Mechanizing Proofs of Computation Equivalence

Marcelo Glusman and Shmuel Katz

Department of Computer Science
The Technion, Haifa, Israel
{marce, katz}@cs.technion.ac.il

Abstract. A proof-theoretic mechanized verification environment that allows taking advantage of the "convenient computations" method is presented. The PVS theories encapsulating this method reduce the conceptual difficulty of proving a safety or liveness property for all the possible interleavings of a parallel computation by separating two different concerns: proving that certain *convenient* computations satisfy the property, and proving that every computation is related to a convenient one by a relation which preserves the property. We define one such relation, the equivalence of computations which differ only in the order of independent operations. We also introduce the computation as an explicit semantic object. The application of the method requires the definition of a "measure" function from computations into a well-founded set. We supply two possible default measures, which can be applied in many cases, together with examples of their use. The work is done in PVS, and a clear separation is made between "infrastructural" theories to be supplied as a *proof environment* library to users, and the specification and proof of particular examples.

1 Introduction

This paper presents a proof environment for PVS [13,17,12] that supports convenient computations and exploits partial order based on the independence of operations in different processes, for the first time in a mechanized theorem-proving context. Thus theoretic work defining this approach [8,9,11] is turned into a *proof environment* for theorem-proving that can be used without having to rejustify basic principles. Besides making convenient computations practical in a theorem-proving tool, we demonstrate what is involved in packaging such a framework into a proof environment for use by nonexperts in the particular theory. The modular structure of the theories (the units of PVS code that contain definitions and theorems) should encourage using parts of the environment whenever convenient computations are natural to the problem statement or proof.

In the continuation, basic theories are described in which computation sequences are viewed as 'first-class' objects that can be specified, equivalence of computations based on independence of operations is precisely defined, and a proof method using measure induction over well-founded sets is encapsulated. Two possible default measures are provided to aid the user in completing the proof obligations, one involving the distance between matching pairs of events,

N. Halbwachs and D. Peled (Eds.): CAV'99, LNCS 1633, pp. 354–367, 1999.

and one for computations that have *layers* of events. As an example among those classes of properties that can be proven with the convenient computations method, (those properties preserved by the chosen reduction relation), we show a subclass of the stable properties: final-state properties.

We summarize how a user can exploit the environment and describe two generic examples. The first demonstrates ideas of *virtual atomicity* for sequences of events local to a single process, and the second shows the use of the layers measure for a pipelined implementation of insertion sorting. In this and the other 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.

Convenient Computations and the Need for Mechanization

Methods that exploit the partial order among independent operations have been used both for model checking and for general theorem proving (see [16] for a variety of approaches). In particular, ideas of the independence of operations that lead to partial order reductions have either been used for (usually linear) temporal logic based model checking reductions [14,18,7], or for theoretical work on general correctness proofs in unbounded domains. [11,15,8]. For general correctness (as opposed to model checking) no mechanization has been implemented until now, and sample proofs have been hand simulated.

The intuitive idea behind convenient computations is simple. A system defines a collection of linear sequences of the events and/or states (where each sequence is called a *computation*). We often convince ourselves of the correctness of a concurrent system by considering some "convenient" computations in which events occur in an orderly fashion even though they may be in different processes. It is usually easier to prove properties for these well-chosen computations than for all the possible interleavings of parallel processes. Two computations are called *equivalent* if they differ only in that independent (potentially concurrent) events appear in a different order. There are classes of safety and liveness properties which are satisfied equally by any two equivalent computations (i.e., either both satisfy the property, or neither does). If we show that any non-convenient computation is equivalent to some convenient one, then we can conclude that any properties of this kind verified for the convenient computations must also be satisfied by the non-convenient ones.

In certain contexts, like sequential consistency of memory models and serializability of database transaction protocols, where the convenient computations' behavior is taken as the correct one by definition, the computation equivalence itself is the goal of the verification effort. Even when the goal is to prove certain properties of a system, if we attempt to reduce the problem to verification of the convenient computations, we might find flaws in our intuitive belief that they "represent" all possible computations. Finding that some computations are *not* equivalent to any convenient one can have as much practical value as finding a counterexample to a property expressed by a logical formula.

