m$, where the formulas preceding the turnstile are the antecedents, and the formulas after the turnstile are the consequents. The intuitive interpretation of a sequent is that

$$\forall free : (p_1 \wedge p_2 \wedge \ldots \wedge p_n) \rightarrow (q_1 \vee q_2 \vee \ldots \vee q_m)$$

where *free* denotes the free (unbound) variables.

The system uses *backwards reasoning.* That is, each proof step results in sequent(s) that are at least as strong as the previous one. The root of the tree is the sequent $\vdash q$, where q is the theorem to be proved.

A.1.1 Axioms

Three axioms are used to recognize leaf sequents as true[1]. They are:

$$\begin{array}{lllll} \text{A1}: & \Gamma, p & \vdash & \Delta, p \\ \text{A2}: & \Gamma & \vdash & \Delta, \text{T} \\ \text{A3}: & \Gamma, \text{F} & \vdash & \Delta \end{array}$$

That is, A1 asserts that a sequent is true if any antecedent is the same as any consequent. (Clearly, it is always the case that $\Gamma \wedge p \rightarrow \Delta \vee p$). Axiom A2 states that a sequent is true if any of its consequents are true and axiom A3 that a sequent is true if any of its antecedents are false.

A.1.2 Proof Rules

In this section we present some of the proof rules used to add subtrees to the proof tree.

Propositional Rules

We consider rules for the basic operators of conjunction, disjunction and negation, noting that implications can be converted to these operators.

We first present four *non-expanding* propositional rules. These are rules in which a single sequent is derived from a parent[2].

$$\frac{\Gamma \vdash \Delta, p \vee q}{\Gamma \vdash \Delta, p, q} \qquad \frac{\Gamma, p \wedge q \vdash \Delta}{\Gamma, p, q \vdash \Delta}$$

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

The first two rules allow consequent disjuncts and antecedent conjuncts to be flattened, so that two formulas are obtained from one.

The second two rules allow negated antecedent or consequent formulas to become non-negated consequent or antecedent formulas, respectively.

We next present two *expanding* propositional rules[3]. These rules derive a subtree of two sequents.

$$\frac{\Gamma, p \vee q \vdash \Delta}{\Gamma, p \vdash \Delta \qquad \Gamma, q \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, p \wedge q}{\Gamma \vdash \Delta, p \qquad \Gamma \vdash \Delta, q}$$

That is, the first rule states that in order to prove that $\Gamma \wedge (p \vee q) \rightarrow \Delta$ it is sufficient to prove both that $\Gamma \wedge p \rightarrow \Delta$ and that $\Gamma \wedge q \rightarrow \Delta$.

[1]The application of these three axioms corresponds roughly to the PVS `assert` command.
[2]The non-expanding propositional rules correspond to the PVS `flatten` command.
[3]The expanding propositional rules correspond to the `split` command.

Quantifier Rules

The notation $p\{x \leftarrow t\}$ represents the result of substituting the term t for all free occurrences of x in p.

The following two rules represent the process of *skolemization*[4]. They require that t be a new constant that does not occur in the sequent.

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

The next two rules represent the effect of *instantiation*[5].(There is no requirement that t be a new constant.)

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

Strengthening Rules

The above rules neither strengthened or weakened the sequent. We present two rules which allow a stronger sequent to be derived from a weaker one by removing formulas[6].

$$\frac{\Gamma, p \vdash \Delta}{\Gamma \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta, p}{\Gamma \vdash \Delta}$$

A.2 The PVS Verification System : Basic Definitions

Specification files are ordinary text files, written in the PVS specification language, generally prepared using the PVS emacs editor interface. They have a `.pvs` extension. All such files must have the structure

```
theory-name: THEORY
 BEGIN

 END theory-name
```

Where theory-name is the name of the theory, and should preferably match the file name. The body of the theory is placed between the begin and end statements.

Proof files (`.prf` extension) save proofs that have been composed. They are also text files, but it is not advisable to try to edit them in any way.

