# Induction in Compositional Model Checking

Kenneth L. McMillan[1], Shaz Qadeer[2], and James B. Saxe[2]

[1] Cadence Berkeley Labs
[2] Compaq Systems Research Center

**Abstract.** This paper describes a technique of inductive proof based on model checking. It differs from previous techniques that combine induction and model checking in that the proof is fully mechanically checked and temporal variables (process identifiers, for example) may be natural numbers. To prove $\forall n.\varphi(n)$ inductively, the predicate $\varphi(n-1) \Rightarrow \varphi(n)$ must be proved for all values of the parameter $n$. Its proof for a fixed $n$ uses a conservative abstraction that partitions the natural numbers into a finite number of intervals. This renders the model finite. Further, the abstractions for different values of $n$ fall into a finite number of isomorphism classes. Thus, an inductive proof of $\forall n.\varphi(n)$ can be obtained by checking a finite number of formulas on finite models. The method is integrated with a compositional proof system based on the SMV model checker. It is illustrated by examples, including the $N$-process "bakery" mutual exclusion algorithm.

## 1 Introduction

In verifying concurrent or reactive systems, we are often called upon to reason about ordered sets. For example, a packet router may be required to deliver a sequence of packets in order, or a set of processes may be ordered by a linear or grid topology. In such cases, it is convenient to reason by induction. For example, we show that if packet $i-1$ is delivered correctly, then so is packet $i$, or that if a ring arbiter with $n-1$ processes is live, then so is a ring with $n$ processes. Indeed, inductive proof may be necessary if the ordered set is unbounded.

For an ordered set of finite state processes, it is natural to consider using model checking to prove the inductive step. This was proposed a decade ago by Kurshan and McMillan [KM89] and by Wolper and Lovinfosse [WL89]. They considered a class of finite-state processes constructed inductively, and an inductive invariant over this class also expressed as a finite-state process. To prove the invariant, it suffices to check the inductive step of the proof for a finite number of instances of the induction parameter. This task can be carried out by finite-state methods. However, no mechanical method was proposed to check the validity of this "meta-proof", or to automatically generate the required instances of the induction step. Further the technique was limited to finite state invariants. This limitation can be observed, for example, in the work of Henzinger, Qadeer and Rajamani [HQR99], who constructed an inductive proof of a cache coherence protocol. The method cannot be applied to protocols that exchange process identifiers (a very common case in practice) because these are not "finite state".

Here, we extend induction *via* model checking beyond finite state invariants, to problems that have not generally been considered amenable to solution by model checking. Our proof method uses a general induction scheme that allows mutual induction over multiple induction parameters. Models for appropriate instances of the induction parameters are generated automatically, and all proof steps are mechanically checked. We illustrate these advantages by examples, including a proof of safety and liveness of a version of the $N$-process "bakery" mutual exclusion algorithm [Lam74].

Our technique has been integrated into the SMV proof assistant, a proof system based on a first order temporal logic [McM97,McM98,McM99].[1] Both the system to be verified and the specification are expressed in temporal logic, though with a great deal of "syntactic sugar". Inductive proofs are reduced to finite state subgoals in the following way. To prove a predicate $\forall n.\varphi(n)$ inductively, we need to prove $\varphi(n-1) \Rightarrow \varphi(n)$ for all values of the parameter $n$ ($\varphi(-1)$ is defined to be true). In general, $\varphi(n)$ may also refer to some fixed constants such as zero, the number of processes, sizes of arrays, etc. To make the problem finite-state for a particular value of $n$, we abstract the natural numbers to a finite set of intervals. Typically, the values $n-1$, $n$, and the fixed constants of interest are represented by singleton intervals, while each interval between these values becomes a single abstract value. We observe that the abstractions for values of $n$ satisfying the same inequality relationships over the given terms (*e.g.*, $0 < n-1 < n$) are isomorphic. Thus it is sufficient to enumerate the feasible inequality relationships (which we call "inequality classes") and verify one representative value of $n$ from each. Thus, we reduce the problem to a finite number of finite-state subgoals.