The availability of general purpose theorem proving tools such as PVS opens the way for a mechanized application of the convenient computations technique. Usually, attempts to carry out mechanized proofs in such tools raise issues that might be overlooked when using "intuitive" formal reasoning. Moreover, proofs can be saved and later, in the face of change, adjusted and rerun rather than just discarded. The down side of mechanized theorem proving methods is the need to prove many facts that are easily understood and believed to be correct by human intuition. Many of these facts are common to all applications of a proof approach. General definitions and powerful lemmas can be packed in theories that provide a comfortable proof environment. These theories also clarify the new approach, and generate the needed proof obligations for any particular application. The proof obligations arise as "importing assumptions" of generic theories, as "type checking conditions" when defining objects of the provided types, or as antecedents (preconditions) in the provided theorems that have the form of a logical implication.

Existing Versus Proposed Verification Styles

The PVS tool is a general-purpose theorem prover with a higher-order logic designed to verify and challenge specifications, and is not tailored to any computation model or programming language. It does provide a *prelude* of theories about arithmetic, inequalities, sets, and other common mathematical structures, and some additional useful libraries. Decidable fragments of logic are treated by a collection of decision procedures that replace many tedious subproofs. However, it has no inherent concept of states, operations, or computations. Usual verification examples, like proving an invariant in a transition system, involve the definition of states, initial conditions (initial-state functions), and transitions (next-state functions) by the user. To prove invariance of a state property P, one just writes and proves an induction theorem of the form:

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

The computations themselves are not mentioned directly, and the property "in every state, P" ($\Box P$ of linear temporal logic) is justified (usually implicitly) by such an induction theorem.

As part of the proof environment presented here, we provide precise definitions for computations, conditional independence of operations, computation equivalence, and verification of properties based on computation equivalence and convenient computations using well-founded sets.

We define a "computation" type as a function from discrete time into "steps" (a state paired with an operation), and specify temporal properties as arbitrary predicates over computations. Thus, if $c : comps$ is a function from $t : time$ to steps, and st is a projection function from a step to its *state* component, then

$$\Box P = global\_P?(c : comps) : bool = \forall t \in time : P(st(c(t)))$$

This style is needed so we can reason about computations and their relationships. Note that we are not limited to linear-time properties by this style of expression. The higher-order logic of PVS allows us for example, to express a CTL^* formula like "GEFp" by means of a predicate on computations:

$$\begin{aligned} &GEFp(c : comps) : boolean = \\ &\quad \forall(t : time) : \exists(d : comps) : \ \forall(tp : time | tp < t) : d(tp) = c(tp) \\ &\qquad\qquad \wedge \exists(tf : time | tf \geq t) : p(st(d(tf))) \end{aligned}$$

or in words, "for every time point t there exists a computation (called d) which is identical to c before time t and, at time t or later, has a state that satisfies p"

2 The Theories

In this section we describe a hierarchy of theories, whose `IMPORTING` relationships can be seen in Figure 1. They provide the foundation for reasoning about equivalence among computations. The top level of the hierarchy contains three main components: the *computation model*, the *equivalence* notion, and the *proof method*. An additional *default measure* component uses the other theories in specific contexts that hide some of the proof obligations in an application.

In the computation model component, transition systems and computations over them are defined. The option of providing global restrictions on possible sequences of transitions (for example, in order to introduce various notions of *fairness* [6,2]) is also provided. In the equivalence component, theories are presented that encode when transitions are independent, and when computations are defined to be equivalent (in that independent transitions occur in a different order). The proof method component shows how to prove that for an arbitrary set and subset, every element of the set is related to some element of the subset, using well-foundedness. Here the elements are arbitrary, and the relation is given as a parameter. When instantiated with the equivalence relation from the equivalence component, the needed proof rule and its justification are provided. As an example of the classes of properties relevant to this method, we include a theory that defines the "final-state properties" and proves that they are preserved by the defined computation equivalence relation.

After presenting these theories in somewhat more detail, two default measures are described, for matching pairs of operations and for layered computations. We then summarize how a user should apply the theories to an application, and describe two examples.
(Note: The PVS files for the proof environment and the examples are available from the Web page at `http://www.cs.technion.ac.il/~marce`).