A *context* is a set of specification and proof files found in one directory. The binary `.pvscontext` file saves the state of the verification from one verification attempt to another. For this reason, all files relating to one proof should be in one directory. Files relating to unrelated proofs are best kept in a separate directory with a separate context.

The PVS *interface* is through an emacs editor. In different buffers in the editor one can write specifications and run proofs. PVS commands are entered in the emacs mini-buffer window and are preceded by `M-x` (alt-x or esc-x).

[4]Skolemization is effected in PVS by `skolem!` and related commands.

[5]The form of instantiation presented here is equivalent to the `inst-cp` command.

[6]The PVS `delete` and `hide` commands explicitly delete and hide formulas, respectively. Other commands, such as `inst`, may also hide formulas.

A.3 The Specification Language

The PVS specification is built on higher order logic.

Variables and constants have types – some types are inbuilt, and the user can build their own (including arrays, records, etc). PVS also allows the user to use uninterpreted types.

Frequently used inbuilt types include `nat` , `int` , `real` , `bool` .

Constants are declared as being elements of a type e.g.

```
z : nat
```

declares constant z of type natural.

To declare a *variable* one must simply precede the type with the `VAR` keyword e.g.

```
x : VAR nat
```

There are many different possibilities in declaring types. One of the most basic, and most useful, are *tuples* e.g.

```
T1 : TYPE = [nat, nat]
```

Records are tuples with labeled fields e.g.

```
T2 : TYPE = [# first, second : nat #]
```

Both T1 and T2 contain two natural numbers. In the first case the fields are unlabeled, in the second they are labeled.

Another useful type is a *function* type. This is a mapping from a domain to a range. e.g.

```
A1 : TYPE = [upto[10] -> T1]
```

is an array A1[0..10] of elements of type [nat, nat].

The basic logical constructs are `OR` (also written \/), `AND` (also &, /\), `NOT` (~), =, \= (disequality), `IMPLIES` (=>) and `IFF` (<=>). The universal and existential quantifiers are `FORALL` and `EXISTS` , respectively.

A.3.1 Example - Hotel Reservations

As a very simple example we consider a hotel reservation system. For every room, for every date, the name of the person reserving the room is stored in the register. If a room is free, then it is recorded simply that it is registered to the constant name `free` .

We define this as

```
reservation: THEORY
 BEGIN

  room: TYPE
  date: TYPE
  name: TYPE
  free: name
  reservations: TYPE = [room, date → name]

 END reservation
```

The room, date and name types are all uninterpreted. The reservations type is a mapping from room and date to name. `free` is a constant of type name.

It is also useful to declare interpreted functions. For example, we can define a function which given a register, adds a booking and returns the updated register:

```
reserve(r: room, d: date, n: name, register: reservations): reservations =
    register WITH [(r, d) := n]
```

This demonstrates the use of the very important `WITH` expression. `WITH` is an override expression used to modify the contents of a function, tuple, or record. `register WITH [(r, d) := n]` is the same as `register` except that the argument at value (r, d) is replaced with n.

Instead of declaring the types when using variables (e.g. r, n, d above) we can define them as global variables once, and then use them without stating the types every time. Thus, we define the function cancel which cancels the reservation for a room, and the predicate reserved which returns the true if a room is reserved.

```
r: VAR room
d: VAR date
n: VAR name
register: VAR reservations

cancel(r, d, register): reservations = register WITH [(r, d) := free]

reserved(r, d, register): bool = register(r, d) ≠ free
```

A.4 Running PVS

To run PVS make and enter the directory you plan to work from. You can then open PVS by calling the executables at ~verify/PVS2-4.1/pvs for sun machines, or ~verify/PVS2-4.1/pvs_linux for linux machines.

This should open up an emacs window. You will be asked whether to create a new context - answer yes.

We will continue with the reservation example.

Open the desired file (emacs command `C-x, C-f`).

It is important to *typecheck* the file. Typechecking parses the file, and checks for semantic errors (e.g. undeclared names). We typecheck the file by typing `M-x tc` . PVS reports that one type correctness condition, TCC, was generated, and that it is unproved. To view the TCCs type `M-x show-tccs` . In this case the typecheck is trivial, and PVS can discharge it if you ask it to prove the typechecks – `M-x tcp` . When PVS cannot discharge the typecheck automatically, it must be done manually. This can be done by placing the cursor on the TCC definition in the buffer generated by `M-x show-tccs` , and proving it as you would any other formula.