**Related work** This work builds on the techniques developed by McMillan [McM98,McM99] —temporal case splitting, reduction of cases by symmetry and data type reduction— to reduce proof goals to a finite collection of finite-state subgoals that can be discharged by model checking. The idea of using model checking to prove an inductive step has also been used by Rajan, Shankar and Srivas in the PVS system [RSS95]. They embedded the mu-calculus in PVS, using model checking to verify formulas on finite abstract models. These abstractions, however, were constructed manually, and their soundness was proved with user assistance. Here, the abstractions are constructed automatically and are correct by construction. The user specifies at most a set of constants to be used in an abstract type, although often a suitable set can be inferred automatically. Another approach to model checking within a theorem prover is *predicate abstraction*, in which the state of the abstract model is defined by the truth values of a set of predicates on the concrete model (*e.g.*, [SG97]). Once suitable predicates are chosen, decision procedures can construct the abstract model. However, a suitable set of predicates must still be chosen, manually or heuristically. Saïdi and Shankar used such a method in PVS [ORS92] to verify a two-process mutual exclusion algorithm, similar to our $N$-process example [SS99]. Only safety (*i.e.*,

[1] SMV can be obtained from `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv`.

mutual exclusion) was proved however, and the two-process case does not require induction (except over time). Bensalem, Lakhnech and Owre also report an abstraction technique in PVS that has been used to prove safety in the two-process case [BLO98]. Das, Dill and Park [DDP99] have extended predicate abstraction to deal with parameterized systems by using predicates that involve quantifiers, but they have also considered only safety proofs, not requiring induction. Here, we prove $N$-process liveness, by induction over a lexical order. We note further that unlike predicate abstraction, which in general requires a high-complexity decision procedure, the present method generates abstract models in linear time.

## 2 Type reductions for ordered sets

We now consider the abstraction technique used in SMV to support induction. We begin with the notion of an ordered set type, or *ordset*. This is a data type isomorphic to the natural numbers $\{0, 1, \ldots\}$. We would like to prove a proposition of the form $\forall i.\ p(i)$, where $p$ is a temporal property and $i$ ranges over an ordset type $T$. There are two complications in proving this fact using model checking. First, $p$ may refer to variables of infinite type $T$, hence the state space is not finite. Second, $p(i)$ must be proved for infinitely many values of $i$. An abstraction of the type $T$ to a finite set can solve both of these problems, in that it makes the state space finite, and also reduces the values of $i$ to a finite number of equivalence classes.

As observed in [McM98], if values of type $T$ were used in a symmetric way, then it would suffice to check only one fixed value of $i$. However, this rules out expressions such as $x + 1$ or $x < y$, where $x$ and $y$ are of type $T$. Since precisely such expressions are needed for proofs by induction over $T$, we cannot rely on this kind of symmetry argument. We can, however, induce a symmetry by abstracting the type $T$ relative to the value of $i$. Suppose we fix a value of $i$. We may then abstract the type $T$ to a set of three values: $i$ itself, an abstract symbol representing the semi-closed interval $[0, i)$, and an abstract symbol representing the open interval $(i, \infty)$. Using a suitable abstract interpretation of each operator in the logic, we obtain a conservative abstraction: if $p(i)$ is true in the abstract (finite) model, then it is true in the concrete (infinite) model.

**Abstract operators** The abstract operators operate on subsets of the abstract type. We use $\perp$ to refer to the set of all abstract values. As an example, consider the *successor* operation on $x$, denoted $x + 1$, where $x$ is of type $T$. The successor of a number in the interval $[0, i)$ might be another value in $[0, i)$, or it might be $i$, since $[0, i)$ contains $i - 1$. The abstraction does not provide enough information to determine which is the answer. Thus, we say $\{[0, i)\} + 1 = \{[0, i), i\}$. The successor of $i$ is $i + 1$, which must be in the interval $(i, \infty)$. Thus,$\{i\} + 1 = \{(i, \infty)\}$. Further, $x + 1$ for any value of $x$ in $(i, \infty)$ is also in $(i, \infty)$. Thus, in the abstraction, $\{(i, \infty)\} + 1 = \{(i, \infty)\}$. When operating on non-singleton sets of abstract values, we simply define the operator so that it is linear with respect to set union. That is, for example, $(x \cup y) + 1 = (x + 1) \cup (y + 1)$.

