TLPVS: A PVS-Based LTL Verification System*

Amir Pnueli and Tamarah Arons
{amir,tamarah}@wisdom.weizmann.ac.il

Weizmann Institute of Science, Rehovot, Israel

To Zohar,
For the many years of close friendship and fruitful, most enjoyable, collaboration, and in celebration of the many beautiful things we built together. – – Amir

Abstract. In this paper we present our PVS implementation of a linear temporal logic verification system. The system includes a set of theories defining a temporal logic, a number of proof rules for proving soundness and response properties, and strategies which aid in conducting the proofs. In addition to implementing a framework for existing rules, we have also derived new methods which are particularly useful in a deductive LTL system. A distributed rank rule for the verification of response properties in parameterized systems is presented, and a methodology is detailed for reducing compassion requirements to justice requirements (strong fairness to weak fairness). Special attention has been paid to the verification of unbounded systems – systems in which the number of processes is unbounded – and our verification rules are appropriate to such systems.

1 Introduction

Temporal logic has proved itself a powerful and expressive means for specifying system properties. Of the temporal logics, we have preferred to use the linear temporal logic (LTL) of Manna and Pnueli [6]. Within this specification language we can clearly and succinctly express the soundness and liveness properties which interest us.

Much verification of temporal logic specifications has been performed using symbolic methods and tools such as SMV [7]. When appropriate, symbolic methods can often be used to verify system properties with a remarkable degree of automation. There has been a wealth of research, in recent years, in applying model checking to parameterized and infinite state systems. So far, however,

* This research was supported in part by the John von Neumann Minerva Center for Verification of Reactive Systems, the European Community IST projects "Omega" and "Advance", the Israel Science Foundation (grant 106/02-1), and NSF grant CCR-0205571.

there is no method for uniformly verifying parameterized systems of different types. New systems may require new methods.

Deductive systems, on the other hand, need no special techniques to verify infinite state systems. It is often very easy and natural to formalize a parameterized system in a higher order logic verification tool (such as PVS [8]). The verification of such systems typically entails no significant additional difficulty compared to the verification of a non-parameterized, or finite-state version. However, deductive verification often requires a significant amount of human expertise and interaction to construct the proof.

In this paper we present a new system, TLPVS, built on the PVS verification system, for formal verification of linear temporal logic (LTL) properties of systems. The system includes a formal PVS specification of LTL based on [6] and a framework for defining systems, and proving certain safety and response rules. A number of proof rules are provided in the system – in each case the rule is formally proved, and strategies are provided to support its use. The advantage of using this system over using a general theorem prover is that the included rules and strategies free the user from much of the drudge work.

Our system uses the same logic (LTL) and similar system definitions to STeP [4]. Whereas STeP includes a theorem prover tailored to handle temporal logic and temporal verification, we have chosen to embed temporal logic and its deductive framework within an existing powerful general-purpose high-order theorem prover. In so doing we enable the user to benefit from the abilities of a well developed theorem prover, with a large community of users who continuously develop and improve the prover's power. Basing our system on PVS, and making the theories available, makes it extremely flexible. When necessary users are able to modify or extend the framework to fit their own needs; they are not restricted by the existing set of proof rules and strategies.

In addition to formulating the LTL definitions, and providing support for known rules, we have also developed new rules and methodologies. We developed a new liveness rule for parameterized systems, in which the rank is distributed over a parameterized fairness domain. Given the difficulty of deductively verifying compassion dependent liveness rules, we developed a novel method for converting compassion to justice requirements. Special attention has been paid to the verification of unbounded systems (systems with an unbounded number of processes), and our rules are robust for such systems. We use an unbounded version of Lamport's BAKERY algorithm [3] as a running example, and, in addition, overview a proof of the correctness of Eratosthenes' PRIME-SIEVE algorithm.

The paper is organized as follows: In Section 2 we define the temporal logic theories, giving both the theoretical background and their implementation in our framework. Section 3 details our implementation of BAKERY. used as a running example throughout. In Section 4 we present a number of safety rules, and demonstrate their use. Section 5 focuses on response rules, and Section 6 discusses the reduction of compassion requirements to justice conditions. For reasons of space, we explain neither the PVS specification language, nor the prover commands; we hope that the usage will be clear from the context. Any potential users of our system are encouraged to acquire a basic knowledge of PVS – we recommend the introductory tutorial presented in [1].

2 Temporal Logic Theories

In this section we present our PVS-model of a linear temporal logic and of a parameterized fair discrete system.

2.1 Parameterized Fair Systems

Our system is based on the computational model of *parameterized fair systems*, PFS. This is a variation of the fair discrete systems of [2] which, in turn is derived from the model of *fair transition systems* [6].

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

- Σ : A non-empty set of *system states*.
  Typically, a state is structured as a record whose fields are the typed system variables, V. For a state s and a system variable $v \in V$, we denote by $s`v$ the value assigned to the field v by the state s.
  A *state predicate* (which we often abbreviate as simply "predicate") is a function which maps states into truth values. We also use the notion of a *bi-predicate* which defines a binary relation over states. Syntactically, we present a predicate as an assertion which refers to the system variables.
- Θ : The *initial condition*. A predicate characterizing the initial states.
- $\rho(V, V')$: The *transition relation*. A bi-predicate relating a state to its successor.
- $\mathcal{F}$: A non-empty *fairness domain*. This is a domain which is used to parameterize the fairness requirements of justice and compassion.
- $\mathcal{J}$: The *justice requirement*. This is a mapping from $\mathcal{F}$ to predicates ($\mathcal{J}$: [$\mathcal{F} \mapsto$ predicate]). It characterizes the just states. For every $t \in \mathcal{F}$, a computation must contain infinitely many $\mathcal{J}[t]$-states. When there is no justice requirement associated with $t \in \mathcal{F}$, we use the *trivial* requirement T, which is fulfilled in every run.
- $\mathcal{C}$: The *compassion requirement*. It is a mapping from $\mathcal{F}$ to pairs of predicates: ($\mathcal{C}$: [$\mathcal{F} \mapsto \langle p, q \rangle$]) where p and q are predicates. 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 $t \in \mathcal{F}$, a computation containing infinitely many $\mathcal{C}[t]`p$-states must also contain infinitely many $\mathcal{C}[t]`q$-states. The *trivial* requirement $\langle \mathrm{T}, \mathrm{T} \rangle$ is fulfilled in every run, and is used when no other requirement is needed.

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

The transition relation $\rho(V, V')$ identifies state s_2 as an *S-successor* of s_1 if $\langle s_1, s_2 \rangle \models \rho(V, V')$, where $\langle s_1, s_2 \rangle$ is the interpretation which interprets $x \in V$ as $s_1`x$ and $x'$ as $s_2`x$.

We define a *run* of S to be an infinite sequence of states $\sigma : s_0, s_1, s_2, \ldots$ satisfying the requirements of

– *Initiality*: s_0 is initial, i.e. $s_0 \models \theta$.
– *Consecution*: For each $j = 0, 1, \ldots$, the state s_{j+1} is an S-successor of the state s_j.

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

– *Justice*: For every $t \in \mathcal{F}$ there are infinitely many $J[t]$-states in σ.
– *Compassion*: For every $t \in \mathcal{F}$, if σ contains infinitely many $\mathcal{C}[t]$‘p-states then it must also contain infinitely many $\mathcal{C}[t]$‘q-states.

A fair run is called a *computation*.