We prove a simple lemma : if a reservation for a room is canceled then the room is not reserved:

```
canceled_not_reserved: LEMMA
  ∀ r, d, register: ¬ reserved(r, d, cancel(r, d, register))
```

To prove this lemma put the cursor on the definition of the lemma and type `M-x pr` .

You will get a new buffer, labeled *pvs*, containing the sequent:

```
canceled_not_reserved :
```

```
  |-------
{1}   FORALL r, d, register: NOT reserved(r, d, cancel(r, d, register))
```

PVS presents sequents as a list of negatively numbered antecedents above above a turnstile symbol `|-------`, and a list of positively numbered consequent formulas below it.

The `Rule?` prompt indicates that PVS is waiting for a new command to be entered. Prover commands use lisp-like syntax, and are always enclosed in round brackets.

We can *skolemize* and simplify by typing `(skosimp*)`. This removes quantified variables and replaces them by skolem constants. Skolem constants have "!1" added to the variable name.

We can now expand out the definitions of reserved and cancel by entering `(expand "reserved")` and then `(expand "cancel")`. The proof is complete.

In fact, the `(grind)` command can complete this proof in one step. `(grind)` expands out expressions, skolemizes, instantiates and simplifies. It is often useful in very small proofs, or towards the ends of large proofs. However, when it does not complete a proof, it can be counterproductive, generating many similar subgoals.

We consider a second lemma,

```
is_reserved: LEMMA
  ∀ r, d, n, register: reserved(r, d, reserve(r, d, n, register))
```

The lemma is intended to show that after reserving, a room is reserved. The reader is invited to try to prove it. It will soon be apparent that this cannot be done – in fact the premise is false. The reader is encouraged to try to understand why this is the case and how it could be rectified.

A.4.1 Example : Queues

As a second example, we consider a simple queue structure modeled as an array[nat] of entries. The queue has a head pointer, pointing to its oldest element. It also has a size field, indicating how many entries are currently occupied. The occupied entries are stored in an array entries, from positions head to head + size - 1, inclusive.

That is, consider a queue with 2 entries, A and B, and its head at position 4. entry[4] = A and entry[5] = B. The queue size is 2. The contents of entry[x] is irrelevant for any $x \neq 4, 5$.

A few notes on the data structures :

- `QUEUE_ENTRY` is an uninterpreted type. We know nothing about it.
- `QUEUE_TYPE` is a record with three fields, `head`, `size` and `entry`.

  There are two means of accessing record fields – using a ‘, and using brackets. So, `queue‘size` and `size(queue)` both return the size field of record queue.
- `pushQentry`, `empty`, `popQentry`, `occupied` and `inQ` are all functions.

  Functions cannot modify their arguments.

  Procedures are functions that return a boolean value.

We examine how the lemmas can be proved.

We start with `empty_no_occupied` . This lemma states that a queue is empty if and only if none of its buffers are occupied.

We first skolemize `(skosimp*)` to remove the quantification. We then expand out the terms : `(expand "empty")` , `(expand "occupied")` , arriving at the sequent

```
  |-------
{1}   size(queue!1) = 0 IFF
       (FORALL qPoint:
          NOT (qPoint >= queue!1`head AND
                qPoint < queue!1`head + queue!1`size))
```

We use the `(split)` command to split the `IFF` statement into its two directions, generating two subgoals, one for the "if", and one for the "only if".

You can view the second subgoal by typing `(postpone )`, or Tab Shift-p.

The first subgoal,

```
  |-------
{1}   size(queue!1) = 0 IMPLIES
       (FORALL qPoint:
          NOT (qPoint >= queue!1`head AND
                qPoint < queue!1`head + queue!1`size))
```

can be flattened `(flatten)` , into a sequent where `size(queue!1) = 0` is assumed:

```
{-1}  size(queue!1) = 0
  |-------
{1}   FORALL qPoint:
        NOT (qPoint >= queue!1`head AND
              qPoint < queue!1`head + queue!1`size)
```

After skolemizing `(skosimp*)` , an `(assert)` completes the proof.