2.1 Computation Model

Our model of computation is defined by three parameterized theories: `step_sequences, execution_sequences`, and `computations`. Each application using these theories must define types for the specific states and the operations as

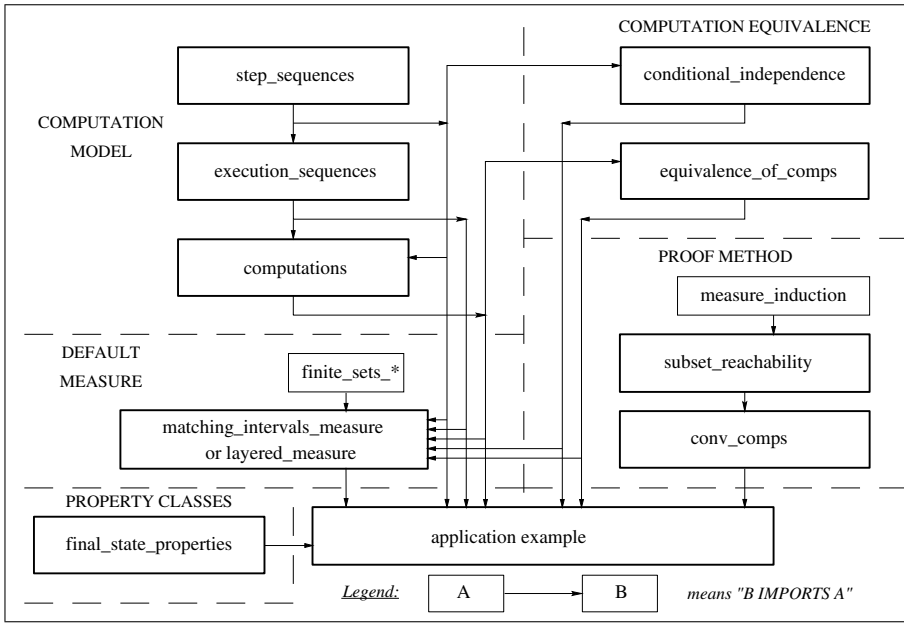

Fig. 1. The hierarchy of theories.

actual parameters upon instantiation of the theories. The theory `step_sequences`, based on these two types, defines function types needed to build a transition system: `initial_conditions`, `enabling_conditions`, and `next_state_functions`. It also defines the types `time`, `steps` (records with two fields: `st: states` and `op: ops`), and `step_seqs` (functions from time to steps).

In an application, the user defines the initial states, the enabling conditions and the next-state functions for each operation, and then instantiates the theory `execution_sequences`. This theory defines the subtype of the well-built `execution_sequences`: the ones that start in an initial state, and whose steps are consecutive, i.e., the operations are enabled on the corresponding state, and the state in the following step is their next-state value.

The theory `computations` has an additional parameter to be provided by a user, namely, a predicate on `execution_sequences` called `relevant?` which is used to define the subtype `comps`. This includes only those `execution_sequences` which satisfy the added predicate. This restriction can be used to focus on a special subset of all the possible execution sequences, for example to express fairness assumptions or to analyze just a part of the system's behavior (such as a finite collection of transactions).

2.2 Computation Equivalence

Equivalence between computations and independence of operations is formalized by the theories `conditional_independence` and `equivalence_of_comps`.

The functional independence defined in the first of these theories is over pairs of operations and states, expressing when two operations are independent in a given state. It requires that the execution of either of the two operations doesn't interfere with the other's enabledness, and that the result of applying both operations from the given state must be the same regardless of their ordering.

Though this functional independence expresses commutativity of operations, it is not practical to prove it each time we need to show that a pair of consecutive operations can be exchanged. To separate this (local) consideration and make later proofs simpler, we allow the user to define a separate conditional independence relation also over pairs of operations and states. This predicate must be symmetric and it must imply the functional independence of the two operations from the given state. (These conditions will appear as proof obligations when the theory is instantiated.) This arrangement allows the user to choose how much independence is to be considered for a particular application.

