# A Mechanized Proof Environment For The Convenient Computations Proof Method †

Marcelo Glusman and Shmuel Katz
(marce,katz@cs.technion.ac.il)
*Department of Computer Science*
*The Technion, Haifa, Israel*

**Abstract.** A mechanized verification environment made up of theories over the deductive mechanized theorem prover PVS is presented, which allows taking advantage of the "convenient computations" method. This method reduces the conceptual difficulty of proving a given property for all the possible computations of a system by separating two different concerns: (1) proving that special *convenient* computations satisfy the property, and (2) proving that every computation is related to a convenient one by a relation which preserves the property. The approach is especially appropriate for applications in which the first concern is trivial once the second has been shown, e.g., where the specification itself is that every computation reduces to a convenient one. Two examples are the serializability of transactions in distributed databases, and sequential consistency of distributed shared memories. To reduce the repetition of effort, a clear separation is made between "infrastructural" theories to be supplied as a *proof environment* PVS library to users, and the specification and proof of particular examples. The provided infrastructure formally defines the method in its most general way. It also defines a computation model and a reduction relation – the equivalence of computations that differ only in the order of finitely many independent operations. One way to prove that this relation holds between every computation and some convenient one involves the definition of a measure function from computations into a well-founded set. Two possible default measures, which can be applied in many cases, are also defined in the infrastructure, along with useful lemmas that assist in their usage. We show how the proof environment can be used, by a step-by-step explanation of an application example.

**Keywords:** Formal Verification, Convenient Computations, Proof Environment, PVS, Deductive Theorem Proving

## 1. Introduction

### 1.1. Background and motivation

The task of proving a property for all possible computations of a system is usually made easier by focusing on a subset of them – those we call *convenient* – for which the property is easier to prove. This reduction is sometimes done informally in manual proofs, where it is introduced with phrases like "it suffices to prove...". Sometimes it is formalized

† A preliminary version of this work appeared in [7].

as an abstraction step. To formally justify such a reduction, one can define a relation among computations, and show that every computation is related to some convenient one. Moreover, that relation must be proved to preserve the property in the opposite direction (i.e., every computation satisfies the property if it is related to one that does).

An example of such a reduction exploits the fact that there is some symmetry in the system being analyzed, as in processes in a ring where any process can initiate an algorithm. This symmetry divides the set of possible computations into equivalence classes, and it is often convenient to consider only one representative of each class (i.e., 'without loss of generality, consider only algorithm initialization for process 0'). Of course, one must show that the property holds equally for all the members of any given equivalence class of computations. Other reductions are based on discarding irrelevant information. For example, one may choose not to consider the effect of concurrent execution in a multitasking system. One may also restrict the attention to a subset of the possible inputs when the rest is not relevant to the property being verified.

Perhaps the most common example of a reduction to convenient computations is the partial order reduction. In a concurrent system, the set of all possible computations includes many permutations of concurrent independent events. System designers usually think of the system's behavior in terms of a subset of the possible permutations of events, which behave in some orderly fashion. For example, it is not unusual to claim that a sequence of local steps in a process can be viewed as if they are executed atomically. By following this intuition, it may be easier to find suitable invariants, than if all the possible interleavings had to be considered. It is important to note that the main difficulty lies in *finding* an inductive invariant, and not in proving it.

In many simple cases, the reduction to convenient computations is not formalized, and therefore left unproven due to its intuitive "obviousness". The convenient computations proof method formalizes this intuition, and allows applying it for cases that are nontrivial. If we attempt to formalize and prove the reduction within a mechanized environment, we might find flaws in our intuitive belief that the convenient computations represent all possible computations with respect to the property being verified. Finding that some computations are *not* represented by any convenient one can have as much practical "debugging" value as finding a counterexample to a property expressed by a logical formula.

Such reductions are not always just an aid in proving some other property. In contexts like sequential consistency of memory models and serializability of database transaction protocols, the behavior of the convenient computations is taken as the correct one by definition, and the reduction itself (e.g., showing that every computation is equivalent to some serial behavior) is the goal of the verification effort.

Methods based on the idea that the operations of a system are only partially ordered [24] have been used for (usually linear) temporal logic based model checking reductions [9, 22, 27]. They have also been applied in theoretical work on general correctness proofs in unbounded domains [14, 16, 23] but without any mechanized implementation—sample proofs have been hand-simulated.

PVS [20, 21, 26] is a general purpose theorem proving tool most often used for program or algorithm verification. Rather than simply showing how to prove an application program using convenient computations in PVS, we attempt to isolate the 'method itself' in PVS, and then show how to apply it to an example. One of the advantages of mechanization is that it makes us prove *everything*, including all the obvious facts, in terms of the proof system's axioms. This, of course, is also one of the main disadvantages of mechanized theorem proving methods. As part of our codification of the convenient computations proof method, we alleviate this problem and reduce the duplication of effort by gradually building up a library of proven theories. Many of the proof obligations, including those that use obvious facts in their justifications, are common to all applications of the approach.

In PVS, a proof environment can be created by packing general definitions and powerful lemmas in general purpose theories. These theories also clarify the new approach and generate the needed proof obligations for any particular application. As will be demonstrated, proof obligations arise as *importing assumptions* when instantiating generic theories, as *type correctness conditions* when defining objects of the provided types, or as *antecedents* (preconditions) of the provided theorems that have the form of a logical implication.

### 1.2. PVS and our proof environment

As noted, PVS is a general-purpose theorem prover for higher-order logic designed to verify and challenge specifications, and is not tailored to any computation model or programming language. It does provide a library of theories about arithmetic, inequalities, sets, and other common mathematical structures (the *prelude*). Decidable fragments of logic are treated by a collection of decision procedures that replace many tedious subproofs. However, it has no built-in concepts of states, operations, or computations. Usual verification examples, like proving an invariant in a transition system, involve the definition of states, initial states, and 'next-state-functions' (transitions) by the user. To prove invariance of a state property P, the following theorem is usually stated and proved for particular definitions of *initial*, *next*, $P$, and $s$:

$$(initial(s) \Rightarrow P(s)) \ \wedge \ (P(s) \Rightarrow P(next(s)))$$

The computations themselves are not mentioned directly. The relationship between the property "in every state, $P$" ($\Box P$ of linear temporal logic) and the induction theorem shown above is neither stated nor justified within the mechanized environment.

Two of the few exceptions to this common usage of PVS are [12, 25]. A related approach is taken in [10] using the Boyer-Moore theorem prover to prove properties of concurrent programs. As in those works, we define an explicit "computation" type that will allow us to specify temporal properties as arbitrary predicates over computations. We also need to deal with explicit computation objects when we prove the reduction to convenient computations, define relations such as computation equivalence, and compute measure (rank) functions on whole computations.

Besides making convenient computations and their use practical in a theorem-proving tool, this paper demonstrates what is involved in packaging such a framework for use by nonexperts in the particular theory, without having to rejustify basic principles. The modular structure of the PVS theories should encourage using parts of the environment whenever convenient computations are natural to the problem statement or proof.

Mechanized verification efforts have been traditionally undertaken within specific domains of application. Typically, a certain algorithm is modelled in a more or less abstract way, and the formal verification is carried out within that computation model. Sometimes the specification (or implementation) to be verified is described using an established methodology like B[1] or VDM[13] for which specialized tools have been developed. These specification methodologies can be embedded or otherwise used when carrying out mechanized verification within general purpose theorem provers. Different ways to do this are described in [19]. In the terms used there, our proof environment is a "deep semantic embedding" of the proof method within the PVS system since the method itself is defined and mechanically proven within the system's logic.

Our work can be seen as "verification middleware", layered between a formally described system and the actual proofs of its properties. In this respect, it can be compared to abstraction steps (though we don't change the description of system states or operations), and to other mechanizations of proof techniques, e.g., Hoare triples in [11].

Most commonly, the user can apply parts of the infrastructure to obtain a useful computation model, by providing only application-dependent details such as a description of the system states, its transitions and added features like fairness assumptions. Whole computations are then defined as sequences of states and operations that satisfy the user-provided restrictions. This built-in generic model is fully supported by our infrastructure, by providing auxiliary theories that facilitate the application of the proof method.

Alternatively, we allow the user to define any computation model, as long as it can support explicit reasoning about computations. It should also be noted that our environment does not require using a specific method to complete proofs of the generated proof obligations, although useful lemmas are provided to aid in proving some of them. The user can choose any proof method or system, including combinations of model checking and inductive proofs, to prove specific properties of the convenient computations.