PVS now returns to the second goal. We again flatten it, generating the sequent

```
{-1}  FORALL qPoint:
        NOT (qPoint >= queue!1`head AND
              qPoint < queue!1`head + queue!1`size)
  |-------
{1}   size(queue!1) = 0
```

The universal quantifier must be instantiated, and we ask PVS to do so by typing `(inst?)` . Unfortunately, PVS guesses the instantiation incorrectly, so we must instantiate manually.

We undo the instantiation by typing `(undo)` or Tab u. The correct instantiation is queue!1`head. (The antecedent -1 asserts that it is NOT the case that qPoint >= queue!1`head and qPoint < queue!1`head + queue!1`size. In other words, qPoint < queue!1`head or qPoint >= queue!1`head + queue!1`size. Setting qPoint = queue!1`head therefore implies that queue!1`size = 0.)

We instantiate with the command `(inst - "queue!1`head")` . The first parameter to the `inst` command is the sequent number, the second the value to be instantiated. We could have used "-1" as the sequent number. By typing "-" we tell PVS to substitute into the first matching antecedent. In general, this is more robust than specifying exact line numbers

(should the specification or proof later be changed, the line numbers may change and the proof might no longer work if sequent numbers are fully specified.)

The proof is now completed with `(assert)` .

We consider the second lemma, `pushed_entry_in_queue` . We first try to complete it using `grind` . Unfortunately, this does not work – `grind` instantiates the existential quantifier incorrectly.

We try `grind` without instantiation by typing `(grind :if-match nil)` . The parameter :if-match determines under which conditions `grind` will do instantiations. By setting it to nil, we prevent `grind` from instantiating.

The resulting formula has all the expressions expanded. (The same effect could have been obtained using the `expand` command).

```
  |-------
{1}   EXISTS (index: nat):
        (index >= queue!1`head AND index < 1 + queue!1`head + queue!1`size)
         AND
         entry(queue!1) WITH [(head(queue!1) + size(queue!1)) := qEntry!1]
             (index)
          = qEntry!1
```

The correct instantiation is `(inst + "queue!1`head + queue!1`size")` . The assert command will now complete the proof.

The third lemma is left as an exercise.

```
queue: THEORY
 BEGIN

  QUEUE_ENTRY: TYPE

  QUEUE_TYPE: TYPE = [# head: nat, size: nat, entry: [nat → QUEUE_ENTRY] #]

  queue: VAR QUEUE_TYPE
  qEntry: VAR QUEUE_ENTRY
  qPoint: VAR nat

  % Returns true if a queue is empty
  empty(queue): bool = size(queue) = 0

  % Pushes an entry onto a queue and returns the updated queue
  pushQentry(queue, qEntry): QUEUE_TYPE =
      queue WITH [size := queue`size + 1,
                  entry := entry(queue) WITH [(head(queue) + size(queue)) := qEntry]]

  % Pops an entry from a queue. It returns the new queue.
  popQentry(queue): QUEUE_TYPE =
      IF ¬ empty(queue)
        THEN queue WITH [head := head(queue) + 1, size := size(queue) − 1]
      ELSE queue
      ENDIF

  % Returns true if entry qPoint of the queue is occupied,
  % i.e. lies between head and head + size
  occupied(queue, qPoint): bool =
      qPoint ≥ queue`head ∧ qPoint < queue`head + queue`size

  % Checks whether there is an occupied entry in the queue with value
  inQ(queue, qEntry): bool =
      ∃ (index: nat): occupied(queue, index) ∧ queue`entry(index) = qEntry

  % If a queue is empty, no entries are occupied
  empty_no_occupied: LEMMA
    ∀ queue: empty(queue) IFF (∀ qPoint: ¬ occupied(queue, qPoint))

  % After pushing an entry onto a queue, the entry is in the queue
  pushed_entry_in_queue: LEMMA
    ∀ queue, qEntry: inQ(pushQentry(queue, qEntry), qEntry)

  % If one pushes an entry onto an empty queue, and then pops one off,
  % the queue is again empty.
  empty_push_pop_empty: LEMMA
    ∀ queue, qEntry: empty(queue) ⊃ empty(popQentry(pushQentry(queue, qEntry)))

 END queue
```

Figure A.1: Theory queue.pvs

A.5 Some Prover Features

In this Section we will discuss some useful features of the PVS environment.