The theory `equivalence_of_comps` first defines the result of swapping two independent operations on a given state in an execution sequence. If the need arises to prove that the result is a legal computation (a `relevant?` execution sequence), it is passed as a proof obligation to the application since `relevant?`is only defined there. The rest of the theory deals only with legal computations that are identical up to the swapping of independent operations, defining:

- `one_swap_equiv?(c1,c2)`: c1 and c2 are different and differ by a single swap, i.e., c2 is the result of swapping consecutive independent operations in c1 at some time t.
- `swap_equiv_n?(c1, c2, n)`: c1 and c2 differ by up to n single swaps.
- `swap_equiv?(c1,c2)`: this is the transitive closure of `one_swap_equiv?` and is true iff there is an n s.t. `swap_equiv_n?(c1, c2, n)`.

In the theory, the relation `swap_equiv?` is proven to be an equivalence relation. This relation is the formalization of the intuitive notion of equivalent computations, and the equivalence classes that it generates in the set of all computations are called *interleaving sets* in the context of partial order reductions and the temporal logic $ISTL^*$[9,10].

2.3 Proof Method

Consider an arbitrary set (or data type) `T`, with a preorder relation `path_to?` over its elements, and choose a subset of `T` - those elements which satisfy a given predicate. We want to prove that from each element in `T` we can reach one in the chosen subset. We first pick a "measure function" which maps elements from `T` into elements of a well-founded structure `(M,<)`. In the theory `subset_reachability` we show that it suffices to prove that each element *outside* the chosen subset has a path to one with a strictly smaller measure.

The theory `conv_comps` has parameters that define a computation model, a `reduces_to?` preorder, a predicate for choosing the `conv?`enient computations, and a measure function into a well-founded set. These are used with the

`subset_reachability` theory to provide a sufficient condition:

$$\forall c: \ \neg conv?(c) \implies \exists d : reduces\_to?(c,d) \ \wedge m(d) < m(c) \tag{1}$$

from which reduction to convenient computations is proved:

$$\forall c : \exists d : conv?(d) \wedge reduces\_to?(c,d)$$

It also provides a theorem defining the two added proof obligations that must be discharged in an application to verify *any* property `p?` for all computations:

$$\forall c : conv?(c) \implies p?(c)$$

$$\forall c, d: \ reduces\_to?(c,d) \implies (p?(d) \implies p?(c)) \tag{2}$$

In other words, `p?` must be true for the convenient computations, and must respect the preorder used in the theory. In a wider context, the theory of convenient computations can be used to reduce the verification of properties of general computations to the simpler problem of verification over the convenient computations. The `reduces_to?` relation can be any preorder for which the required premises ((1) and (2)) can be proven. Since the theory is parametric, other computation models and notions of equivalence can be used, besides those seen here.

2.4 Property Classes

Any property preserved by the relation (preorder) chosen as a reduction to convenient computations, is a candidate to be verified by this method. A common example is that of stable properties. The theory `final_state_properties` exemplifies a special case of stable properties. It defines a final state of a computation as any point in time after which the computation remains quiescent i.e. every operation-state pair is the same as the next one. A function is defined that, given a state property, generates a computation predicate that enforces that state property on all final states. The "final-state properties" thus generated are proven invariant under the swap-equivalence relation.

2.5 Default Measures

The choice of measure functions should address the intuitive notion of "how close" a computation is to a convenient one. (e.g. how many independent operation pairs should be swapped). Only then will the proof obligations generated be easy (if not trivial) to discharge. We provide theories with two measures that widen the support given to the user of the method.

Matching intervals measure: In [8] the convenient computations method was applied (manually) to the sequential consistency problem. The measure involved intervals of selected events (computation steps) and their length. The measure value was lowered by moving unrelated events out of the interval until all the selected events happened consecutively. We provide a simpler version which can be applied to achieve the same effect. An *interval* is defined as a pair of points in time $(t1, t2)$, and its distance (length) is $t2 - t1 - 1$ (thus a consecutive pair $(t, t+1)$ has distance zero). The measure value for a computation is defined as the sum of all the distances of its matching intervals.