### 1.3. Overview

In Section 2 we describe—in a theoretical setting—the convenient computations proof method and the added support provided by our proof environment. This includes:

- A *computation model* in which computations are viewed as 'first-class' objects. The model supports an infinite state space and the use of fairness assumptions.
- A notion of *equivalence* of computations based on independence of operations, which can be used as the reduction relation.
- A proof technique for proving the global reduction to convenient computations. This technique uses *measure induction* over well-founded sets. Two possible measure functions which aid the user in completing the proof obligations are also provided:
    - The *matching-intervals* measure, involving the distance between matching pairs of events, used to apply ideas of *virtual atomicity* for sequences of events local to a single process.
    - The *layered* measure, used for computations whose events can be organized into independent layers.
- Examples of families of properties preserved by the computation equivalence relation mentioned above:
    - Terminating state properties (e.g., partial correctness specifications).

- Eventuality of stable state properties (e.g., eventual termination, existence of deadlocks).
- Recurrence and persistence of future path properties (i.e., $GF\phi$ and $FG\phi$ where $\phi$ is a predicate on computation suffixes).

Section 3 presents the building blocks of the proof environment: a set of PVS theories in which the concepts mentioned above are formalized and encapsulated in a modular way. These theories are described as they are applied to a detailed proof of the reduction to convenient computations for a pipelined implementation of insertion sorting. Some familiarity with PVS notation is helpful in understanding parts of this section. Section 4 summarizes the decisions and tradeoffs the user has to make when applying this proof method.

In the examples we have done, the proof environment (infrastructural theories) contains over half of the lines in the PVS specification files and also of the interactive PVS prover commands, taken as a rough indication of the proof effort.

## 2. The theoretic view

### 2.1. The main theorem

This is the core of the convenient computations proof method:

THEOREM 1. *Let $COMPS$ be the set of all computations, $CONV \subseteq COMPS$ a chosen subset (the convenient computations), $R \subseteq COMPS \times COMPS$ a reduction relation, and $p : COMPS \rightarrow bool$ a property of computations such that:*

*1. $p$ holds for all the elements of $CONV$:*

$$\forall c \in CONV \,.\, p(c)$$

*2. $R$ preserves the property $p$:*

$$\forall c, d \in COMPS \,.\, (R(c,d) \wedge p(d)) \Rightarrow p(c)$$

*3. $R$ relates every element of $COMPS$ to some element of $CONV$:*

$$\forall c \in COMPS \,.\, \exists d \in CONV \,.\, R(c,d)$$

*THEN: $p$ holds for all elements of $COMPS$:*

$$\forall c \in COMPS \,.\, p(c)$$

Note the direction of the property preservation in proof obligation 2 above. This theorem and all the other lemmas and theorems presented in this paper have been formally proven within the mechanized proof environment of PVS.

### 2.2. A model for computations

We now describe a sample computation model that can be used to apply the proof method (together with a suitable reduction relation $R$).

Let $STATES$ denote the (possibly infinite) set of states, and $INIT \subseteq STATES$ the subset of initial states. Let $OPS$ be the (possibly infinite) set of operations, with the enabling predicate $EN : STATES \times OPS \rightarrow bool$ and the next-state function $NEXT : STATES \times OPS \rightarrow STATES$. Assume that the set of points in time is $TIME = \mathbb{N}$.

A function $e : TIME \rightarrow STATES \times OPS$ is an execution sequence iff:

- $e(0) = (st, op) \Rightarrow st \in INIT$
- $\forall t \in TIME\ .\ e(t) = (st_t, op_t) \Rightarrow EN(st_t, op_t)$
- $\forall t \in TIME\ .\ [(e(t) = (st_t, op_t) \ \wedge e(t+1) = (st_{t+1}, op_{t+1})) \Rightarrow st_{t+1} = NEXT(st_t, op_t)]$

Sometimes we need to impose additional global restrictions on the execution sequences to be considered, like various notions of *fairness* [2, 6, 17] and/or finiteness assumptions. We define all these constraints as a predicate called $RELEVANT$. The legal computations are all the execution sequences that satisfy the $RELEVANT$ predicate.

Since we define a computation as a sequence, temporal operators can be expressed by quantifying over time variables (whose values are just indices into the execution sequence, and should not be confused with real-time clocks). Path quantification is also possible, by quantifying over computations in the higher-order logic of PVS. Thus we are not limited to linear temporal logic properties. For example, we write a $CTL^*$ path formula like "GEFp" as a predicate on computations:

$$\begin{aligned} GEFp(c) = \forall t.\ \exists c'.\quad & (\forall t_p < t.\ c'(t_p) = c(t_p)) \\ & \wedge (\exists t_f \geq t.\ p(c'(t_f))) \end{aligned}$$

or in words, "for every time point $t$ there exists a computation $c'$ that is identical to $c$ before time $t$ and satisfies p at time $t$ or later".

### 2.3. A reduction relation

As an example of a suitable reduction relation $R$, we proceed to define a notion of computation equivalence, which we call *swap-equivalence*. It is similar to the equivalence relation defined in [14].

This reduction is based on the equivalence between different linearizations of the same partially ordered set of events. The only difference between two such linearizations is the order in which independent events occur. When the user proves the reduction from one computation to another, using the swap-equivalence relation, several pairs of operations must be proved functionally independent in certain states.

#### 2.3.1. *Functional operation independence*

Two operations $op_1$ and $op_2$ are *functionally independent* at state $s$ iff:

- neither operation modifies the other's enabledness at the state $s$:

$$EN(s, op_1) \Rightarrow (EN(s, op_2) \Leftrightarrow EN(NEXT(s, op_1), op_2))$$
$$EN(s, op_2) \Rightarrow (EN(s, op_1) \Leftrightarrow EN(NEXT(s, op_2), op_1))$$

- $op_1$ and $op_2$ are commutative at $s$, i.e., the result of performing them in either order must be the same.

$$EN(s, op_1) \wedge EN(s, op_2) \Rightarrow$$
$$NEXT(NEXT(s, op_1), op_2) = NEXT(NEXT(s, op_2), op_1)$$

Note that this independence relation takes into account the current state. It allows us to consider two operations as commutable in some states, even when in other states they are dependent.

#### 2.3.2. *User-defined conditional operation independence*

Instead of letting the user repeat the functional independence proof for two operations every time a swap is considered, we follow [14, 15] by choosing a *conditional independence* relation

$$I \subseteq OPS \times OPS \times STATES$$

that defines the independence of two operations as a function of the current state. Such a chosen predicate must be symmetric and must imply functional independence as defined above.

There is a tradeoff involved when choosing this relation. The functional independence relation is a trivial candidate, and it is a maximal one. But it is not a practical choice, since every time we consider a pair of operations, we need to repeat the proof of independence from its definition. It is better to define a simpler independence relation and to prove *separately* from the rest of the justification of the reduction that it implies functional independence. This user-defined relation can hold for fewer cases than strictly possible, if only some of them are needed to show equivalence between computations. This can make it easier to complete the (separate) proof of functional independence.

2.3.3. *One-swap-equivalence*

Once we pick a suitable $I$, we define that two computations $c$ and $d$ are *one-swap-equivalent* ($c \equiv_{1swp} d$) iff they are different, and there exists a time point $t$ s.t. $c$ and $d$ are equal everywhere, except for the swapping of two **independent** operations at $t$ and $t+1$, and (possibly) for the intermediate state at time point $t+1$.