A.5.1 Stepping Through a Proof

PVS allows you to walk through, or step through, a proof, viewing the effects of each command. The step through command `M-x step-proof` initiates a proof from the beginning, and also opens a Proof buffer in which the last saved proof is displayed.

The first command in the buffer is highlighted. Typing Tab 1 will cause it to be executed. Typing Tab n, for number n, will cause the next n commands to be implemented. If you put the cursor in the Proof buffer, Tab 1 will execute the next command after the cursor, and not the highlighted command.

To break a continuous execution generated by Tab n, type C-g. This will complete the currently executed command and highlight the next one.

At any point, you can enter commands at the Rule? prompt rather than from the proof buffer.

A.5.2 X-Displaying a Proof

One can generate an X-display of a proof. This is a window graphically showing the current proof as a tree. Every sequent in the proof is represented by a $\vdash$ symbol. The root, at the top, is the initial sequent. The proof commands used to create the proof are shown between the $\vdash$ symbols. To see the full text of a sequent click on the $\vdash$ symbol.

Status information about the proof is indicated by colors: blue indicates a completed branch, brown the current branch, purple the current sequent, and black any incomplete branch which is not the current branch.

The proof tree is automatically updated as the proof is run.

To initiate a proof with its X-display enter `M-x xpr` when the cursor is on the formula. Similarly, step through a proof with its X-display using `M-x x-step-proof`.

To open an X-display for a proof already in progress one must use the main PVS menu. From the PVS menu go to display-commands then show-current-proof.

Warning: The X-display slows the proof down significantly. Furthermore, trying to access the display (either to start it, or to view sequents etc) while PVS is running (i.e. (ILISP : run) displayed on the *pvs* buffer label line, rather than (ILISP : ready)) can cause PVS to be interrupted. The result of this is that PVS must be reset, and the proof restarted!

A.5.3 Hiding and Revealing Formulas

When a formula is instantiated, its uninstantiated copy is "hidden" by PVS. Hidden formulas can be viewed by typing `M-x show-hidden-formulas`.

The formula can then be revealed by typing `(reveal fnum)` where fnum is the sequent number of the desired formula.

Formulas which are not needed can be hidden by typing `(hide fnum)`.

(Both the hide and reveal commands can take lists of formulas).

A.5.4 Strategies

The PVS user can construct strategies which combine proof rules together into more powerful proof rules. Strategies are written in a lisp based language, and can include recursion, branching and backtracking. (C.f. Chapter 5 of *PVS Prover Guide*).

At this point we will discuss only two basic, and useful, constructs: `then` and `repeat` .

The then keyword is used in the format (`then` step1, rest-steps). Step1 is applied to the current goal and then rest-steps to each of the sequents generated. For example, `(then (split -1)(assert))` will split formula -1 and then apply `assert` to each of the sequents generated.

Iteration can be effected by using (`repeat*` step). Rule step is replied iteratively along all subgoals until its application has no effect e.g. `(repeat* split)` would repeatedly split until there is nothing more to split.

A.6 Summary of Some Often-Used System Commands

We first list some PVS system commands. These commands are generally entered while the .pvs specification is the current buffer. Command relating to a single formula require that the cursor be on the formula. More information can be found in the *PVS System Guide*.