To use this measure, the application must supply a predicate `match?(c,i)` that defines the "matching" intervals `i` (pairs of points) in a given computation `c`. In a matching interval we want two events to ideally happen immediately one after the other, in a certain order, even if in many computations there are intervening events. Typical cases are sending and receiving a value over an empty communication channel, or performing a series of local steps in a process. 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 is to make the measure finite. (An alternative would be to require that the set of nonzero-distance matching intervals be finite, and sum distances only over that set.)
- The matching intervals in two one-swap-equivalent computations are the same, up to the exchange of the end-points affected by the swap.
- No two matching intervals start in the same time point and no two end together. This is used to simplify the number of cases.
- Swappable (i.e., independent consecutive) operations cannot appear at the ends of a (zero-distance) matching interval.

These requirements mainly restrain the choice of the `match?` function to a usable one. Again, for reasonable choices their proof is straightforward.

The theory also provides and proves a heuristic for finding a computation `d` which is equivalent to a given computation `c` and has a smaller measure. Such a `d` exists if `c` satisfies that for some t:

$$(only\_starts\_interval?(c,t) \ \wedge \neg only\_starts\_interval?(c,t+1)) \ \vee \\ (only\_ends\_interval?(c,t+1) \ \wedge \neg only\_ends\_interval?(c,t))$$

where $only\_starts\_interval?(c,t) \ = starts\_interval?(c,t) \wedge \neg ends\_interval?(c,t)$ (and $only\_ends\_interval?$ is defined similarly).
The predicates `starts_interval?(c,t)` and `ends_interval?(c,t)` state that there is a matching interval in c that starts(ends) at time t. This means that either an event only starting an interval is followed by one *not* only starting an interval, or an event only ending an interval is preceded by one not only ending an interval. Due to the other assumptions, if this holds, the relevant pair can be exchanged, yielding a computation with a smaller measure.

Layered measure Another way of thinking of convenient computations of a program is to define ordered phases or layers of execution [5,4]. Each event is associated with a layer. If the events in every layer appear contiguously in the computation, 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.

The `layered_measure` theory considers programs with a finite number of layers, where all but the last one must be finite and eventually finish, i.e., for each computation and for each layer in it, there is a time after which all the events belong to other layers. If infinite computations are considered, this can be achieved by applying some sort of fairness assumption.

For each event (computation step) except those associated with the last layer, we count the number of previous events that belong to a (strictly) later layer than the layer of that event. 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.

The application must define a natural number `lastlayer` and function (`layer`) that maps a computation and a time point into a natural number less than or equal to `lastlayer`. This function must meet the following requirements:

- As mentioned before, for every layer below `lastlayer` there is a time after which there are no more time points belonging to it. Proofs of this requirement are based on basic progress of the computation, which can be supported by fairness assumptions from the `relevant?` predicate.
- The `layer` function is the same for one-swap-equivalent computations, except at the two time points involved in the swap, where the layer values are interchanged. This is trivial for reasonable definitions of the `layer` function.
- For any time t where $layer(t) > layer(t+1)$, the operations at t and $t+1$ must be independent, i.e. a swap must be possible. This seemingly strong requirement is easy to prove if layering is appropriate for the application.

In this theory, it is proven that any assignment of layers satisfying these conditions guarantees that any computation in which a later-layer event comes before a previous-layer event (and thus having a non-zero measure) is equivalent to one with a smaller measure. Thus, showing a drop in the measure value is hidden from a user, if the three conditions above can be shown.

3 Using the Method: A Summary

3.1 The User's Problem Description

First, the **computation model** must be described by defining the types of the states and operations, the initial states, the operations' enabling conditions, and

their next-state functions. Any necessary global restrictions such as fairness or finiteness are then added to define the relevant computations.

Second, the user must define the conditional independence relation between the operations at given states. This is used to instantiate the **computation equivalence** theory which will provide the `swap-equiv` relation. The theory will generate proof obligations to show that the user's suggested relation is a valid independence relation. Finally, in the **proof method** theories, the convenient computations must be provided in the instantiation of the theory `conv-comps`, and a measure function must be defined (either by the user, or using one of the two provided).

These are all the definitions needed to prove computation equivalence. Aside from the importing assumptions of the theories used, the user is left to prove that for every non-convenient computation there is a reduction to a computation with a lower measure (for the two default measures provided there is a sufficient condition that makes that proof much easier).

To prove any property (predicate over computations) for all the computations of an application, the theorem provided in `conv_comps` leaves the user to prove that the property holds for convenient computations and that equivalence over the user's independence relation preserves the property.