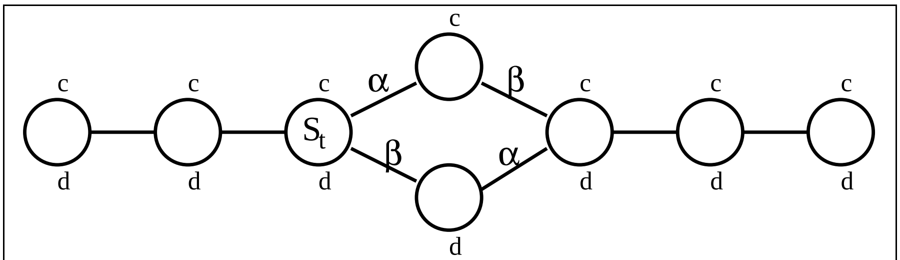


*Figure 1.* Computations c,d are one-swap-equiv (and operations $\alpha$ and $\beta$ are independent at $S_t$)

Formally: $c \equiv_{1swp} d \Leftrightarrow \quad c \neq d \wedge$

$$
\begin{aligned}
\exists t \in \mathit{TIME} \;.\quad & [\ \forall t' \in \mathit{TIME}\ \ .\ (t' < t \vee t' > t+1) \Rightarrow c(t') = d(t')\ ] \\
& \wedge\ [\ c(t) = (s_t, \alpha) \wedge c(t+1) = (\mathit{NEXT}(s_t, \alpha), \beta) \Rightarrow \\
& \quad\ d(t) = (s_t, \beta) \wedge d(t+1) = (\mathit{NEXT}(s_t, \beta), \alpha)\ ] \\
& \wedge\ I(\alpha, \beta, s_t)
\end{aligned}
$$

It should be noted that if a legal computation (i.e., a relevant execution sequence) is selected, and two independent consecutive operations are swapped, we may get an execution sequence that is not relevant. This difficulty arises because in our proof environment the definition of the relevant execution sequences is arbitrarily chosen by the user. For example, if we restrict ourselves to computations in which the length of some queue is bounded by a constant, then swapping consecutive "get" and "put" operations might lead to a non-relevant execution sequence, if the queue's length was equal to that bound before the "get" took place. Since we use operation swapping to show equivalence to convenient computations, we will probably apply swaps leading to shorter queues, thus remaining within the set of legal computations. For the fairness notion supplied by our proof environment this is also not a problem. Since one swapping only affects a finite prefix of the computation, fairness will always be preserved by single swaps.

In our infrastructure, one-swap-equivalence is defined on legal computations, not on execution sequences. Therefore, every time a user applies a swap to a legal computation and wants to prove that the result is one-swap-equivalent to the original, the proof environment will automatically generate a proof obligation asking to prove that the resulting execution sequence indeed complies with the relevance restriction.

Let the predicate $swap\_at\_swt(c, d, t)$ hold iff the computation $d$ is the result of swapping independent operations in the computation $c$, at time $t$. The predicate $can\_swap : COMPS \times TIME \rightarrow bool$ is defined as: $\forall c \in COMPS . \forall t \in TIME . can\_swap(c, t) \Leftrightarrow \exists d \in COMPS . swap\_at\_swt(c, d, t)$. This is useful for expressing other predicates in the infrastructure, as in Section 2.4.2.

### 2.3.4. *Swap-equivalence*

We now take the reflexive-transitive closure of the one-swap-equivalence relation. Two computations are *swap-equivalent* if they are equal, or if they differ only by the ordering of a finite number of independent operations (See Figure 2 for an example). Formally:

$$c \equiv_{swp} d \Leftrightarrow \exists n \in \mathbb{N}, \exists c_0, c_1, ..., c_n \in COMPS . c = c_0 \wedge d = c_n \wedge \\ \forall i \in 0 \ldots n-1. \ one\text{-}swap\text{-}equivalent(c_i, c_{i+1})$$

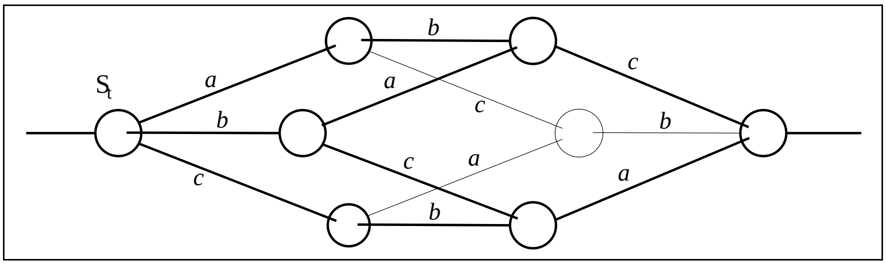


*Figure 2.* Some interleavings of operations $a, b, c$ (independent at $s_t$). $abc \equiv_{1swp} bac \equiv_{1swp} bca \equiv_{1swp} cba$, therefore $abc \equiv_{swp} cba$

Swap-equivalence is similar to *trace equivalence* for finite traces in [18] and the finite string equivalence in [16], but with a state-conditional independence relation instead of a fixed $I \subseteq OPS \times OPS$.

## 2.4. Proving the reduction using well-founded induction

We now show a technique that can be used to prove the global reduction to convenient computations (the third requirement of the proof method, as seen in Theorem 1). It is suitable for *transitive* reduction relations where some convenient computation can be reached from every computation by finitely many applications of the reduction relation; the swap-equivalence defined in the previous section satisfies these conditions.

The technique proposed is based on well-founded induction. We need to choose a well-founded domain $(M, <)$, and a *measure* function $m : COMPS \rightarrow M$. All that is needed to prove the global reduction is that every nonconvenient computation is reducible to some computation with a strictly smaller measure value:

LEMMA 2. *Let R be a transitive reduction relation.*
*If*

$$\forall c \in (COMPS \setminus CONV).\ \exists d \in COMPS\ .\ m(d) < m(c) \wedge R(c, d)$$

*Then*

$$\forall c \in COMPS\ .\ \exists d \in CONV\ .\ R(c, d)$$

In most cases a standard approach can be taken to define the set of convenient computations and the appropriate measure function. Typically, the measure function is related to the number of "operation swaps" that need to be done to arrive at a convenient computation. If the choice of measure function is done on this basis, the proof obligations generated will be easy (if not trivial) to discharge.

Below we describe two sample measure functions provided in the proof environment, along with some added support to make them easier to use.

2.4.1. *The matching intervals measure*

Sometimes we might want to consider as "convenient" those computations in which certain pairs of events (computation steps) occur consecutively. One common example could be sending and receiving a value over an empty communication channel. Another example is viewing some series of events local to certain processors as "virtually atomic". In [14] the convenient computations method is applied (manually) to the sequential consistency problem. The measure function used there involves the length of intervals of selected events (computation steps). The measure value is lowered by moving unrelated events out of the interval until all the selected events happen consecutively. We provide below a simpler version which can be applied to achieve the same effect. An *interval* is defined as a pair of points in time: $i = (t_1, t_2),\ t_1 < t_2$, and its distance (length) is $distance(i) = t_2 - t_1 - 1$ (An interval of consecutive events has zero distance). We intuitively call an interval $(t_1, t_2)$ *matching* if the operations that occur at $t_1$ and $t_2$ should occur consecutively for the computation to be convenient. The set of matching intervals of the computation $c$ is defined by the user-provided predicate $match(c, i)$. The *matching intervals measure* for a computation is defined as the sum of all the distances of its matching intervals:

$$m(c) = \sum_{\{i|match(c,i)\}} distance(i)$$

The minimum value is attained when all the matching intervals have zero distance. In a reasonable application of the method, the definition of the matching intervals should make it easy to prove that nonconvenient computations have a nonzero measure.

The *match* predicate must satisfy the following requirements:

– Every computation has finitely many matching intervals. This requirement is necessary to make the measure finite. (An alternative would be to require that the set of nonzero-distance matching intervals be finite, and then sum distances only over that set.)

– Swappable (i.e., independent consecutive) operations cannot appear at the ends of a matching interval:

$$can\_swap(c, t) \Rightarrow \neg match(c, (t, t + 1)).$$

This requirement is to avoid negative-distance intervals, which would otherwise be implied by the next requirement.

– One-swap-equivalent computations have the same matching intervals, except for those intervals whose end-points are affected by the swap. If computation $d$ is the result of a swap in computation $c$ at time $t$, any interval end-point at time $t$ is moved to $t + 1$ and viceversa. Formally, (for one of the four combinations): $match(c, (t, t_2)) \Rightarrow match(d, (t+1, t_2))$. Without this requirement, nothing general could be said about the influence of a swap on the value of the measure function.

– No two matching intervals start in the same time point and no two end together.

$$(match(c, (t_1, t_2)) \wedge match(c, (t_3, t_4))) \Rightarrow (t_1 = t_3 \Leftrightarrow t_2 = t_4)$$

The user's goal is to find, given a nonconvenient computation, one point in time where a swap can be done, and that will yield a lower measure value. We can reason about the measure once and for all, saving the user tedious arithmetical reasoning, if we characterize all the cases that will lead to a measure decrease by a single swap. For example, an interval starting at time $t$ can be shortened by swapping the operations executed at time points $t$ and $t+1$. If no interval ended at time $t$ before doing the swap, this may lead to a decrease in the measure. However, if another interval also started at time $t+1$, then that swapping will make it actually longer, leaving the total measure value unchanged. The proof environment defines and justifies a precise condition that, along with the requirements above, characterizes all the situations leading to a measure decrease.