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

```
PFS: TYPE =
 [# initial: PREDICATE,
    rho: BI_PREDICATE,
    justice: JUSTICE_TYPE,
    compassion: COMPASSION_TYPE #]
```

where

```
PREDICATE: TYPE = [STATE → bool]
BI_PREDICATE: TYPE = [STATE, STATE → bool]
JUSTICE_TYPE: TYPE = [FAIRNESS_DOMAIN → PREDICATE]
COMPASSION_PAIR: TYPE = [# p: PREDICATE, q: PREDICATE #]
COMPASSION_TYPE: TYPE = [FAIRNESS_DOMAIN → COMPASSION_PAIR]
```

We note that this differs from the definition given above in that it does not have state or fairness domain components. When importing the PFS theory, the user must instantiate it with `STATE` and `FAIRNESS_DOMAIN` parameters. These are both uninterpreted types, which must be defined by the user for each system.

A state sequence is represented in our system as a mapping from time (the natural numbers) to states:

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

We define the `RUN` and `COMPUTATION` types as follows:

```
consecution(seq, pfs): bool =
   ∀ (t: TIME): pfs‘rho(seq(t), seq(t+1))

run(seq, pfs): bool =
   pfs‘initial(seq(0)) ∧ consecution(seq, pfs)

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

compassionate(seq, pfs): bool =
```

```
∀ (f: FAIRNESS_DOMAIN):
   (∀ (t: TIME): ∃ (j: TIME):
     j ≥ t ∧ pfs`compassion(f)`p(seq(j))) →
    (∀ (k: TIME): ∃ (l: TIME):
       l ≥ k ∧ pfs`compassion(f)`q(seq(l)))

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

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

2.2 Linear Temporal Logic

We will first give a brief overview of LTL, and then discuss our implementation of this theory within PVS.

Temporal formulas are interpreted over *models*, infinite runs $\sigma : s_0, s_1 \ldots$ in which each state s_i provides an interpretation for the variables in V. We give an inductive definition for the notion of temporal formula p holding at position $j \geq 0$ in σ, denoted by $(\sigma, j) \models p$:

For a propositional predicate p,

- $(\sigma, j) \models p \iff s_j \mid\models p$
  That is, we evaluate p locally using the interpretation given by s_j.

For the boolean connectives,

- $(\sigma, j) \models \neg p \iff (\sigma, j) \not\models p$
- $(\sigma, j) \models p \vee q \iff (\sigma, j) \models p$ or $(\sigma, j) \models q$

For the temporal operators,

- $(\sigma, j) \models \Box\, p \iff$ for all $k \geq j, (\sigma, k) \models p$
- $(\sigma, j) \models \Diamond\, p \iff$ for some $k \geq j, (\sigma, k) \models p$
- $(\sigma, j) \models p \mathcal{U} q \iff$ for some $k \geq j, (\sigma, k) \models q$ and for all $j \leq i < k, (\sigma, i) \models p$
- $(\sigma, j) \models \bigcirc\, p \iff (\sigma, j+1) \models p$

The derived *entailment* operator is defined by

$$p \Longrightarrow q \quad \text{is} \quad \Box(p \rightarrow q)$$

We now give our PVS implementation: Recalling that *predicates* are properties that are defined on individual states, we define a *temporal predicate*, TP, as a property that is interpreted over the state sequence:

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

```
OR: [TP, TP → TP] =
    (λ a, b: (λ state_seq, t: a(state_seq, t) ∨ b(state_seq, t)));
AND: [TP, TP → TP] =
    (λ a, b: (λ state_seq, t: a(state_seq, t) ∧ b(state_seq, t)));
NOT: [TP → TP] = (λ a: (λ state_seq, t: ¬ a(state_seq, t)));
IMPLIES: [TP, TP → TP] =
    (λ a, b: (λ state_seq, j: ¬ a(state_seq, t) OR b(state_seq, t)));
G: [TP → TP] =
    (λ a: (λ state_seq, j: ∀ k: k ≥ j → a(state_seq, k)));
F: [TP → TP] =
    (λ a: (λ state_seq, j: ∃ k: k ≥ j ∧ a(state_seq, k)));
X: [TP → TP] = (λ a: (λ state_seq, j: a(state_seq, j+1)));
U: [TP, TP → TP] = (λ a, b: (λ state_seq, j: ∃ k: k ≥ j ∧
     b(state_seq, k) ∧ (∀ l: j ≤ l ∧ l < k → a(state_seq, l))));
```

Fig. 1. Temporal operator definitions

State predicates and temporal predicates can be combined together using the boolean connectives (disjunction, conjunction, negation or implication) or temporal operators to generate new temporal predicates. The temporal operators henceforth ($\Box$), eventually ($\Diamond$), next ($\bigcirc$) and until ($\mathcal{U}$) are defined within the system as `G`, `F`, `X` and `U`, respectively (Fig. 1). The user can define and add any other temporal operators desired.

2.3 Validity of LTL Properties in PFS Systems

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

Temporal predicate *tp* is *P-valid* if it holds in the first state of every computation of program P, i.e. $P \models tp$. A predicate a 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: TP, pfs): bool =
    ∀ seq: computation(seq, pfs) → tp(seq, 0)
is_P_state_valid(a: PREDICATE, pfs): bool =
    ∀ seq, t: computation(seq, pfs) → a(seq(t))
```

There is automatic conversion from `PREDICATE` to `TP`, so it is syntactically correct to write `is_P_valid(a, pfs)` for predicate a.

3 The BAKERY Algorithm

In this section we define a parameterized fair system (PFS) for a version of Lamport's BAKERY algorithm [3] as presented in Fig. 2.

$$
\begin{array}{l}
\textbf{local } y : \textbf{array } [1\ldots] \textbf{ of natural where } y = 0 \\
\Big\|_{p \geq 1} P[p] :: \left[\begin{array}{l} \ell_0 : \quad \textbf{loop forever do} \\ \left[\begin{array}{l} \ell_1 : \textbf{NonCritical} \\ \ell_2 : y[p] := \textbf{choose } m \textbf{ such that } \forall q : (m > y[q]) \\ \ell_3 : \textbf{await } \forall q : (y[q] = 0 \lor y[p] < y[q]) \\ \ell_4 : \textbf{Critical} \\ \ell_5 : y[p] := 0 \end{array} \right] \end{array} \right]
\end{array}
$$

Fig. 2. Parameterized mutual exclusion algorithm BAKERY

The program is written in the simple programming language (SPL) of [6]. Processes co-ordinate access to a critical section (ℓ_4). On desiring to enter the critical region, process $P[i]$ is allocated a "ticket number" $y[i]$, which is greater than the ticket numbers of all other processes waiting to enter the critical region. The process with the lowest positive ticket number is allowed to enter the critical region. On exiting the critical region, the ticket number is changed to 0, indicating that the process is not competing in entering the critical region.

We note that our program is parameterized by an unbounded domain – it contains a process $P[i]$ for every $i \geq 1$.

The properties which we would like to show are:

- *Mutual exclusion:* At most one process can be at location ℓ_4 at a time.
- *Accessibility:* If a process is at location ℓ_2, it will eventually reach location ℓ_4. That is, a process desiring to enter the critical region will succeed in doing so.

The system state defines the current location and y-values for each of the processes. In Fig. 12, at the end of the paper, we define BAKERY in the PVS specification language.

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 process p, we create a justice requirement stating that process p must infinitely often be in a state in which the transition at line ℓ_i is not enabled.