Command	Alias	Comments
Exiting and Interrupting PVS		
M-x exit-pvs	C-x C-c	Exit PVS
	C-c C-c	Interrupt PVS process (useful if PVS is taking unreasonably long). Entering `(restore)` allows you to resume the proof.
M-x reset-pvs	C-z C-g	Abort PVS and resynchronize. You will have to restart the proof from the beginning
Initiating and Stepping Through Proofs		
M-x prove, M-x pr	C-c p	Prove formula pointed to by cursor
M-x step-proof	C-c C-p	Step through an existing poof. (C.f. Section A.5.1)
M-x x-prove	M-x xpr	Prove with X-display. (C.f. Section A.5.2)
M-x x-step-proof		Step through with X-Display
M-x prove-theory	M-x prt	Reruns all proofs in the theory (non-interactively).
Typechecking		
M-x typecheck	M-x tc	Typecheck theories in current buffer
M-x typecheck-prove	M-x tcp	Typecheck theories and try to prove TCCs
M-x show-tccs	M-x tccs	Show the TCCs of the current theories. TCCs can be proved from the new buffer created
Editing and Viewing Proofs		
M-x edit-proof		Edit / view the proof of the formula
M-x install-proof	C-c C-i	Installs a proof on a formula The format must be like that obtained when editing proofs, not the format in the .prf file. You can highlight a proof buffer of one formula and install it on a second
Proof Status		
M-x status-proof	M-x sp	Status of formula at cursor. proved – complete = fully proved, proved – incomplete = formula is proved but depends on some unproved formula unchecked = changes since the proof succeeded may invalidate it. You should rerun the proof (M-x pr). untried = proof never attempted unfinished = proof attempted, but never completed.
M-x status-proof-theory	M-x spt	Status of all formulas in theory
M-x status-proof-importchain	M-x spi	Status of formulas on importchain
M-x status-proofchain	M-x spc	Displays proofchain of formula at cursor i.e. lists the formulas on which it is dependent
Context and Prelude Commands		
M-x change context	M-x cc	Switch to a new context (new directory)
M-x load-prelude-library		Allows the current context to use all theories in the loaded context (directory).

A.7 Summary of Some Often-Used Prover Commands

This section lists a selection of the more frequently used prover commands. More information can be obtained from the *PVS Prover Guide*.

Note that PVS is case insensitive regarding prover commands i.e. you may type the command using any combination of upper and lower case letters.

A.7.1 Parameters

Proof commands take a list of zero or more required and optional parameters. Each optional parameter has an associated default value. In this section optional parameters are bracketed with their default values. Required parameters are not bracketed.

We have not listed here all parameters to the commands, but only those that are frequently used. When we have omitted some intermediate parameters, this is node by "..." in the parameter listing.

When invoking a proof command actuals are associated with formal parameters according to the order in which they are given. If there are fewer actuals than formals, then those parameters for which no actual was provided are bound with their default values. To give parameter values out of order, or without specifying a previous parameter in the list, one must name the formal when giving the actual.

For example, consider the proof command
`replace` Parameters : fnum, (fnums *), (dir LR), (hide? nil)

This command takes the formula at line fnum (it must be an equality) and replaces it in formulas fnums. The replacement is in direction LR (left to right), and the formula is hidden if hide? is set to t.

Consider, for example, a sequent with an antecedent
`[-3] i!1 = j!1`

Command `(replace -3)` will replace i!1 with j!1 in all sequents.

Command `(replace -3 (2, 3) :hide?  t)` will replace i!1 with j!1 only in sequents 2 and 3, and then hide -3.

Command `(replace -3 :dir rl :fnums +)` will replace j!1 with i!1 in all consequents (reversing the direction of replacement).

A.7.2 Formula and Truth Value Selection

Many commands have arguments taking the number(s) of the sequent formulas where the rule is to be applied. By convention, a single formula is called "fnum" (for "sequent formula number"), a list is "fnums".

The list of antecedent sequent formulas can be indicated by "-", the list of consequent sequent formulas by "+", and the list of all sequent formulas by "*".

Where truth values are requested, "t" stands for true, "nil" for false.