### 2.4.2. *The layered measure*

Another way of thinking of convenient computations of a program is to define ordered phases or layers of execution [4, 5]. Each event is associated with a layer. If the layers execute sequentially, without events from a layer getting mixed with events from an earlier layer, the computation is considered convenient. Examples where this approach seems natural are programs with communication-closed layers and distributed snapshot algorithms [3]. In contrast to some of those previous works, however, we do not focus on syntactic layers: the same program instruction occurring more than once might produce events belonging to different execution layers.

Given a natural number *lastlayer* and function

$$layer : COMPS \times TIME \rightarrow 0..lastlayer$$

The *layered measure* function is defined as follows:

$$m(c) = \sum_{l=0}^{lastlayer-1} \left( \sum_{\{t_0 | layer(c,\ t_0)=l\}} \mid \{t | t < t_0 \wedge layer(c,\ t) > l\} \mid \right)$$

For each layer, and for each event associated with it, we count the number of preceding events that belong to a (strictly) later layer (therefore, events associated with lastlayer don't contribute to this summation). The measure value of a computation is the sum of those counts. Clearly, computations with a zero measure value should be convenient in an application since no event is preceded by an event from a later layer.

Because the measure function must have a finite value, the program must be divided into a finite number of layers, where all layers (except for the last one) must eventually terminate, i.e., for each computation and for each layer except for the last one, there is a time after which all the events belong to other layers.

The *layer* function must be preserved by one-swap-equivalence: If computations $c$ and $d$ differ only by an operation swap at time $t_s$, then

$$layer(c,\ t_s) = layer(d,\ t_s+1) \land$$
$$layer(c,\ t_s+1) = layer(d,\ t_s) \land$$
$$\forall t.(t < t_s \ \lor\ t > (t_s+1)) \Rightarrow layer(c,\ t) = layer(d,\ t)$$

In other words, the layer number for an event (an instance of an operation) should not be affected by the swapping. This will be trivial to prove for any reasonable definition of the *layer* function.

LEMMA 3. (Sufficient condition for measure decrease).

*If:* $\forall c \in COMPS\ .\ \forall t \in\ TIME\ .$
$\quad layer(c,\ t) > layer(c,\ t+1) \Rightarrow can\_swap(c, t)$

*Then:* $\forall c \in\ COMPS\ .\ m(c) > 0 \Rightarrow$
$\quad \exists d \in\ COMPS\ .\ c \equiv_{swp} d\ \land\ m(d) < m(c)$

The predicate $can\_swap(c, t)$ was defined in Section 2.3.3.

This seemingly strong sufficient condition will be easy to verify (though it might involve a lengthy case analysis), if layering is appropriate for the application.

### 2.5. Families of properties preserved by a reduction

To verify a property of computations by using a reduction to convenient computations, it is necessary to prove that the reduction preserves the specific property. Sometimes, certain reductions preserve whole families of properties. If these families can be specified and their preservation proved once (in the proof environment), then it is enough to show that an individual property belongs to one such family to conclude that it is preserved by the reduction. A simple lemma helps us prove preservation of families of properties by the swap equivalence reduction:

LEMMA 4. *Any property preserved by one-swap-equivalence is also preserved by the swap-equivalence relation.*

*If:* $\forall c, d \in COMPS\ .\ (c \equiv_{1swp} d \land d \models p) \Rightarrow c \models p$
*Then:* $\forall c, d \in COMPS\ .\ (c \equiv_{swp} d\ \land d \models p) \Rightarrow c \models p$

**Proof:** The definition of swap-equivalence involves a finite chain of one-swap-equivalences. The lemma is then easily proved by induction on the length of this chain. □

We now define various families of properties preserved by swap-equivalence.

2.5.1. *Terminating-state properties*

We say that a state is *terminating* if only self-transitions are enabled on it, i.e., the next-state function applied on every operation enabled there yields the same state. Given a state predicate $p$, we can define the corresponding property of computations $terminating\_state\_property(p)$:

$$\forall c \in COMPS \ . \ c \models terminating\_state\_property(p) \Leftrightarrow \\ \forall t. \ (c(t) = (st, op) \wedge is\_terminating\_state(st)) \Rightarrow st \models p$$

Common examples of terminating-state properties are partial correctness specifications of programs which compute an input-output transformation, distributed agreement protocols, etc.

LEMMA 5. *Terminating-state properties are invariant under the swap-equivalence relation.*

**Proof:** By Lemma 4, it suffices to show that these properties are preserved by one-swap-equivalence. Let $c \equiv_{1swp} d$. Without loss of generality, assume that $c$ violates a terminating-state property while $d$ satisfies it. In $c$, there exists a terminating state $s$ that violates the state property for which the terminating-state property is defined. Since the only enabled operations in $c$ at $s$ are idling ones, all the suffix of $c$ following $s$ remains at the same state $s$. Two one-swap-equivalent computations converge to the same state after executing the two independent operations involved in the swap, so $c$ and $d$ share a common suffix. Therefore, $d$ also has a suffix which consists of idling on $s$, so $d$ also violates the terminating-state property, a contradiction to our assumption. □

2.5.2. *Eventuality of stable state properties, and the invariance of their negation.*

A state property is stable if the fact that it holds in a given state implies that it continues to hold along the computation. For a stable state property $p$, the *eventuality* of $p$ can be defined by the path formula $Fp$ ($Fp$ is a property of computations that requires $p$ to hold in some state of the computation). Common examples of this kind of property are termination of programs, and the existence of deadlocks. This family of properties, together with the previous one, provides support for the verification of total correctness.

LEMMA 6. *The eventuality of stable state properties is preserved by the swap-equivalence relation.*

**Proof:** The proof follows the same lines as that of the previous lemma. □

LEMMA 7. *The invariance of the negation of stable state properties is preserved by the swap-equivalence relation.*

**Proof:** By Lemma 4, it suffices to show preservation by one-swap-equivalence. Let $c \equiv_{1swp} d$, and $d \models G\neg p$. If we assume $c \not\models G\neg p$, then $c \models \neg G\neg p$, or $c \models Fp$. Since one-swap-equivalence is symmetric, by Lemma 6 we have $d \models Fp$, or $d \not\models G\neg p$, a contradiction. □

In Section 4.2 we show an example of a property that belongs to this family.

2.5.3. *Recurrence and persistence of future path properties*
A *future* path property is a property of suffixes of computations.

LEMMA 8. *Given a future path property $\phi$, the computation properties $GF\phi$ (infinitely often $\phi$) and $FG\phi$ (eventually forever $\phi$) are preserved by the swap-equivalence relation.*

**Proof:** As before, by Lemma 4, it suffices to show that these properties are preserved by one-swap-equivalence. If a computation $d$ satisfies $GF\phi$, it has infinitely many suffixes that satisfy $\phi$. As seen in the previous lemmas, any other computation $c$ one-swap-equivalent to $d$ will also have infinitely many $\phi-$suffixes, so $c$ will also satisfy $GF\phi$. This proves preservation for recurrence of future path properties. We now turn to preservation of persistence: let $d \models FG\phi$, and assume $c \not\models FG\phi$. Therefore, $c$ satisfies $GF\neg\phi$, which is the recurrence of a future path property. If $c \equiv_{1swp} d$, then $d \equiv_{1swp} c$, and as we just proved, this implies $d \models GF\neg\phi$, or $d \not\models FG\phi$ – a contradiction. □

## 3. The proof environment and its use.

### 3.1. General description of the provided infrastructure

In the previous section we presented the proof method's theoretic elements without dealing with implementation details expressed in PVS, and without showing how to apply these elements in a particular proof.

The proof environment consists of a library of parameterized PVS theories (modules with definitions and proven theorems). These theories can be selectively instantiated (*imported*) by the user, who has to provide application-dependent information. The hierarchical structure in which the theories are organized is shown in Figure 3. The "Proof Method" theories enunciate and justify the reduction theorem (`convenient_computations`) and the proof technique based on measure induction (`subset_reachability`). To use them, a notion of computation and a reduction relation are needed. The "Computation Model" and the "Computation Equivalence" theories address these needs. The former provide a skeleton that can be filled with application-dependent details to define the set of legal computations, and the latter defines swap-equivalence based on user-provided operation independence information. Additional support theories include two default measure functions defined in `matching_intervals_measure` and `layered_measure`, families of properties proved to be preserved by swap-equivalence (such as in `terminating_state_properties`), and other theories with useful lemmas, which are not shown in Figure 3 (e.g., `swap_fairness`, where it is proved that swap-equivalence preserves fairness, and also `linear_progress`, which assists in the proof of liveness by taking advantage of the fairness assumptions).