There is no justice requirement associated with location ℓ_1 as there is no restriction that any process leave the non-critical section. We note that the justice requirements associated with locations ℓ_0, ℓ_4 and ℓ_5 simply assert that the process p is not at location ℓ_0, ℓ_4 or ℓ_5, respectively. In contrast, the requirement associated with location ℓ_3 asserts that either the process is not at ℓ_3 or it cannot progress as it must "wait" for its y-value to become minimal.

The justice requirement condition associated with location ℓ_2 asserts that either the process p is not at location ℓ_2, or there is no natural number m greater than all the current y-values. The second condition is always false – in our interleaving model, at any time only a finite number of processes can have progressed from the initial states and have positive y-value. The justice conditions must include all conditions in which the transition cannot be taken,

```
BINV: LEMMA
  ∀ pfs: ∀ (a: PREDICATE): ∀ seq:
     computation(seq, pfs) →
       (a(seq(0)) ∧
         (∀ t: pfs`rho(seq(t), seq(t+1)) ∧ a(seq(t)) → a(seq(t+1))))
           → is_P_valid(G(a), pfs)
INV: LEMMA
  ∀ pfs: ∀ (a, b: PREDICATE): ∀ seq:
     computation(seq, pfs) →
       ((∀ t: b(seq(t)) → a(seq(t))) ∧ b(seq(0)) ∧
         (∀ t: pfs`rho(seq(t), seq(t+1)) ∧ b(seq(t)) → b(seq(t+1))))
           → is_P_valid(G(a), pfs)
MONI: LEMMA
  ∀ pfs: ∀ (a, b: PREDICATE):
    is_P_state_valid((a → b), pfs) ∧ is_P_valid(G(a), pfs)
       → is_P_valid(G(b), pfs)
CONI: LEMMA
  ∀ pfs: ∀ (a, b: PREDICATE):
    is_P_valid(G(a), pfs) ∧ is_P_valid(G(b), pfs)
       → is_P_valid(G(a ∧ b), pfs)
```

Fig. 3. Invariance rules

and syntactically (if not semantically) the case of there being no satisfying m is one of them. The fulfillment of this requirement must be formally proved.

The compassion conditions are left "empty" (using an empty compassion definition defined in the PFS theory which assigns the trivial requirement $\langle \mathrm{T}, \mathrm{T} \rangle$ to every $t \in \mathcal{F}$).

The detailed definition of the PFS for the BAKERY algorithm is presented in order to illustrate the principles underlying such a representation. We are currently developing a compiler from SPL to PVS which will generate automatically such a representation, given (the ASCII version of) the SPL program of Fig. 2.

4 Proving Safety Properties

In this section we examine methods for verifying safety properties, that is properties of the form `is_P_valid(G(`a`), pfs)` for predicate a and system `pfs`.

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. The rules are taken from [6]. We list here the four used most often, and give their PVS definitions in Fig. 3.

Each rule is also presented symbolically in a figure. We note that, by convention, in these figures all proof lines with temporal operators are to be interpreted as P-valid, all proof lines without temporal operators as P-state valid.

Rule BINV

For predicate a,

$$\frac{\begin{array}{l} \text{B1. } \Theta \;\rightarrow\; a \\ \text{B2. } a(V) \wedge \rho(V, V') \;\rightarrow\; a(V') \end{array}}{\Box\, a}$$

Fig. 4. Rule BINV (basic invariance)

Rule BINV**:** This is the basic invariance rule (Fig. 4). It states that if a holds at the initial system state, and is preserved by all transitions, then a is a system invariant.

Rule INV

For predicates a, b,

$$\frac{\begin{array}{l} \text{I1. } b \rightarrow a \\ \text{I2. } \Theta \;\rightarrow\; b \\ \text{I3. } b(V) \wedge \rho(V, V') \;\rightarrow\; b(V') \end{array}}{\Box\, a}$$

Fig. 5. Rule INV (general invariance)

Rule INV**:** The general invariance rule (Fig. 5). It states that if b holds at the initial system state and is preserved by all transitions, and b implies a, then a is a system invariant.

We note that the symbolic representation of BINV and INV, as given in Figs. 4 and 5, differs from the implementation (Fig. 3) in that the latter is formulated in terms of computations, and the former in terms of states. The premises in the symbolic representation are a little stronger than in the implementation – B2 requires that for any states *current* and its ρ-successor *next*, if a holds at *current* then it holds at *next*. This includes the case where *current* is not reachable, that is, can never occur in a run. If, in order to prove the validity of premise B2, we want to use another invariant, we must show that *current* is reachable. Even when this is the case, it may be difficult to prove.

Our implementation considers only reachable states. Doing so allows us to easily add previously proven invariants at all stage of the proof.

Rule MONI

For predicates a, b,

$$\frac{\begin{array}{c}\text{M1. } \Box\ a \\ \text{M2. } a \rightarrow b\end{array}}{\Box\ b}$$

Fig. 6. Rule MONI (monotonicity of invariances)

Rule MONI: Monotonicity of invariants rule (Fig. 6). If a is a system invariant, and a implies b, then b is a system invariant.

Rule CONI

For predicates a, b,

$$\frac{\begin{array}{c}\text{C1. } \Box\ a \\ \text{C2. } \Box\ b\end{array}}{\Box\ (a \wedge b)}$$

Fig. 7. Rule CONI (conjunction of invariances)

Rule CONI: Conjunction of invariances rule (Fig. 7). If a and b are each system invariants, then so is $a \ \wedge \ b$.

Each of these proof rules has an associated strategy. Calling this strategy, with the appropriate parameters, effectively applies the rules. e.g. invoking `(INV "a" "b")` will import rule INV and generate three subgoals: the first showing that b implies a, the second that b holds at the initial state, and the third that b is preserved by all transitions. The strategy attempts to discharge the first two subgoals and, depending on the success of these attempts, the system returns to the user one, two, or three subgoals.

4.1 Example: BAKERY Implements Mutual Exclusion

We return to the BAKERY algorithm of Section 3 and show how mutual exclusion can be proved using safety rules.

The property of mutual exclusion is not inductive, and so it cannot be proved using rule BINV. Instead we prove a reachability invariant, showing that the inductive predicate `reachable` is true of all reachable states. Many verification efforts include such an invariant, and our system treats the reachability invariant specially. The 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 (including liveness proofs).

```
reachable: PREDICATE =
  (λ (st: STATE): ∀ (i: PROC_ID):
    (st‘y(i) = 0 ↔ (st‘pc(i) ≤ 2)) ∧
    (∀ (q: PROC_ID): q ≠ i ∧ st‘y(i) ≠ 0 → st‘y(q)≠st‘y(i)) ∧
    (st‘pc(i) ≥ 4 →
      st‘y(i) > 0 ∧
      (∀ (q: PROC_ID): q ≠ i →
      st‘pc(q) < 4 ∧ (st‘y(q) = 0 ∨ st‘y(i) < st‘y(q)))))

reachableInv: LEMMA is_P_valid(G(reachable), pfs)
```

In the [6] notation, this predicate expresses the assertion

$$\begin{aligned}&\forall i : (y[i] = 0 \iff \ell_{0..2}[i])\\ &\quad \wedge (\forall q \neq i : y[i] \neq 0 \rightarrow y[q] \neq y[i])\\ &\quad \wedge (\ell_{4,5}[i] \rightarrow \forall q \neq i : \ell_{0..3}[q] \wedge (y[q] = 0 \vee y[i] < y[q]))\end{aligned}$$

We will now show how we prove this formula. First, the PVS prover is invoked for the lemma, and we are prompted for a command:

```
reachableInv :

  |-------
{1}   is_P_valid(G(predicate_to_TP(reachable)), pfs)

Rule?
```

The appropriate command is `(BINV "reachable")`, which tells the system to try to use the basic invariance rule to prove formula `reachable`. More precisely, we call the strategy `BINV` with the parameter `reachable`. This strategy invokes the `BINV` lemma defined in Fig. 3, expands out definitions, instantiates and simplifies. It generates two requirements: the first showing that `reachable` holds in the initial state, and the second that it is preserved by all transitions. The first requirement is discharged by the strategy, and the system returns the second to the user.