3.2 The User's Design Decisions and Tradeoffs

As in any proof method, experience is essential in successfully applying the elements of this method. Choosing the *relevant* computations can be critical, especially in 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 in order to find 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 nonconvenient computations. On the other hand, we reduce the benefit of the use of equivalence by having to prove the desired properties directly for a larger class of convenient computations.

As seen in the proof obligation (2), the properties that can be verified when the theories are combined in an application are those which are preserved by the reduction relation. A lemma in the theory `equivalence_of_comps` simplifies this requirement: it suffices to show that two computations which differ only in the order of *one* pair of independent operations, must satisfy `p?` equally:

$$\forall c, d : one\_swap\_equiv?(c, d) \implies p?(d) \implies p?(c) \tag{3}$$

This requirement is easy to prove for large classes of properties, e.g., those defined in the theory `final_state_properties`.

In certain cases, one might need to add "history" variables to the state, (without affecting the behavior of the rest of the state components) to support property verification. For example, in order to verify mutual exclusion, a flag that records a violation of the mutual exclusion should be added. This is done so that two computations which differ only by the order of a pair of operations are not considered equivalent if one of them violates the mutual exclusion requirement and the other does not. The original system variables might not suffice to make those operations functionally dependent.

The characterization of the properties which can be proven by this method is a subject worth further research. In this paper we have focused on the proofs that computations are equivalent, and particularly on showing that every computation is equivalent to one of the convenient computations.

4 Example 1: Using the Matching Intervals Measure

Our first example (a full listing is at the Web page given earlier) shows how a sequence of local actions in a process can be considered atomic. It is typical of many situations where a sequence of local actions can be viewed as virtually atomic [1].

```
%      flag: bool=FALSE      tl,tm,x: nat
% PL:  l0: tl=1       % local || PM:  m0: tm=2             % local
%      l1: x=tl       % global||      m1: await flag=TRUE
%      l2: flag=TRUE          ||      m2: x=x+tm           % global
%      l3: STOP               ||      m3: STOP
```

Here the operation l2 must occur before m1, so in fact we observe all the possible interleavings of the operation m0 (PM's initialization) with the operations l0-l2. The `states` type contains the two program counters explicitly. The `ops` type is `{l0,l1,l2, m0,m1,m2, stop}`. The `initial?` predicate on states is straightforward. The `en?` enabling condition, and the `next` next-state-function are defined in table format to enhance readability.

In this example we define two operations as independent if they are both `stop` or if they belong to different processes and satisfy `indep_l_m?`, a predicate given in tabular form. The table's rows and columns represent operations which belong to different processes, and the entries are state predicates, though in this particular case they are not state-dependent (always TRUE or FALSE). This table was filled based on our understanding of the semantics of the programming language. The independence relation must be proved to imply functional equivalence and to be symmetric, as a type-correctness requirement. After that is done, to decide if two operations can be swapped, we only need to look them up in the table. The convenient computations are chosen as those in which `m0` is executed immediately before `m1` (and after `l0`–`l2`). We choose to use the default measure with matching pairs. Here we can define the `match?` predicate

so that only the pair (`m0,m1`) matches. Clearly, when the measure is zero, the computation is convenient.

The proof that for any nonconvenient computation there is an equivalent one with a smaller measure was accomplished by using the theorem provided in the `matching_intervals_measure` theory. Note that if the instructions in question were in a loop, the definition of "matching intervals" would have to guarantee that the proper occurrences of the instructions are matched, e.g., by using a loop counter as well as the operations. There is another condition: computations must have a finite number of matching intervals. In the present example, this is easy to show since each operation is done exactly once. In general, this would be proven by using some kind of finiteness constraint, typically from the `relevant?` predicate.

Although our main concern is proving computation equivalence, we show the remaining proof obligations for a final-state property. The proof obligation (`conv_implies_p`) shows that p holds for the convenient computations and is not completed here. The other obligation (`one_swap_equiv_preserves_p`) is easily discharged by invoking a theorem from the theory `final_state_properties`.