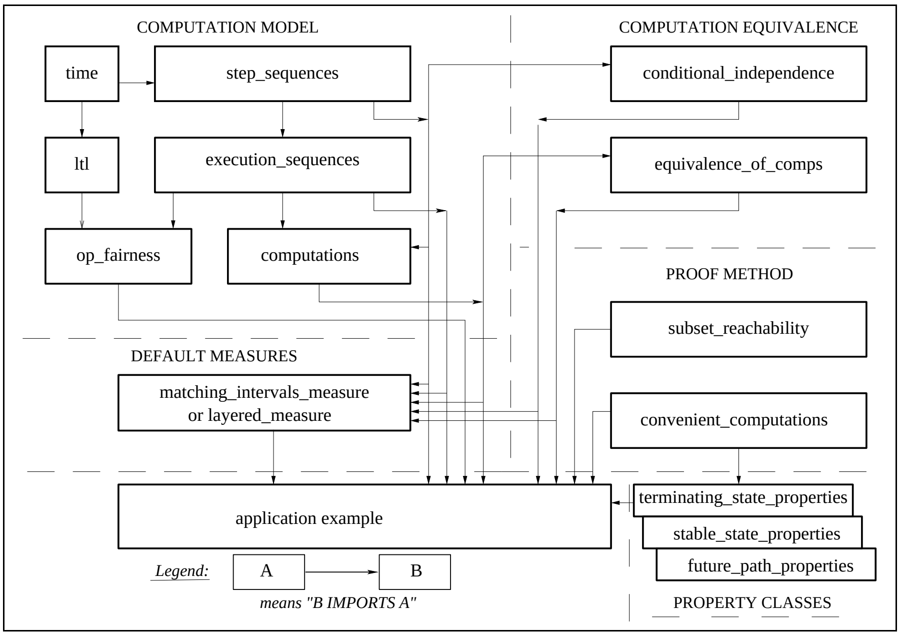


*Figure 3.* The hierarchy of theories.

Here is a "roadmap" of the main steps to be taken when applying the provided proof environment to prove desired properties for all the computations of a given system.

1. **Building a computation model:** The user must define the type of all the legal computations. In principle it can be any type, but to make use of the provided reduction relation and predefined measure functions, this type must be defined through the use of the supplied computation model theories. The information supplied by the user in this step includes the definition of the system states, the operations and their enabling predicates and next-state functions, the set of initial states, and any additional constraints such as fairness assumptions.

2. **Defining a reduction relation:** The reduction relation, as required by the main reduction theorem (Section 2.1), can be any binary relation between computations. However, in order to exploit the proof technique based on measure functions, the reduction has to be a preorder. If the user wants to use the provided notion of *swap-equivalence* as the reduction relation, the conditional independence between operations must be defined first.

3. **Proving the global reduction:** The user must define the set of the convenient computations, and then prove that every non-convenient computation is related to some convenient one by the reduction relation. When using the measure functions technique, the user must define the measure function and prove that every non-convenient computation is related to one with a strictly smaller measure (though not necessarily a convenient computation).

   a) **Using the matching intervals measure:** The user must define the *match* predicate and prove that it meets all the requirements listed in Section 2.4.1. Then a sufficient condition can be used to prove that in every non-convenient computation there is a point in time where an operation swap can lower the measure.

   b) **Using the layered measure:** The user must define the number of layers in the program and the *layer* function and prove that it meets the requirements listed in Section 2.4.2. The global reduction is then proved by another sufficient condition.

   In both cases, the provided sufficient conditions shield the user from directly reasoning about the measure function.

4. **Proving properties:** In addition to proving the global reduction, if the user wants to prove that a specific property holds for all possible computations, then the reduction must be proven to preserve the property and the property must be verified for the convenient computations.

### 3.2. Application example walk-through

We now present the PVS theories as they are incorporated into an example. We present the PVS code (sometimes abridged) in framed boxes. The application example's code appears in numbered boxes with thin frames, and the code for the infrastructural theories making up the proof environment appears with darker frames. The full PVS files

(theories and proofs) are available from
`http://www.cs.technion.ac.il/~ssdl/pub/conv_PVS` .

Our example is a representative of the pipelined processing paradigm. We choose as the convenient computations those that execute "one wave at a time", i.e., in which a new input is entered into the pipeline only when all the operations related to the previous inputs have been finished.

The program is a pipelined insertion-sort algorithm taken from [16] in which the buffers between the processors can hold a single value. Each processor takes its input, compares it with a previously stored local value, then sends the maximum between them to the next processor in the pipeline and stores the minimum locally (See Figure 4). If there was no previously stored value, the current input is just stored locally and no value is forwarded. For simplicity, we model all these steps as a single atomic operation. A more refined model could have been used and proven by applying the same layering technique.

To understand why it is complicated to prove that the algorithm correctly sorts the inserted values without the convenient computations approach, consider a general computation. In a typical state, the $k$ first processors already have a stored value, and some of them have pending inputs. There could be several such processors whose successor has an empty buffer, so many different operations would be enabled in such a state. To verify the sorting algorithm we need a general invariant, much harder to find than the one needed if we only have to consider convenient computations in which there is at most one possible continuation at a time.

In Box 1 we see that the example is parametric over the type of the elements being sorted, their total-ordering relation, the number of input values (and of processors) `NUM`, and an array from 1 to `NUM` holding those values:

1

```
pipeline_sort[V: NONEMPTY_TYPE, <= : (total_order?[V]),
           NUM: posint, inarr: [subrange(1,NUM) -> V] ]: THEORY
BEGIN
  procs : TYPE = subrange(1,NUM+1)
```

The processors' indices also range from 1 up to `NUM`. We added a "dummy" processor (NUM+1) so that the last processor will not need special treatment.

Following the road map described in the previous subsection, the first step is to define a model of computation, as described in Section 2.2. In Box 2, we start by defining the type of the system states. A state includes an array of processor states, and a counter that indexes into the (constant) input array `inarr`, by pointing to the next input to be entered. Each processor state includes a locally stored value, an input buffer with room for a single value, and an integer (`input_layer`) that holds the "wave number" associated with that input: to make it simpler to compute the *layer* value for an event, we will want each processor to keep track of which "wave" caused a value to reach a given processor. This number is taken from the global counter when inputting a new value into the first processor in the pipeline, and is copied to the next processor when a value is propagated forward, regardless of the result of the comparison between the input and the locally stored value.

Due to our choice of atomicity for the processors' transitions, there is only one operation for each processor, so we can use the processor number to identify operations. As special cases, the operation that enters a fresh input into the pipeline is identified by the number 0, and the *idling* transition by NUM+1:

```
                                                             2
procstates: NONEMPTY_TYPE =
      [# has_input   : bool,   % flag pending input
         input       : V,      % the input buffer.
         input_layer : upto(NUM), % added for the proof
         has_val     : bool,   % flag locally held value
         val         : V  #]   % the value held by this proc.
states: NONEMPTY_TYPE = [# cnt : subrange(1,NUM+1),
                           pr  : [procs->procstates] #]
ops   : TYPE = upto(NUM+1)  % 0: new input, NUM+1: idle
```

A typical system state is shown in Figure 4.

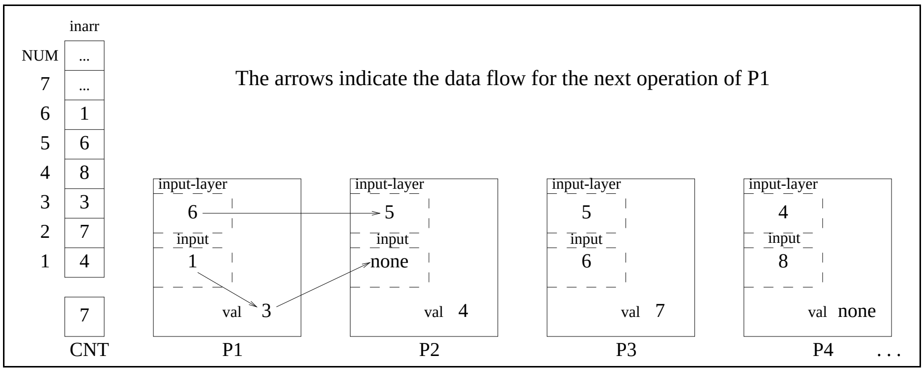


*Figure 4.* A typical system state of Pipelined Insertion Sorting.