We present now the *output* of the TLPVS system during this proof, followed by an explanation of what occurs. The only inputs by the user are the commands appearing immediately after the PVS Rule? prompt.

```
reachableInv :

  |-------
{1}   is_P_valid(G(predicate_to_TP(reachable)), pfs)

Rule? (BINV "reachable")

lemma BINV,
this simplifies to:
reachableInv :
;;;The inductive step of the BINV rule

{-1,(rho)}
          current!1`pc(p!1) = 0 AND
           next!1`y = current!1`y AND
            next!1`pc = current!1`pc WITH [(p!1) := 1]
       OR current!1`pc(p!1) = 1 AND
           next!1`y = current!1`y AND
            next!1`pc = current!1`pc WITH [(p!1) := 2]
       OR current!1`pc(p!1) = 2 AND
           (EXISTS (m: nat):
              (FORALL q: current!1`y(q) < m) AND
               next!1`y = current!1`y WITH [(p!1) := m])
            AND next!1`pc = current!1`pc WITH [(p!1) := 3]
       OR current!1`pc(p!1) = 3 AND
           (FORALL q:
              q /= p!1 IMPLIES
               current!1`y(q) = 0 OR
                 current!1`y(p!1) <= current!1`y(q))
            AND next!1`y = current!1`y AND
             next!1`pc = current!1`pc WITH [(p!1) := 4]
       OR current!1`pc(p!1) = 4 AND
           next!1`y = current!1`y AND
            next!1`pc = current!1`pc WITH [(p!1) := 5]
       OR current!1`pc(p!1) = 5 AND
           next!1`y = current!1`y WITH [(p!1) := 0] AND
            next!1`pc = current!1`pc WITH [(p!1) := 0]
{-2,(reachable invariant)}
      FORALL (i: PROC_ID):
        (current!1`y(i) = 0 IFF
          (current!1`pc(i) = 0 OR
            current!1`pc(i) = 1 OR current!1`pc(i) = 2))
```

```
          AND
          (FORALL (q: PROC_ID):
             q /= i AND current!1`y(i) /= 0 IMPLIES
              current!1`y(q) /= current!1`y(i))
           AND
           (current!1`pc(i) >= 4 IMPLIES
             current!1`y(i) > 0 AND
              (FORALL (q: PROC_ID):
                 q /= i IMPLIES
                  current!1`pc(q) < 4 AND
                   (current!1`y(q) = 0 OR
                      current!1`y(i) < current!1`y(q))))
  |-------
{1,(rtp)}
      (next!1`y(i!1) = 0 IFF
        (next!1`pc(i!1) = 0 OR next!1`pc(i!1) = 1
           OR next!1`pc(i!1) = 2))
       AND
       (FORALL (q: PROC_ID):
          q /= i!1 AND next!1`y(i!1) /= 0 IMPLIES
           next!1`y(q) /= next!1`y(i!1))
        AND
        (next!1`pc(i!1) >= 4 IMPLIES
          next!1`y(i!1) > 0 AND
           (FORALL (q: PROC_ID):
              q /= i!1 IMPLIES
               next!1`pc(q) < 4 AND
                (next!1`y(q) = 0 OR next!1`y(i!1) < next!1`y(q))))

Rule? (SPLIT-RHO-ALL ("q!1" "i!1"))

Q.E.D.
```

We note that like many other strategies, this one labels formulas. The first antecedent is labeled (rho), as it represents the transition relation. The second is the reachable property, and states that `reachable` holds in the current state. The consequent, what we are "required to prove", labeled rtp, states that `reachable` holds in the next state.

We now invoke a very useful strategy `split-rho-all`. This strategy splits the transition relation generating a number of sub-goals. On each branch it then performs various splitting and simplification methods, as well as instantiating any universally quantified formulas (forall) with the constants in the parameter list given to the strategy.

In this case the proof contains one constant `i!1`, and another, `q!1`, will be created from the skolemization of the `FORALL (q: PROC_ID)` quantification in the consequent (rtp). It turns out that these two instantiations are necessary, and sufficient, for the proof to be completed.

Thus the very brief proof

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

suffices.

Calling strategy `BINV` and then `split-rho-all` is standard practice when proving reachability invariants. The only real understanding required of the user, for this simple example, is identifying the constants `i!1` and `q!1`.

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

```
mutex: PREDICATE =
  (λ (st: STATE):
    ∀(i, q: PROC_ID): st‘pc(i) = 4 ∧ st‘pc(q) = 4 → i = q)

mutualExclusion: LEMMA
  is_P_valid(G(mutex), pfs)
```

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

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

Of these commands, strategies `use-invariant` and `split-all-inst` were written by us, while `inst?` and `expand*` are builtin PVS commands.

5 Proving Response Properties

In this section we consider rules for proving simple response properties. These are properties of the form `is_P_valid(`G`(p → `F`(q)), pfs)`, for predicates p and q. In the syntax of [6], these are formulas of the form $P \models p \Longrightarrow \Diamond q$.