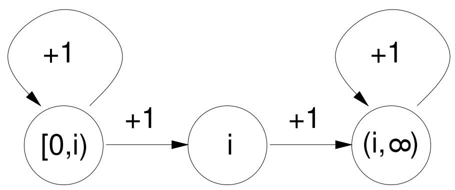


**Fig. 1.** Abstraction of the natural numbers

| $=$ | $[0,i)$ | $i$ | $(i,\infty)$ |
|---|---|---|---|
| $[0,i)$ | $\{0,1\}$ | 0 | 0 |
| $i$ | 0 | 1 | 0 |
| $(i,\infty)$ | 0 | 0 | $\{0,1\}$ |

| $<$ | $[0,i)$ | $i$ | $(i,\infty)$ |
|---|---|---|---|
| $[0,i)$ | $\{0,1\}$ | 1 | 1 |
| $i$ | 0 | 0 | 1 |
| $(i,\infty)$ | 0 | 0 | $\{0,1\}$ |

**Fig. 2.** Truth tables for abstract comparison operations

This abstract model of type $T$ is depicted in figure 1. As we increment a value of type $T$, it stays in the interval $[0,i)$ for some arbitrary amount of time, then shifts to $i$, then stays forever in the interval $(i,\infty)$. Note that this model is finite, and homomorphic to the natural numbers, where each number maps to itself, or the interval that contains it (in the extreme case $i = 0$, note that no value maps into the interval $[0,i)$, but the homomorphism still holds). Note that some of the abstract operators are too abstract to be truly useful. For example, we abstract $x + y$ to be simply $\perp$, and any operator on mixed types yields $\perp$.

Now consider the operators $x = y$ and $x < y$ over type $T$. Figure 2 shows suitable abstract truth tables for these operators. For example, any value in $[0,i)$ is less than $i$, so $\{[0,i)\} < \{i\}$ is true. Therefore, we let $\{[0,i)\} < \{i\}$ equal $\{1\}$. On the other hand, if both $x$ and $y$ are in $[0,i)$, then $x < y$ may be either true or false, depending on the choice of $x$ and $y$. Thus, let $\{[0,i)\} < \{[0,i)\}$ equal $\{0,1\}$. By suitable abstract definitions of all the operators in the logic, we maintain a homomorphism from the concrete to the abstract model. Thus, if the abstract model satisfies $p(i)$, then the concrete one does.

**Equivalence classes of abstractions** This abstraction solves the problem of non-finiteness of the model, but it still leaves us with an infinite set of values of $i$ to check. Note, however, that the abstraction of type $T$ has been chosen so that the abstract models for all values of $i$ are isomorphic. Thus, if we denote by $M_i$ the abstract model for parameter value $i$, then for any two values $x$ and $y$ of type $T$, $M_x$ satisfies $p(x)$ exactly when $M_y$ satisfies $p(y)$. It therefore suffices to verify that $M_i$ satisfies $p(i)$ for just one fixed value of $i$ to infer $\forall i.\ p(i)$. Of course, the abstract model may yield a false negative. However, we know at least that all values of $i$ will give the same truth value.

Now, suppose that we have more than one parameter to deal with. For example, we may want to prove $\forall i, j. p(i,j)$, where $i$ and $j$ range over type $T$. In this case, we introduce more than one concrete value into the abstract type. We do

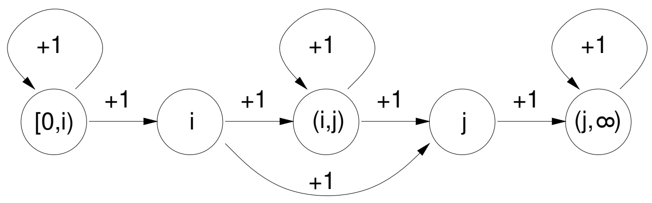


**Fig. 3.** Abstraction with two parameters

not, however, expect all the abstract models to be isomorphic. Rather, we must distinguish three cases: $i < j$, $i = j$ and $j < i$. In the first case, for example, we can abstract the type $T$ to the set

$$\{[0, i), i, (i, j), j, (j, \infty)\}.$$

The abstract model of type $T$ would then be as pictured in figure 3. Note in particular that the value of $i + 1$ here can be either $(i, j)$ or $j$. This is because the the interval $(i, j)$ can be either empty or non-empty, and both cases must be accounted for. The case where $j < i$ is the same, with the roles of $i$ and $j$ reversed, and the case $i = j$ is the same as for the simpler problem above (figure 1). Within each of these three classes, the abstract models are all isomorphic. This schema can be extended to any number of parameters. However, the number of equivalence classes increases exponentially with the number of parameters. In general, we must consider as a separate class each feasible theory of inequality over the given parameters.