The first theory from the proof environment we present is `time`. This small theory defines the type `time`, whose values are the natural numbers. It also defines the time intervals used by the matching intervals measure:

```
time: THEORY
BEGIN
  time : TYPE = nat
  intervals : TYPE = [# t1: time, t2:{t:time|t>t1} #]
  distance(i: intervals): nat = i`t2 - i`t1 - 1
END time
```

The generic theory `step_sequences` is parametric on the states and operations' types. It makes use of `time` to define sequences of state-operation pairs.

```
step_sequences [states,ops: NONEMPTY_TYPE]: THEORY
BEGIN
  steps: TYPE = [# st : states, op : ops #]
IMPORTING time
  step_seqs : TYPE = [time->steps]
  initial_conditions  : TYPE = [states     -> boolean]
  enabling_conditions : TYPE = [states,ops -> bool   ]
  next_state_functions: TYPE = [states,ops -> states ]
END step_sequences
```

We now import `step_sequences` and define our example's transition system elements:

```
                                                              3
IMPORTING step_sequences[states, ops]
initial?(st: states): bool = st`cnt = 1 AND
    FORALL(p: procs): LET p_st=st`pr(p) IN
          NOT p_st`has_input AND NOT p_st`has_val
          AND p_st`input_layer=0
en?(st: states, op: ops): bool = TABLE
  %--------%------------------------------------------------%%
  |op=0    | st`cnt <= NUM AND NOT st`pr(1)`has_input       ||
  %--------%------------------------------------------------%%
  |op=NUM+1| TRUE                                           ||
  %--------%------------------------------------------------%%
  |ELSE    | st`pr(op)`has_input AND
            (st`pr(op)`has_val => NOT st`pr(1+op)`has_input)||
  %--------%------------------------------------------------%%
  ENDTABLE
next(st: states, op: ops): states =
  IF NOT en?(st,op) THEN st
  ELSIF op=0 THEN st WITH [       % insert new input
    `cnt := cnt(st)+1,
    `pr(1)`has_input   := TRUE,
    `pr(1)`input       := inarr(st`cnt),
    `pr(1)`input_layer := st`cnt ]   % continues next page...
```

```
  ELSIF op=NUM+1 THEN st          % idling operation           4
  ELSIF NOT st`pr(op)`has_val THEN st WITH [ % processor "op"
    `pr(op)`has_input  := FALSE,        % stores local value
    `pr(op)`has_val    := TRUE,
    `pr(op)`val        := st`pr(op)`input ]
  ELSE st WITH [
    `pr(op)`has_input  := FALSE,        % store the minimum
    `pr(op)`val        := IF st`pr(op)`input <= st`pr(op)`val
                          THEN st`pr(op)`input
                          ELSE st`pr(op)`val ENDIF,
    `pr(op+1)`has_input := TRUE,        % forward the maximum
    `pr(op+1)`input     := IF st`pr(op)`input <= st`pr(op)`val
                           THEN st`pr(op)`val
                           ELSE st`pr(op)`input ENDIF,
    `pr(op+1)`input_layer := st`pr(op)`input_layer ]
  ENDIF
```

The following theory defines the legal execution sequences of any transition system:

```
execution_sequences[
   states: NONEMPTY_TYPE, ops: NONEMPTY_TYPE,
   (IMPORTING step_sequences[states,ops])
   initial?: initial_conditions, en?: enabling_conditions,
   next: next_state_functions   ]: THEORY
BEGIN
  well_built?(seq: step_seqs): bool = initial?(seq(0)`st) AND
    FORALL(t): en?( seq(t)`st, seq(t)`op ) AND
               seq(t+1)`st = next( seq(t)`st, seq(t)`op )
  execution_sequences: TYPE = {seq: step_seqs|well_built?(seq)}
END execution_sequences
```

Here we can see that as we combine our definitions with those from the infrastructural theories, their list of parameters gets longer. In the rest of this section we will explain in detail the parameters as they appear for the first time, and only mention briefly those that were previously described. We import `execution_sequences` to define the possible runs of our computation model:

```
IMPORTING execution_sequences[states,ops,initial?,en?,next]    5
```

Since the next-state function includes an idling (i.e., stuttering) transition (that of operation NUM+1, for every state), there are execution sequences that stall forever, even in states where other (non-idling) operations are enabled. Such computations are not interesting, so we will rule them out as being *not relevant*. Clearly, we need to introduce some weak fairness assumption. To express the fairness assumptions, we use the infrastructural theory `op_fairness` which in turn uses LTL operators defined in the `ltl` theory. These are summarized in Appendix A. This semantic encoding of the LTL operators is based on the definition of path formulas as the type `point_pred`. If we quantified over computation variables, we could also use this type to represent any branching-time temporal formula. This theory takes as its single parameter a definition of a computation step, so it is not restricted to our model of computation. In another application for this theory, one may consider computations as sequences of states, without explicit reference to the operations chosen at each step. The LTL operators are introduced as `point_pred` transformers. The boolean operators are "lifted" to this type by automatic application of conversion functions by the PVS system. The `ltl` theory provides some lemmas which can save repeated temporal reasoning in common situations. The "useful lemma" shown in the appendix can be applied to show that if an operation will eventually be taken, it will eventually be taken *for the first time*[1].

The notion of fairness used is called "operation-fairness". It is similar to the notion used in [17], but with a small difference. There, computations are sequences of states, and any number of different transitions must be considered *taken* at the same step from state $s$ to state $s'$ if the pair $(s, s')$ belongs to their transition relation. In our computation model the chosen operation is an explicit part of the computation step, so we only consider "taken" the operation that actually appears in the state-operation pair.

We can now complete the definition of our computation model, by defining the sets of operations that should get *weakly fair* treatment (every pipeline operation except for idling) and strongly fair treatment (none in our case). We use these sets as parameters to the `op_fairness` theory, which defines when an execution sequence is *fair*. We choose fair execution sequences as the relevant ones when instantiating the `computations` theory.

6
```
weakly_fair_ops?(op: ops): bool = op <= NUM
strongly_fair_ops?(op: ops): bool = FALSE
IMPORTING op_fairness[states,ops,initial?,en?,next,
                      weakly_fair_ops?, strongly_fair_ops? ]
relevant?(seq: execution_sequences): bool = is_fair?(seq)
IMPORTING computations[states,ops,initial?,en?,next,relevant?]
```

The theory `computations` defines the `comps` type, whose objects will be our legal computations:

[1] This is similar to the meaning of the *leads-to-via* operator that appears in [12].

```
computations[states, ops, initial?, en?, next,
(IMPORTING execution_sequences[states,ops,initial?,en?,next])
  relevant? : pred[execution_sequences] ] : THEORY
BEGIN
  comps: TYPE = {exec: execution_sequences| relevant?(exec)}
END computations
```

We now move on to Step 2, where we define the reduction relation. We start by defining our own independence relation. Two operations are clearly independent if they occur in two pipeline processors that are not neighbors. Moreover, the idling operation is independent of any other one:

7

```
indep?(op1,op2: ops, s: states): bool =
    op1>op2+1 OR op2>op1+1 OR op1=NUM+1 OR op2=NUM+1
```

From now on, we omit showing the text of some PVS theories, if it doesn't add any insight over what was defined in Section 2.

The theory `conditional_independence` formalizes the functional independence relation defined in Section 2.3.1, and defines the type `independence_relations`, which restricts a user-provided independence relation to one that implies functional independence, and is symmetric. We now proceed to import this theory, and then the theory defining the reduction relation:

8

```
IMPORTING conditional_independence[
              states, ops, initial?, en?, next]
JUDGEMENT indep? HAS_TYPE independence_relations
IMPORTING equivalence_of_comps[states, ops, initial?, en?,
                               next, relevant?, indep?]
```

The `JUDGEMENT` is a theorem that we must prove, which tells the type checker of PVS not to generate a "TCC" (type correctness condition) every time we use our `indep?` relation where `independence_relations` are expected. The theory `equivalence_of_comps` defines the auxiliary functions and predicates described in Section 2.3.3, and uses them to define the equivalence relations `one_swap_equiv?` and `swap_equiv?` mentioned in Sections 2.3.3 and 2.3.4.