For example, the formula

```
is_P_valid(G((λ st: st‘pc(z) = 2) → F(λ st: st‘pc(z) = 4)), 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 ℓ_2, it will eventually enter the critical section at location ℓ_4.

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. Since the rank cannot decrease infinitely often, a q-state must eventually be reached.

We give some background definitions, and then proceed to detail some of the proof rules that we have implemented in our system. The soundness of each of the rules has been formally verified. Due to space considerations, we will not discuss the proofs of the soundness of these rules, but only give a brief, intuitive justification. Each rule has an associated strategy, which, similarly to the strategies of Section 4, invokes the rules and perform some simplification. In Section 5.4 we illustrate the use of the proof rules by proving accessibility in the BAKERY algorithm.

5.1 Background

Well-foundedness Our liveness 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 $a, b \in \mathcal{B}$, $a = b$ 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.

Proving well-foundedness is often a non-trivial exercise in itself, even in the case of obviously well-founded comparators such as the less-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.

These proofs are formulated as theories which can be imported when appropriate.

Fairness Domain The *fairness domain* $\mathcal{F}$ is a domain used to parameterize the fairness requirements of the system. Typically, it effectively partitions the transition relation ρ into transitions, one transition for each element of the fairness domain.

As an example we consider BAKERY, for which we defined the fairness domain as the tuple `[# loc: LOCATION, pid: PROC_ID #]`. Consider a state s in BAKERY in which process 1 is at location ℓ_0 and process 2 at location ℓ_5. Fairness domain element `(# loc:=0, pid:=1 #)` signifies the transition by which process 1 advances from location ℓ_0 to location ℓ_1; element `(# loc:=5, pid:=2 #)` signifies process 2 taking a step from location ℓ_5 to location ℓ_0. The transition corresponding to element `(# loc:=1, pid:=1 #)` is not enabled – it will be enabled only when process 1 reaches location ℓ_1. In this example, the idling transition is not covered by the fairness domain.

Rule WELL

For a PFS with	
fairness domain	$\mathcal{F}$,
state space	Σ,
and justice conditions	$\mathcal{J}$;
Given	
initial and goal predicates	p, q
helpful predicates	$\{h_t : t \in \mathcal{F}\}$,
a well-founded relation	$\succ$ over $\mathcal{A}$,
and ranking function	$\delta : \Sigma \mapsto \mathcal{A}$

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

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

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

Fig. 8. Rule WELL

5.2 Rule WELL

Rule WELL (Fig. 8) is a variation of rule WELL-P of [9].

The WELL rule traces the progress of a computation from an arbitrary p-state to an unavoidable q-state. With each *helpful* predicate h_t of the rule we associate the justice requirement $\mathcal{J}[t]$. Intuitively, the helpful predicate 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.

Premise W2 assures that the application of a transition to a state satisfying predicate h_t will never cause the rank 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.

Our system also includes a parameterized version of the WELL rule. This extended version of WELL allows us to prove parameterized properties of the form $\forall i : p_i \Longrightarrow \Diamond\, q_i$.

Using Rule WELL in Unbounded Systems Rule WELL is suited to both bounded and unbounded systems. However, in modifying a system from bound-

Rule DIST-RANK

For a PFS with
countable fairness domain $\mathcal{F}$,
state space Σ,
and justice conditions $\mathcal{J}$;
Given
initial and goal predicates p, q,
helpful predicates $\{h_t : t \in \mathcal{F}\}$,
and ranking functions $\{\delta_t : \Sigma \mapsto \mathbb{N} \mid t \in \mathcal{F}\}$ with finite support

$$\begin{array}{lll}
\text{D1. } p & \rightarrow q \ \vee \ \bigvee\limits_{t \in \mathcal{F}} h_t & \\
\text{D2. } h_t \ \wedge \ \rho & \rightarrow q' \ \vee \ h'_t \ \vee \ \left(\delta_t > \delta'_t \ \wedge \ \bigvee\limits_{u \in \mathcal{F}} h'_u\right) & \\
\text{D3. } h_t \ \wedge \ \rho & \rightarrow q' \ \vee \ \bigvee\limits_{d \in \mathcal{F}} \bigwedge\limits_{u \in \mathcal{F}} (\delta_u \geq \delta'_u \ \vee \ \delta_d > \delta'_d \ \wedge \ \delta_d > \delta'_u) & \\
\text{D4. } h_t & \rightarrow \neg \mathcal{J}[t] &
\end{array} \left.\vphantom{\begin{array}{l}a\\a\\a\\a\\a\\a\end{array}}\right\} \text{For } t \in \mathcal{F}$$

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

Fig. 9. Rule DIST-RANK

edly parameterized to unbounded, care should be taken to ensure that the ranking function $\succ$ remains well-founded over the domain $\mathcal{A}$. Ranking functions which are well-founded over a parameterized rank domain $\mathcal{A}$, are not necessarily well-founded when $\mathcal{A}$ is an unbounded domain.

5.3 Rule DIST-RANK

We have found it useful to derive a new, *distributed* response rule in which the rank is distributed over the fairness domain. In many applications this rule is easier to use than the WELL rule. The distributed rank rule, DIST-RANK (Fig. 9) is a restricted version of the general WELL rule where the ranking functions assume a very special form. We note:

- The fairness domain must be *countable*. That is, there must be an injective mapping from it to the naturals.
- The system must exhibit *finite support*. In any state, at most a finite number of fairness domain elements can have a positive rank.
- When a domain element $t \in \mathcal{F}$ becomes unhelpful, its rank δ_t decreases.
- The rank of fairness domain element u is allowed to increase provided the new value δ'_u is smaller than δ_d for some $\delta_d > \delta'_d$.
- The relationship between helpful and just transitions is separated from the ranking function and is formulated as premise D4. This typically simplifies

proofs. However, in the special case of all-true justice conditions we would need to use a version of the rule where D2 and D4 are combined (as in WELL).

Intuitively, whereas in the WELL rule we required that the value of a centralized ranking function eventually decreases, in the DIST-RANK rule we require that one element of a distributed ranking function eventually decreases. When an element decreases, other elements are allowed to increase providing that their new values are strictly smaller than the old value of the decreasing element.

The well-foundedness of the ranking function depends on it always having finite support. That is, it must be shown that only a finite number of fairness domain elements can have a positive rank at any point. This is always the case when the fairness domain is finite, however when it is unbounded it must be shown to be a system property. Finite support can be guaranteed by requiring that the rank of all transitions for a process which is still in its initial state are zero. Rules and strategies are available to aid the user in proving finite support.

The derivation of rule DIST-RANK from rule WELL is as follows: Consider a ranking function $\zeta_i : \Sigma \mapsto \mathbb{N}$ defined for all $i \in \mathbb{N}^+$ as follows:

$$\zeta_i = \sum_{t \in \mathcal{F}} \delta_t = i$$

That is, for every state st and $i \geq 1$, ζ_i returns the number of fairness domain elements t which have rank i in st. It can be shown that the lexicographical ordering of ζ_i is well-founded over a computation satisfying the premises of rule DIST-RANK. Using this ranking function, DIST-RANK can be derived from WELL.

5.4 Proving Accessibility in BAKERY

In this section we demonstrate how the WELL and DIST-RANK rules can be used to prove accessibility of the critical region in BAKERY.

The property that we would like to prove is that if an arbitrary process z is waiting to enter the critical region (i.e. is at location ℓ_2) it will eventually do so (i.e. it will reach location ℓ_4). The predicates characterizing the initial and goal states (p and q of Figs. 8 and 9) are:

```
waiting: PREDICATE = (λ st: st`pc(z) = 2)
critical: PREDICATE = (λ st: st`pc(z) = 4)
```

and the property we want to prove is:

```
accessibility: LEMMA
   is_P_valid(G(waiting → F(critical)), pfs)
```

Proof Using Rule WELL We first define the helpful set. Intuitively, at every step we want to define some transition by some process as helpful inasmuch as it will bring process z closer to entering the critical section. Taking the helpful transition should decrease the rank.

When process z is at location ℓ_2, there is no restriction on it progressing to location ℓ_3, and this is the most helpful step that can occur in the system. However, once z is at location ℓ_3, it cannot progress if there is any other process in the system with smaller y-value. Thus, process z must wait for all processes with positive y-values smaller than its own to enter and exit the critical region. Of these processes only the one with the smallest y-value is able to progress. The most helpful transition is thus for the process with the smallest y-value to take a step. More formally, a transition $d \in \mathcal{F}$ is *helpful* at state $st \in \Sigma$ under the following conditions:

```
IF d`loc = 2
   THEN d`pid = z ∧ st`pc(z) = 2
ELSE st`pc(z) = 3 ∧ d`loc > 2 ∧ small(st) = d`pid
   ∧ st`pc(small(st)) = d`loc
ENDIF
```

where `small` returns the process identity (pid) of the process with the smallest positive y-value at a given state, if such a process exists.

Before defining the ranking functions we define the *stage* of a process as follows:

```
stage((st: STATE), (p: PROC_ID)): upto[3] =
     COND st`pc(p) = 3 ∧ st`y(p) < st`y(z) → 3,
          st`pc(p) = 4 → 2,
          st`pc(p) = 5 → 1,
          ELSE → 0
     ENDCOND
```

The rank element type ($\mathcal{A}$) and ranking function (δ) are defined as:

```
RANK_ELT: TYPE = [bool, nat, upto[3]]

rank: [STATE → RANK_ELT] =
     λ (st: STATE):
       IF st`pc(z) = 3 ∧ st`y(z) > 0
         THEN (FALSE, st`y(z) − st`y(small(st)),
                      stage(st, small(st)))
       ELSE (TRUE, 0, 0)
       ENDIF
```

Intuitively, the first, boolean, component of a rank element is true if process z is at location ℓ_2, false if z is at location ℓ_3 (we are not interested in cases were z is at another location). If z is at location ℓ_3, then the second component is defined as the difference between its y-value and that of the process with the smallest y-value. This difference is an upper bound on the maximum number of processes that can precede z to the critical section. The third component decreases as process `small(st)` progresses into and out of the critical section, clearing the way for next process to enter.

The rank comparator `gt` is defined as:

```
gt(r, s: RANK_ELT): bool =
    r‘1 = TRUE ∧ s‘1 = FALSE ∨
     r‘1 = FALSE ∧ s‘1 = FALSE ∧
        (r‘2 > s‘2 ∨ r‘2 = s‘2 ∧ r‘3 > s‘3)
```

Where $r`1$ gives $r$'s first component, $r`2$ and $r`3$ its second and third components, respectively.

Function `gt` is the natural lexicographic evaluation of two rank elements, where the boolean true is greater than the boolean false.

Proof Using Rule DIST-RANK The helpful set is defined as for rule WELL.

Whereas for the WELL rule we defined a centralized ranking function, for the DIST-RANK rule, we define a rank for every fairness domain element. Intuitively, for every process p, if it will definitely have a helpful transition in the future, then p's first transition to become helpful will have a positive rank.

When process z is at location ℓ_2 the only transition that we know will definitely be helpful is that of z moving to ℓ_3. However, once process z reaches ℓ_3, every process p with a positive y-value smaller than that of z will eventually have a helpful transition (when it becomes process `small` with smallest y-value). The next transition to be enabled, that from p's current location, is assigned a positive rank. Our ranking function is defined as:

```
rank: [FAIRNESS_DOMAIN → [STATE → nat]] =
    λ (t: FAIRNESS_DOMAIN):
      λ (st: STATE):
        COND t‘loc = 2 ∧ st‘pc(t‘pid) = 2 ∧ t‘pid = z → 4,
             t‘loc = 3 ∧ st‘pc(t‘pid) = 3 ∧ st‘pc(z) = 3
                                ∧ st‘y(t‘pid) ≤ st‘y(z) → 3,
             t‘loc = 4 ∧ st‘pc(t‘pid) = 4 ∧ st‘pc(z) = 3 → 2,
             t‘loc = 5 ∧ st‘pc(t‘pid) = 5 ∧ st‘pc(z) = 3 → 1,
             ELSE → 0
        ENDCOND
```

We note that when z is at location ℓ_2, the fairness domain element `(# loc:=2, pid:=z #)`, with rank 4, is the only element with positive rank. When z reaches location ℓ_3, that element becomes unhelpful and assumes rank zero. Process z's next helpful transition is that from location ℓ_3, and the associated element has rank 3. In fact, there may be many processes p at location ℓ_3, and with y-value no greater than that of z. For each of these the fairness domain element `(# loc:=3, pid:=p #)` is allocated rank 3. This increase in ranks is allowed by our ranking function as all the positive ranks are less than the old rank, 4, of `(# loc:=2, pid:=z #)`.

```
type   ℕ^{>1}     : {i : ℕ | i > 1}
local  generator  : [counter : ℕ^+]
                    init counter = 1
       sieves     : [ℕ^{>1} ↦ [pc : [0..4], prime : ℕ^{>1}]]
                    init pc = 0
       queues     : [ℕ^{>1} ↦ list[ℕ^{>1}]]
                    init empty

      generator :: [ m_0 : loop forever do
                         [ m_1 : push(counter++, queues[2]); ] ]

||    ||         sieves[i] :: [ l_0 : await ¬empty(queues[i]);
  i∈ℕ^{>1}                      l_1 : prime := pop(queues[i]);
                                l_2 : loop forever do
                                    [ l_3 : await ¬empty(queues[i]);
                                      l_4 : t := pop(queues[i]);
                                            if ¬divides(prime, t)
                                            then push(t, queues[i + 1])
                                            endif; ] ]
```

Fig. 10. Parameterized prime sieve algorithm PRIME-SIEVE

5.5 Prime Numbers Sieve

Eratosthenes' prime sieve algorithm is an ancient algorithm for identifying the prime numbers. The algorithm is inherently unbounded as it can be used to verify the primality of arbitrarily large numbers. In this section we verify a parameterized, parallel version of this algorithm.

This example is interesting for two reasons: Firstly, the algorithm is intrinsically unbounded and not fully symmetric, making it more natural, and more challenging, than BAKERY. Secondly, though we again use rule DIST-RANK, the ranking function that we construct is very different from that of BAKERY, highlighting the flexibility which this rule allows.

The processes in the PRIME-SIEVE algorithm (Fig. 10) are a single generator (process identifier 1) and an unbounded number of queues with identifiers $2, 3, \ldots$. The generator outputs the list of natural numbers from two upward, putting them on its out-queue, queue 2.

The numbers bubble through an unbounded sequence sieves. Composite numbers are eventually eliminated, and prime numbers identified. The sieves are linked to one another by queues (of unbounded length). For every $i > 1$, queue i is the in-queue of sieve i, and queue $i + 1$ is its out-queue.

Each sieve can identify at most one prime, and on identifying one, stores it. The sieve then compares its prime to all "potentially prime" numbers it receives, eliminating tbose which are multiples of its prime. Thus the sieves sift through the numbers, eliminating those which are not prime, and storing the primes.

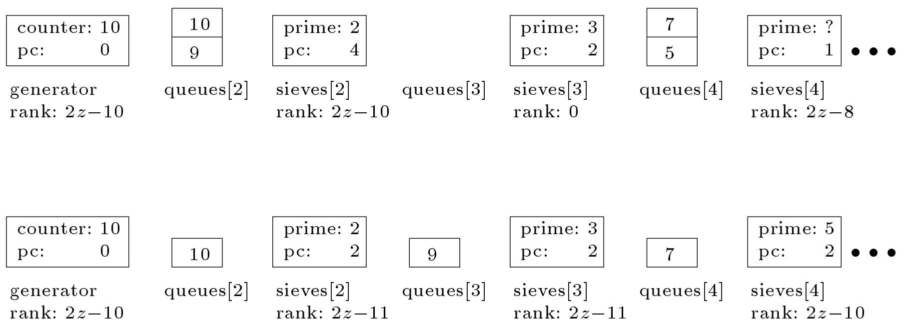

Fig. 11. An example execution of PRIME-SIEVE. The top row illustrates a system in which the first two primes have been found. All sieves from sieves[4] onward are inactive, and similarly all queues from queues[5] onward are empty. The second row shows the same system after sieves[2] and sieves[4] have each taken one step. The rank given in the diagram is that for the fair domain element corresponding to the process's program counter. The example assumes that $z > 10$.

Every sieve i starts at location ℓ_0 where it waits for a number to enter its in-queue. The first number popped off the in-queue is stored in the sieve's *prime* field (ℓ_1). The sieve, now termed *active*, loops at locations $\ell_{2..4}$ where it checks numbers on its in-queue against its prime. Numbers which are multiples of the prime are eliminated, others are pushed onto the sieve's out-queue, queue $i+1$. Sieve $i+1$ compares these numbers to its prime, and so on. A number z which is not a prime will be eliminated when it reaches a sieve whose prime is a factor of z. If z is a prime, then no sieve will eliminate it and it will bubble through the system until it reaches an inactive sieve, which will store z as its prime.

We consider the example of Fig. 11. The top row shows a system in which the first two primes have been found. The bottom row shows the same system after sieves 2 and 4 have each taken another step. Sieve 4 pops the value 5 off its in-queue. Since it does not yet have a prime, it recognizes 5 as being a prime and saves it in its *prime* field. Sieve 2 pops the value 9 off its in-queue, and finding that it is not a multiple of 2, puts it in its out-queue. Sieve 3 will eventually recognize 9 as a multiple of 3, and eliminate it.

Proof of Correctness We again use the standard fairness domain: [# `loc`: `LOCATION`, `pid`: `PROC_ID` #], where `PROC_ID` is $\mathbb{N}^+$.

Process 1 is the generator, all other processes are sieves. The justice requirement for the generator is that it not be at location l (for l greater than 1, this is trivially true). The justice requirement for sieves at locations $l \in \{1, 3, 4\}$ is that the sieve leave location l. For $l \in \{0, 2\}$ the sieve is required to leave location l providing the in-queue is non-empty.

We consider the algorithm to be correct if every prime number, and no non-prime number, is eventually recognized by a sieve. We define primality by the predicate *is_prime*:

$$is\_prime[i : \mathbb{N}^+] : \mathbf{bool} = i \geq 2 \ \wedge \ \forall j : j \geq 2 \ \wedge \ j \neq i \rightarrow \neg divides(j, i)$$

The safety property `sieve_prime` asserts that numbers identified as primes are indeed prime. It is verified using the rules of Section 4. Lemma `prime_found` states the response property that for an arbitrary number z, if z is prime then it will eventually be identified as such.