5 Example 2: Using the Layered Measure

Our second example (also available at the Web page) is a typical representative of the pipelined processing paradigm. In our example, all computations are equivalent to those that execute "one wave at a time," i.e., in which a new input is entered only when all the operations related to the previous inputs have been finished. The program is a pipelined insertion-sort algorithm in which the buffers between the processors can hold a single value. We assume that each processor does its local actions atomically: taking its input, comparing it with the value it holds, and sending the maximum between them to the next processor in the pipeline. 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 value, and some of them have a nonempty incoming buffer. There could be several such processors whose successor has room in its 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.

The example is described in the theory `pipeline_sort`, parametric on the type of the values being sorted, their total-ordering relation, the number of input values (and of processors) `NUM`, and an array from 0 to `NUM-1` holding those values.

The processors' indices range from 1 up to `NUM`. Since we choose to use the layers approach, we augment the state variables and next-state functions to allow defining the layer value of each computation step. The system state includes a counter of the number of inputs already inserted, and an array of processor states. Each such processor state includes a locally held value, an input buffer,

and an integer (`input_layer`) that holds the *layer* associated with that input. 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 held value.

The layer value of an "input-new-value" operation is the value of the global input counter. For a normal computation step by any processor, the layer value is the `input_layer` stored in that processor's state. Since it originated from the global counter's value when the layer began, this value ranges from 0 up to `NUM-1`. The idling operation, enabled only at the end of the whole computation, has a layer value `NUM`.

The initial states, enabling conditions and next-state-functions are coded in a straightforward way. In this case, we imposed no added restrictions when describing the `relevant?` computations.

The independence relation is defined as TRUE only between operations done by non-adjacent processors (and for two idling steps). This simplified relation is much easier to use during the proofs than the functional independence relation.

These are the (nontrivial) proof obligations generated after instantiating all the needed infrastructural theories with the above mentioned definitions:

- The user's independence relation implies functional independence and is symmetric. Proving this requires only local reasoning.
- Each layer eventually ends. To prove this we used sublemmas that show eventual progress by simple induction.
- The layering function is consistent for one-swap-equivalent computations. This is easily proven because the `layer` function's definition is local.
- Consecutive events whose layer values are not in ascending order can be swapped. To prove this, we show that any two such events can only involve non-contiguous processors, whose operations are independent by definition.

To prove computation equivalence using the theorem from `conv_comps`, we need to prove that each nonconvenient computation is equivalent to one with a smaller measure. Using the theorem from the `layered_measure` theory, we only need to prove that in each non-convenient computation there is an event a that precedes an event b where b's layer value is strictly smaller than a's. To prove this we show that the layer value of an input operation is bigger than that of any operation belonging to a processing "wave" that started with a previous input.

Note that none of the proof obligations involve the specification (sorting is not mentioned, the values sorted are not relevant) and all are local or structural in nature. Since the layer measure is appropriate to the structure of the system, any difficulty in the proofs is technical, not conceptual.

Acknowledgment

This research was supported by the Bar-Nir Bergreen Software Technology Center of Excellence at the Technion.

References

1. K. Apt and E. R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 1991.
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 Trans. 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 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. 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*, pages 289–303. American Mathematical Society, 1997. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 29.
8. S. Katz. Refinement with global equivalence proofs in temporal logic. In D. Peled, V. Pratt, and G. Holzmann, editors, *Partial Order Methods in Verification*, pages 59–78. American Mathematical Society, 1997. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 29.
9. S. Katz and D. Peled. Interleaving set temporal logic. *Theoretical Computer Science*, 75:263–287, 1990. Preliminary version was in the 6th ACM-PODC, 1987.
10. S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101:337–359, 1992.
11. S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6:107–120, 1992.
12. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference.* Computer Science Lab, SRI International, Menlo Park, CA, 1998.
13. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich 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.
14. D. Peled. Combining partial order reductions with on-the-fly model checking. *Journal of Formal Methods in System Design*, 8:39–64, 1996.
15. D. Peled and A. Pnueli. Proving partial order properties. *Theoretical Computer Science*, 126:143–182, 1994.
16. D. Peled, V. Pratt, and G. Holzmann(eds.). *Partial Order Methods in Verification.* American Mathematical Society, 1997. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 29.
17. John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.
18. P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *Proceedings of CONCUR'93 (Eike Best, ed.), LNCS 715*, 1993.