We now arrive at Step 3 in the road map, where we define the convenient computations, and set out to prove the global reduction. We define the convenient computations as those in which no new inputs are fed into the pipeline while there are processors with a pending input in their buffers:

9

```
conv?(c: comps): bool =
  FORALL(t: time): ( c(t)`op=0 =>
    FORALL(p: procs|p <= NUM): NOT c(t)`st`pr(p)`has_input )
```

We choose to apply the measure functions technique to prove the reduction. The PVS theory `layered_measure` from the environment that defines the layered measure (described in Section 2.4.2) is summarized in Appendix B, showing only the assumptions, the measure function and the sufficient condition that encapsulates all the arithmetic reasoning involved in showing a measure decrease. To instantiate this theory, we first need to define the layerization of the events in a computation. The number of layers is `NUM+1` since every event related to an input belongs to a different layer, and every event related to the idling operation belongs to the last layer. The function `layer` computes an event's layer based on the kind of operation involved and on the information available in the current state:

10

```
lastlayer: nat = NUM+1
layer(c: comps, t: time): upto(lastlayer) =
  LET st=c(t)`st, op=c(t)`op IN
      IF    op=NUM+1 THEN lastlayer  % IDLE events
      ELSIF op=0 THEN st`cnt         % new inputs
      ELSE  st`pr(op)`input_layer ENDIF
IMPORTING layered_measure[states,ops,initial?,en?,next,
                          relevant?,indep?,lastlayer,layer]
```

The proof of the assumption `layer_preservation`, that layer numbers are not affected by swapping, as expected, is quite straightforward due to the choice of the layerization function. In proving the assumption `can_order_layers` we have to show that doing a swap yields a legal computation, i.e., a fair one. The proof environment contains an auxiliary lemma to deal with this in a general way:

```
swap_fairness[states,ops,... ,weakly_fair_ops?...]:THEORY
BEGIN
   swap_preserves_fairness: LEMMA
       FORALL( c: execution_sequences,
               t: time|can_swap_op?(c,t) ):
           LET d=swap_op(c,t) IN
               is_fair?(c) IMPLIES is_fair?(d)
END swap_fairness
```

Proving the final assumption `F_G_different_layers` (every layer has finitely many events) is technically more complicated, involving several lemmas to show that the layer values grow monotonically and that they are bounded by the input counter. After discharging the three assumptions of the `layered_measure` theory, its sufficient condition for measure decrease can be applied (`post_before_prev_has_smaller_equiv`). In other words, to complete the proof of the reduction we must show that every non-convenient computation has a pair of (not necessarily consecutive) events, the earlier one having a bigger layer value:

```
                                                              11
not_conv_has_post_before_prev: LEMMA
  FORALL (c: comps):
    NOT conv?(c) IMPLIES
    EXISTS(t1,t2: time): t2 > t1 AND layer(c,t1) > layer(c,t2)
```

Proving this lemma involves showing that in the pipelined sorting example, every time an operation is enabled it will stay enabled unless it is taken. Then, by fairness[2], it will eventually be taken.

Having proved that any non-convenient computation is equivalent to one with a smaller measure value, the following theory can be applied to prove the global reduction (Lemma 2):

```
subset_reachability [T: TYPE, in_subset?: pred[T],
                     M: TYPE, m: [T -> M],
                     <: (well_founded?[M]),
                     path_to?: (preorder?[T]) ]: THEORY
BEGIN
  measure_drop: bool =
    FORALL(c: T): NOT in_subset?(c) IMPLIES
                  EXISTS(d: T): path_to?(c,d) AND m(d) < m(c)
  subset_reachability: bool =
    FORALL(c: T): EXISTS(d: T): in_subset?(d) AND path_to?(c,d)

  measure_drop_implies_subset_reachability: THEOREM
    measure_drop IMPLIES subset_reachability
END subset_reachability
```

We instantiate it with the layered measure function `m` and the reduction relation `swap_equiv?`:

```
                                                              12

IMPORTING subset_reachability[comps,conv?,nat,m,<,swap_equiv?]
all_equiv_to_conv: THEOREM
   FORALL(c:comps): EXISTS(d:comps):
                     conv?(d) AND swap_equiv?(c,d)
```

The global reduction to convenient computations (`all_equiv_to_conv`) is then proved. If we now want to prove properties of all the computations, (Step 4 in our road map) we instantiate the main theory of the verification method:

```
                                                              13
IMPORTING convenient_computations[ comps ]
```

[2] This reasoning is encapsulated in the infrastructural theory `linear_progress`.

This theory only contains `convenient_computations_rule`, the main theorem given in Section 2.1. The global reduction theorem we proved (in Box 12) is the first proof obligation needed to apply this rule, using `swap_equivalence` as a reduction. We can now verify any property for all computations by discharging the other two obligations: back-preservation of the property by the reduction itself, and the verification of the property for the convenient computations alone.

Upon termination, the values stored locally at the processors should be sorted. This is an example of a terminating-state property (Section 2.5.1). In the theory `terminating_state_properties`, this family of properties is defined, and it is also proved to be preserved by the `swap_equivalence` reduction.

Assuming that `conv_implies_p` is proved for the desired property, we can apply `convenient_computations_rule` to prove:

```
check_p: THEOREM FORALL(c: comps): p?(c)                14
```

This concludes the detailed description of the proof environment's components, and the way it is expected to be applied. It should be recalled that the user only needs to contribute the parts that appear in the thin numbered boxes—the thicker ones are a summary of the theories supplied, and not a part of what the user has to develop.

## 4. Discussion

### 4.1. The user's design decisions and tradeoffs

As in any proof method, experience is essential in successfully applying the elements of this method. The choice of the *relevant* computations can be critical, especially for proving the importing assumptions of the theories that define measure functions.

When proving that the reduction preserves the property to be verified, and also when proving that the independence relation implies functional independence, it helps to have as small an independence relation as possible. This conflicts with the interest of having more opportunities to swap operations and get a computation with a smaller measure.

If we include more computations in the class of convenient computations, it may be easier to show a reduction to a smaller measure for the remaining non-convenient computations. On the other hand, we reduce the benefit of the reduction by having to prove the desired properties directly for a larger class of convenient computations.

### 4.2. Other considerations

There are certain properties that are not directly preserved by the swap equivalence reduction. Swap equivalence involves "diamonds" as the one depicted in Figure 1, so equivalent computations don't necessarily go through the same states. It is possible that a computation violates mutual exclusion while a swap-equivalent one doesn't (e.g., if the violation occurs on the upper vertex of that diamond). The functional independence of the operations being swapped doesn't take into account the intermediate states. This problem can be solved by adding auxiliary "history" variables, like a flag that records an event like the violation of mutual exclusion. After the addition of such a flag, an operation leading to a state in which mutual exclusion is violated (for the first time in the computation) cannot be functionally independent of another operation leading to a state in which this doesn't happen. The two permutations of these operations do not converge into the same state. They now lead to two different states, which differ by the value of the added flag, and therefore the two computations are made non-equivalent. In fact, in Section 7 we showed that the invariance of the negation of stable state properties is preserved by swap-equivalence. If we consider the state property "the flag is up" then the invariance of its negation expresses the computation property of mutual exclusion. This shows how a special provision done when modelling the system can support the verification of additional properties by this method.

The swap-equivalence reduction relation can relate computations with finitely many differences. Due to the functional independence required, all equivalent computations finally converge to a common state, so properties depending only on suffixes are naturally preserved. The characterization of the properties that can be proved by this method using different reduction relations is a subject worth further research.

In this paper we have focused on the proofs that every computation can be reduced to a convenient one. None of the proof obligations generated while proving this reduction involves the specification (in our example, sorting is not mentioned, and the values being sorted are not relevant). A clear separation of concerns has been achieved: we can prove the specification (only for the convenient computations) and the reduction separately.

### 4.3. The applicability of the method

As seen in the non-implemented examples in previous works, this approach is especially appropriate for systems that are already specified by equivalence to a subset of computations. A classic example is database serializability, where a concurrency control protocol is defined as correct if every sequence of operations is equivalent to one where the operations are done transaction by transaction. Similar specifications are made about processors that have out-of-order execution of instructions, or when a distributed system is specified using typical scenarios of communications. Those scenarios can be taken as the convenient computations, and more complex interactions of messages are shown equivalent to computations with sequences of scenarios.

Well-known equivalence properties, like *sequential consistency* of memory accesses in systems with caching or distributed shared memory, may involve infinitely many permutations between a computation and its selected representative. Swap equivalence can be used to prove sequential consistency for all *finite prefixes* of computations. In [8], the importance of considering the infinite case is demonstrated, and it is shown how to extend sequential consistency of finite prefixes to whole infinite computations. We are currently looking into other reductions, which are not equivalence relations, and are not restricted to finitely many transformations separating a computation from its representative (convenient) counterpart.