Command	Parameters	Comments
		Control Rules
quit		Terminates the proof attempt
postpone		Go to next remaining goal. Alias: Tab shift-p
undo	(to 1)	Undoes commands.
		If "to" is a number, n, the last n commands are undone
		If "to" is a proof rule, undoes the proof to the last
		occurrence of this proof rule
		(undo undo) Undoes the undo, if it was the last command executed.
		Alias : Tab u
		Propositional Rules
case	exprs	Introduces case split.
		On one branch exprs is assumed to be true, on another false.
		exprs must be in double-quotes e.g. (case "i!1 = j!1")
split	(fnum *)	Conjunctive splitting.
		Splits one formula to generate 2 subgoals :
		Antecedent $A \vee B$ into antecedent A and antecedent B.
		Antecedent A implies B into antecedent B and consequent A
		Consequent $A \wedge B$ into consequent A and consequent B
		Antecedent $IF(A, B, C)$ into antecedents $A \wedge B$ and $\neg B \wedge C$
		Consequent $IF(A, B, C)$ into consequents A implies B and $\neg A$ implies C
flatten	(fnums *)	Disjunctive Simplifications. Generates 1 subgoal converting :
		Antecedent formula $\neg A$ into consequent A
		Antecedent formula $A \wedge B$ into 2 antecedent formulas, A and B.
		Consequent formula $\neg A$ into antecedent A
		Consequent formula $A \vee B$ into two consequent formulas, A and B.
iff	(fnums *)	Converts boolean equality (=) into equivalence (iff)
lift-if	(fnums *)	Lifts the left-most if or cases statement to the topmost level
		Quantifier Rules
skosimp*		Repeatedly skolemizes then flattens
inst	fnum, terms	Instantiates formula fnum with terms in terms
inst-cp	fnum, terms	Instantiates, retaining a copy of the uninstantiated formula
inst?	(fnums *),...,	PVS chooses the terms to instantiate into formula fnums.
	(copy? nil)	Retains a copy of uninstantiated formula if copy? is t.
		Equality Rules
replace	fnum,	Rewriting using equalities
	(fnums *),	Formula fnum must be of form $l = r$.
	(dir LR),	All occurrences of l in formulas fnums are replaced with r.
	(hide? nil),	Instead, replace r with l if dir RL specified.
		Formula fnum is hidden if hide? is t.
replace*	fnums	Replace formulas in fnums into all formulas.

Using Definitions and Lemmas		
expand	name, (fnum *)	Expands the definition of name in formulas fnum e.g. (expand "empty" 1 -3) expands empty in formulas 1 and -3
expand*	names	Expands all occurrences of all expressions listed in names
lemma	name	Introduces an instance of lemma name. e.g. (lemma "empty_no_occupied") introduces forall queue: ...
Simplification with Decision Procedures		
simplify		Simplifies using decision procedures
assert		More aggressive simplification. (Calls simplify.)
grind	... (if-match t)	Expands all definitions, replaces, simplifies, skolemizes, splits etc. Can be used to automatically complete a proof. Performs instantiation unless if-match is set to nil.
ground		Invokes propositional simplification, splits and asserts.
Making Type Constraints Explicit		
typepred	exprs	If exprs is of type p, antecedent p(exprs) is introduced. E.g. (typepred "head(queue!1)") introduces head(queue!1) >= 0
Structural Rules – Hiding and Revealing Formulas		
hide	fnums	Moves formulas fnums to the Hidden buffer (C.f. Section A.5.3)
hide-all-but	keep-fnums, (fnums *)	Hides all the formulas in fnums except those in keep-fnums
reveal	fnums	Copies formulas fnums from Hidden buffer to sequent.
Annotation Rules – Labeling		
label	string, fnums, (push? nil)	Attaches the label string to formulas fnums If push? is t, then any previous label is retained. Else, the label replaces previous labels
unlabel	(fnums *)	Removes all labels from formulas fnums

Bibliography

[COR$^+$95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995. Available, with specification files, at `http://www.csl.sri.com/wift-tutorial.html`.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[SPBA00] E. Sedletsky, A. Pnueli, and M. Ben-Ari. Formal verification of the ricart-agrawala algorithm. In S. Kapoor and S. Prasad, editors, *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*, volume 1974 of *Lect. Notes in Comp. Sci.*, pages 325–335. Springer-Verlag, 2000.