```
sieve_prime: PREDICATE =
  (λ (st: STATE):
    ∀ (sid: PROC_ID):
      st‘sieves(sid)‘pc > 1 → is_prime(st‘sieves(sid)‘prime))

sieve_prime: LEMMA is_P_valid(G(sieve_prime), pfs)

found(p: posnat): PREDICATE =
  (λ (st: STATE):
    ∃ (sid: PROC_ID):
      st‘sieves(sid)‘pc > 1 ∧ st‘sieves(sid)‘prime = p)

prime_found: LEMMA
   is_P_valid(G((λ st: is_prime(z)) → F(found(z))), pfs)
```

We prove lemma `prime_found` using the DIST-RANK rule. We distribute the rank over the sieves with the rank of fairness domain elements being inversely related to both the value at the head of the sieve's in-queue, and the sieve identifier:

```
rank: [FAIRNESS_DOMAIN → [STATE → nat]] =
  λ t: λ st:
     IF t‘pid = 1 THEN
        IF st‘generator‘counter < z ∧ t‘loc ≥ st‘generator‘pc
           THEN 2z − st‘generator‘counter
        ELSE 0
        ENDIF
     ELSE LET inQ = st‘queues(t‘pid), h = head(inQ) IN
        IF ¬ empty?(inQ) ∧ h ≤ z ∧
              t‘loc ≥ st‘sieves(t‘pid)‘pc ∧ t‘pid ≤ z
           THEN (z − h) + (z − t‘pid) + 1
        ELSE 0
        ENDIF
     ENDIF
```

For sieve $i \leq z$ with a non-empty in-queue with value $h \leq z$ at its head, the rank is calculated as $2z - h - i + 1$. Since the numbers in the in-queues are monotonically increasing, the rank of the sieve decreases as it processes queue elements. A sieve with an empty in-queue is assigned rank 0. Therefore, when an element is pushed onto sieve i's empty in-queue, the rank of sieve i increases.

The rank of sieve $i-1$ will, however, decrease (its in-queue is either empty or has a larger value at its head). To ensure that the new rank of sieve i is smaller than the old rank of sieve $i-1$, we subtract the process (sieve) identifier from the rank. Returning to the example of Fig. 10, when sieve 2 pops the value 9 off its in-queue its rank decreases from $2z-10$ to $2z-11$. The rank of sieve 3 increases as its in-queue is no longer empty. Due to the processor identifier component in the ranking function, the new rank of sieve 3, $2z-11$, is lower than the old rank of sieve 2.

Finite support is guaranteed by defining the rank of a sieve i to be zero if $h > z$ or $i > z$. It is easy to see that in both cases the activity of the sieve is no longer of interest for verifying the primality of z: If $h > z$ then z must already have passed through the sieve, or have been eliminated. The prime number of an active sieve is never smaller than the sieve's identifier, and so for $i > z$, $\mathit{sieves}[i].\mathit{prime} > z$ and cannot be a factor of z.

We note that a number of fairness domain elements of one sieve may have a positive rank at the same time. As the sieve progresses to location ℓ_4 the rank of its various domain elements are set to zero. The *number* of domain elements with positive rank can thus be viewed as a counter of sorts, decreasing as the sieve approaches location ℓ_4. (On moving from location ℓ_4 back to ℓ_2, a new, lower, rank is allocated to relevant fairness domain elements.) This mechanism is different from that used in BAKERY where at most one fairness domain element for a process could have non-zero rank at a time, and this rank decreased when the process took a step. In this example we have exploited the option of increasing ranks to construct a creative, and we believe relatively simple, ranking function for PRIME-SIEVE.

A transition of the generator is *helpful* if it is enabled, and the generator counter is less than z. Once z is has been generated, the sieve transitions become helpful. A transition of sieve i is helpful if z is in queue i and the transition is enabled:

```
hset: [FAIRNESS_DOMAIN → PREDICATE] =
  λ t: λ st:
    IF t‘pid = 1
      THEN st‘generator‘counter < z ∧ t‘loc = st‘generator‘pc
    ELSE
      member(z, st‘queues(t‘pid)) ∧ t‘loc = st‘sieves(t‘pid)‘pc
    ENDIF
```

6 Reduction of Compassion to Justice

The rules presented in Section 5 are suitable for proving response properties in systems with no compassion requirements, or where the validity of the property is not dependent on the system compassion requirements. Compassion dependent response properties can be verified using other response rules [5]. Typically, these rules for the general fairness case contain among their premises a temporal premise, while all the other deductive rules infer a temporal property based on non-temporal premises. This deviation from the standard form often causes the verification of response properties for systems with compassion requirements to become more awkward.

An alternative approach proposed here suggests reducing the verification problem $S \models \varphi$, where S is a PFS with non-empty compassion set, and φ is a response property, to the verification problem

$$S_J \models \varphi$$

where S_J is a *compassion free* PFS.

6.1 The Reduction

Assume that the system S is given by the PFS $\langle \Sigma, \Theta, \rho, \mathcal{F}, \mathcal{J}, \mathcal{C} \rangle$.

We recall that the compassion component is of the form $[\mathcal{F} \mapsto \langle p, q \rangle]$, where an *empty* compassion requirement for transition $t \in \mathcal{F}$ was represented by the *trivial* condition $\langle \mathrm{T}, \mathrm{T} \rangle$. We assume that no element $t \in \mathcal{F}$ has both a non-trivial justice and a non-trivial compassion requirement[1].

We let $\mathcal{N} = \{n_t : \text{boolean} \mid t \in \mathcal{F}\}$ be a set of boolean variables disjoint from the set V of system variables of S.

The reduced system S_J is given by $\langle \Sigma_J, \Theta_J, \rho_J, \mathcal{F}, \mathcal{J}_J, \mathcal{C}_J \rangle$ where

- $\Sigma_J = \{s_J \mid \exists (s \in \Sigma) \text{ such that } s_J \Downarrow_V = s\}$
  That is, Σ_J is the set of all states s_J which agree with some state s on the interpretation of all variables in V. In addition to the fields found in states in Σ, states in Σ_J contain fields for all the variables in $\mathcal{N}$.
- $\Theta_J = \Theta \ \wedge \ \bigwedge_{t \in \mathcal{F}} n_t = \mathrm{F}.$
  That is, initially $n_t = \mathrm{F}$ for all the newly introduced boolean variables.
- $\rho_J = \left(\rho \ \vee \ \bigvee_{t \in \mathcal{F}} (n_t = \mathrm{F} \ \wedge \ n'_t = \mathrm{T} \ \wedge \ \mathit{pres}(V \cup \mathcal{N} - \{n_t\})) \right) \ \wedge$
  $\bigwedge_{t \in \mathcal{F}} \neg(n'_t \ \wedge \ \mathcal{C}[t]`p'_t)$
  The augmented transition relation allows each of the n_t variables to change non-deterministically from F to T. It also requires that, for each $\mathcal{C}[t]$, it is never the case that n_t and $\mathcal{C}[t]`p$ are both true at the same time. (If $\mathcal{C}$[t] is trivial we expect n_t to remain false throughout the computation.)
- $\mathcal{J}_J = \lambda(t : \mathcal{F}) : \mathcal{J}[t] \ \wedge \ (n_t \ \vee \ \mathcal{C}[t]`q)$
  We note that when $\mathcal{C}[t]$ is trivial, $\mathcal{C}[t]`q = \mathrm{T}$ and so $\mathcal{J}_J[t] = \mathcal{J}[t]$.
  In the case of $\mathcal{C}[t]$ being non-trivial $\mathcal{J}[t]$ is trivial (equals T), and so $\mathcal{J}_J[t] = n_t \ \vee \ \mathcal{C}[t]`q$. This defines the additional requirement demanding that either $n_t$ turns true sometime, implying that $\mathcal{C}[t]`p$ is continuously false from that time on, or that $\mathcal{C}[t]`q$ holds infinitely often.

[1] This is generally the case. If there are fairness domain elements with both justice and compassion requirements, we can artificially extend the fairness domain to ensure that no element has two fairness requirements.

– $\mathcal{C}_J = \lambda(t : \mathcal{F}) : \langle \mathrm{T}, \mathrm{T} \rangle$
The compassion set is empty, containing only the trivial $\langle \mathrm{T}, \mathrm{T} \rangle$ condition.

Claim (Reduction is sound). Let $S : \langle \Sigma, \Theta, \rho, \mathcal{F}, \mathcal{J}, \mathcal{C} \rangle$ be a PFS and S_J be the corresponding compassion-free reduction of S. Then σ is a computation of S iff there exists a σ_J, a computation of S_J such that $\sigma = \sigma_J \Downarrow_V$, i.e. σ and σ_J agree on the interpretation of all variables in V.

7 Conclusion

In this paper we have presented a new system, TLPVS, for the deductive verification of LTL properties. In addition to the PVS implementation of logic and deductive rules defined in the literature (e.g. [6, 9]), we have also derived new rules and methods which are particularly appropriate for a deductive LTL system. Most notable here are the distributed rank rule and reduction of compassion to justice, both of which greatly simplify the deductive verification of response properties. A notable feature of our system is its suitability for the verification of unbounded systems.

Work on this system continues, and includes the building of a compiler from the SPL programming language [6] into a PFS in the PVS specification language. We are also working on developing PVS strategies to make the system easier to work with, and on a project exploiting its abilities in the verification of unbounded systems in order to verify dynamic systems (object systems).

References

1. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, Apr. 1995. Available, with specification files, at `http://www.csl.sri.com/wift-tutorial.html`.
2. Y. Kesten and A. Pnueli. Control and data abstractions: The cornerstones of practical formal verification. *Software Tools for Technology Transfer*, 2(1):328–342, 2000.
3. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Comm. ACM*, 17(8):453–455, 1974.
4. Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. D. Alfaro, H. Devarajan, H. Sipma, and T. Uribe. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Dept. of Comp. Sci., Stanford University, Stanford, California, 1994.
5. Z. Manna and A. Pnueli. Completing the temporal picture. *Theor. Comp. Sci.*, 83(1):97–130, 1991.
6. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.
7. K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, 1993.
8. S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. *PVS System Guide.* Menlo Park, CA, November 2001.

9. 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.

```
bakery_definition: THEORY
 BEGIN
  LOCATION: TYPE = upto[5]
  PROC_ID: TYPE = posnat
  FAIRNESS_DOMAIN: TYPE = [# loc: LOCATION, pid: PROC_ID #]
  STATE: TYPE = [# y: [PROC_ID → nat], pc: [PROC_ID → LOCATION] #]
  IMPORTING PFS[STATE, FAIRNESS_DOMAIN]
  p, q: VAR PROC_ID
  rho: BI_PREDICATE = (λ (current, next: STATE):
    next = current ∨
    (∃ p:
       current`pc(p) = 0 ∧
       next`y = current`y ∧ next`pc = current`pc WITH [(p) := 1]
     ∨ current`pc(p) = 1 ∧
       next`y = current`y ∧ next`pc = current`pc WITH [(p) := 2]
     ∨ current`pc(p) = 2 ∧
       (∃ (m: nat): (∀ q: current`y(q) < m) ∧
                     next`y = current`y WITH [(p) := m]) ∧
       next`pc = current`pc WITH [(p) := 3]
     ∨ current`pc(p) = 3 ∧
       (∀ q: q ≠ p → current`y(q) = 0 ∨ current`y(p) ≤ current`y(q)) ∧
       next`y = current`y ∧ next`pc = current`pc WITH [(p) := 4]
     ∨ current`pc(p) = 4 ∧ next`y = current`y ∧
       next`pc = current`pc WITH [(p) := 5]
     ∨ current`pc(p) = 5 ∧
       next`y = current`y WITH [(p) := 0] ∧
       next`pc = current`pc WITH [(p) := 0]))
  justice: [FAIRNESS_DOMAIN → PREDICATE] =
    (λ t: FAIRNESS_DOMAIN (λ st: STATE:
       IF t`loc = 2
          THEN st`pc(t`pid) ≠ 2 ∨ ¬ (∃ (m: nat): ∀ p: st`y(p) < m)
       ELSIF t`loc = 3
          THEN st`pc(t`pid) ≠ 3 ∨
            ¬ (∀ q: q ≠ t`pid → st`y(q) = 0 ∨ st`y(t`pid) ≤ st`y(q))
       ELSIF t`loc = 0 ∨ t`loc = 4 ∨ t`loc = 5 THEN st`pc(t`pid) ≠ t`loc
       ELSE TRUE
       ENDIF))
  bakery: PFS =
      (# initial := λ (st: STATE): ∀ p: st`y(p) = 0 ∧ st`pc(p) = 0,
         rho := rho,
         justice := justice,
         compassion := empty_compassion #)

 END bakery_definition
```

Fig. 12. A PFS for the BAKERY algorithm