In organizing the proof environment we have consciously attempted to keep open the way in which the subgoals are verified. This should allow maximal flexibility in integrating other techniques, such as abstraction and model checking.

## Acknowledgements

This research was supported by the Fund for the Support of Research at the Technion.

## References

1. J.-R. Abrial. *The B-Book — Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. K. R. Apt, N. Francez, and S. Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2:226–241, 1988.
3. K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb 1985.
4. C. Chou and E. Gafni. Understanding and verifying distributed algorithms using stratified decomposition. In *Proceedings of 7th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 44–65, 1988.
5. T. Elrad and N. Francez. Decompositions of distributed programs into communication closed layers. *Science of Computer Programming*, 2(3):155–173, 1982.

6. N. Francez. *Fairness*. Springer-Verlag, 1986.
7. M. Glusman and S. Katz. Mechanizing proofs of computation equivalence. In N. Halbwachs and D. Peled, editors, *Proceedings of 11th International Conference on Computer-Aided Verification, (CAV'99)*, volume 1633 of *LNCS*, pages 354–367, Trento, Italy, 1999. Springer-Verlag.
8. M. Glusman and S. Katz. Extending memory consistency of finite prefixes to infinite computations. In K. G. Larsen and M. Nielsen, editors, *Proceedings of 12th International Conference on Concurrency Theory, CONCUR'01*, volume 2154 of *LNCS*, pages 411–425, Aalborg, Denmark, August 2001. Springer-Verlag.
9. P. Godefroid. On the costs and benefits of using partial-order methods for the verification of concurrent systems. In D. Peled, V. Pratt, and G. Holzmann, editors, *Partial Order Methods in Verification*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 289–303. American Mathematical Society, 1997.
10. D. Goldschlag. *Mechanically Verifying Concurrent Programs.* PhD thesis, Department of Computer Science, University of Texas at Austin, 1992.
11. J. Hooman. Program design in PVS. In K. Berghammer, J. Peleska, and B. Buth, editors, *Workshop on Tool Support for System Development and Verification*, Bremen, Germany, 1997. Available at `http://www.win.tue.nl/cs/tt/hooman/PDPVS.html`.
12. P. B. Jackson. Verifying a garbage collection algorithm. In J. Grundy and M. Newey, editors, *Proceedings of 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume 1479 of *LNCS*, pages 225–244. Springer-Verlag, September 1998.
13. C. B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International, second edition, 1990.
14. S. Katz. Refinement with global equivalence proofs in temporal logic. In D. Peled, V. Pratt, and G. Holzmann, editors, *Partial Order Methods in Verification*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 59–78. American Mathematical Society, 1997.
15. S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101:337–359, 1992.
16. S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6:107–120, 1992.
17. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - Specification.* Springer-Verlag, 1992.
18. A. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and editors G. Rozenburg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *LNCS*, pages 279–324. Springer-Verlag, 1986.
19. C. Muñoz and J. Rushby. Structural embeddings: Mechanization with method. In J. Wing and J. Woodcock, editors, *FM99: The World Congress in Formal Methods*, volume 1708 of *LNCS*, pages 452–471, Toulouse, France, September 1999. Springer-Verlag.
20. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
21. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference.* Computer Science Laboratory, SRI International, Menlo Park, CA, September 1998.

22. D. Peled. Combining partial order reductions with on-the-fly model checking. *Journal of Formal Methods in System Design*, 8:39–64, 1996.
23. D. Peled and A. Pnueli. Proving partial order properties. *Theoretical Computer Science*, 126:143–182, 1994.
24. D. Peled, V. Pratt, and G. Holzmann, editors. *Partial Order Methods in Verification*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1997.
25. S. Qadeer and N. Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In D. Gries and W. P. de Roever, editors, *IFIP International Conference on Programming Concepts and Methods (PROCOMET '98)*, pages 424–443, Shelter Island, NY, June 1998. Chapman & Hall.
26. J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.
27. P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In E. Best, editor, *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *LNCS*, 1993.

# Appendix

## A. Summaries of the `ltl` and `op_fairness` theories

```
ltl [ steps: TYPE ] : THEORY
BEGIN
IMPORTING time
  step_seqs : TYPE = [time->steps]
  point_pred: TYPE = [step_seqs,time->bool] % path formulas
  t,t0,t1: VAR time
  s: VAR step_seqs
  p?, q?: VAR point_pred

  G(p?): point_pred =
    LAMBDA(s,t0): FORALL(t): t0<=t IMPLIES p?(s,t)
  F(p?): point_pred =
    LAMBDA(s,t0): EXISTS(t): t0<=t AND p?(s,t)
  X(p?): point_pred = LAMBDA(s,t0): p?(s,t0+1)
  until(p?,q?): point_pred =
    LAMBDA(s,t0): EXISTS(t1): t0<=t1 AND q?(s,t1) AND
                    FORALL(t): t0<=t AND t<t1 IMPLIES p?(s,t)
  wuntil(p?,q?): point_pred = G(p?) OR until(p?,q?)
  ...
  % some useful lemmas, like:
  future_p_equals_not_p_until_p: LEMMA F(p?)=until(not p?, p?)
END ltl
```

```
op_fairness[states,ops,initial?,en?,next,
  (IMPORTING execution_sequences[states,ops,initial?,en?,next])
  weakly_fair_ops?: pred[ops],
  strongly_fair_ops?: pred[ops]  ] : THEORY
BEGIN
 ...
  taken?(op): point_pred =
           LAMBDA(seq:step_seqs[steps] ,t): op=seq(t)`op
  en?(op): point_pred =
           LAMBDA(seq: step_seqs[steps],t): en?(seq(t)`st,op)
  continually_enabled_beyond?(op): point_pred = G( en?(op) )
  never_taken_beyond?(op): point_pred = G( not taken?(op) )
  is_weakly_fair_wrt?(e, op): bool =
      G( not ( continually_enabled_beyond?(op) AND
                      never_taken_beyond?(op) ) )(e,0)
  is_weakly_fair?(e: execution_sequences): bool =
      FORALL(op: (weakly_fair_ops?)): is_weakly_fair_wrt?(e,op)
  enabled_infinitely_many_times?(op): point_pred = G(F(en?(op)))
  taken_finitely_many_times?(op): point_pred =
      F(never_taken_beyond?(op))
  is_strongly_fair_wrt?(e,op): bool =
      G( not ( enabled_infinitely_many_times?(op) AND
                      taken_finitely_many_times?(op) ) )(e,0)
  is_strongly_fair?(e): bool =
      FORALL(op: (strongly_fair_ops?)): is_strongly_fair_wrt?(e,op)
  is_fair?(e): bool = is_weakly_fair?(e) AND is_strongly_fair?(e)
END op_fairness
```

## B. Summary of layered_measure theory

```
layered_measure [states,ops,initial?,en?,relevant?,indep?,

   lastlayer: nat,
   layer: [ comps, time -> upto(lastlayer) ]  ]: THEORY
BEGIN
 ASSUMING
  can_order_layers: ASSUMPTION
    FORALL(c: comps, t:time):
              layer(c,t) > layer(c,t+1) IMPLIES can_swap?(c,t)
  F_G_different_layers: ASSUMPTION
    FORALL(c: comps, l: below(lastlayer)): EXISTS(tl: time):
              FORALL(t: time | t >= tl): layer(c,t) /= l
                             % continued in the next page ...
```

```
  layer_preservation: ASSUMPTION
    FORALL(c,d:comps, swt:time|swap_at_swt?(c,d,swt), t:time):
              layer(d,t) = COND t=swt   -> layer(c,swt+1),
                                t=swt+1 -> layer(c,swt),
                                ELSE    -> layer(c,t) ENDCOND
 ENDASSUMING
m(c:comps): nat =            % layered measure function
  sum(fullset[below(lastlayer)], LAMBDA(l: below(lastlayer):
     sum({t: time|layer(c,t)=l}, LAMBDA(t:time):
        card({t2: time|t2<t AND layer(c,t2)>layer(c,t)})
% A sufficient condition for measure decrease
post_before_prev_has_smaller_equiv: THEOREM
  FORALL(c: comps):
    ( EXISTS(t1,t2: time): t2>t1 AND layer(c,t1)>layer(c,t2) )
    IMPLIES EXISTS(d: comps): swap_equiv?(c,d) AND m(d) < m(c)
END layered_measure
```