**Introduction of fixed constants** Generally, proofs by induction refer to one or more fixed constants. For example, the base case (typically 0) often occurs as a constant in the formula $p$, and other constants may occur as well, as for example the upper bound of an array. Note that in the above system, constants must be abstracted to $\perp$, otherwise the isomorphism between cases is lost. However, it is possible to treat fixed constants in the same way as parameters in the abstract model. The only difference is that certain inequality classes may be vacuous. For example, suppose that we wish to prove $\forall i.\ p(i)$, and the formula $p(i)$ contains the constant 0. We have three classes to consider here: $i = 0$, $i < 0$ and $0 < i$. However, since $i$ ranges over the natural numbers, the case $i < 0$ is infeasible, thus we have only two feasible cases to check. Note that the number of values in the abstract type may sometimes be reduced when fixed constants are involved. For example, for the case where $0 < i$, we could abstract the type $T$ to the set $\{[0, 0), 0, (0, i), i, (i, \infty)\}$. Since the interval $[0, 0)$ is empty, it can be dropped from the abstraction.

In a proof by induction over the parameter $i$, it is clearly important to be able to refer to the value $i - 1$. That is, the formula $p(i)$ that we wish to prove is typically of the form $q(i - 1) \Rightarrow q(i)$. If the formula $i - 1$ yielded an abstract constant, the proof would be unlikely to succeed. However, we can also include terms of the form $i + c$, where $c$ is a fixed value, in the abstraction without

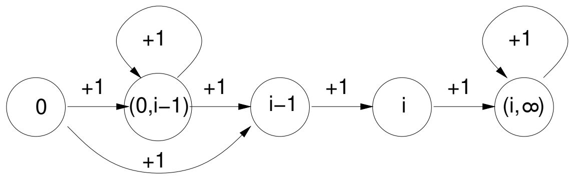


**Fig. 4.** Abstract type for induction proof

breaking the isomorphism. It is simply a matter of enumerating all the feasible inequality relations and eliminating the interval constants that are necessarily empty. For example, suppose we choose to an abstraction using the terms 0, $i$ and $i-1$. There are three feasible inequality relations among these terms:

$$i - 1 < 0 = i$$

$$0 = i - 1 < i$$

$$0 < i - 1 < i$$

In the case $0 < i-1 < i$, for example, the intervals $[0,0)$ and $(i-1,i)$ are necessarily empty. Thus, we abstract the type $T$ to the set $\{0, (0, i-1), i-1, i, (i, \infty)\}$, as shown in figure 4. To prove $p(i)$, we need only choose one value of $i$ satisfying each inequality relation (for example, $i = 0, 1, 2$) as all values satisfying a given relation yield isomorphic abstractions.

**Example** To see how this abstraction technique can be used in an inductive proof, suppose we have a counter that starts at 0 and increments at each time step by one. This can be represented by the following temporal formula $\phi$:

$$(x = 0) \wedge G(x' = x + 1).$$

Here, assume $x$ to be a variable of type $T$. We would like to prove that for all $i$ in $T$, eventually $x = i$. That is, $\forall i.q(i)$, where $q(i)$ is $F(x = i)$. By induction, it is sufficient to prove $\forall i.\ p(i)$, where

$$p(i) \equiv (\phi \wedge q(i-1)) \Rightarrow q(i)$$

(assuming we take $q(-1)$ to be equivalent to true). Using the above abstraction (containing the values 0, $i-1$ and $i$) we have three values of $i$ to check, each with a different abstraction of $T$. For example, the case $i = 2$ falls in the class $0 < i - 1 < i$ (fig 4). Note that if $x = i - 1$, then in this abstraction $x$ must be $i$ at the next time. Thus, if $F(x = i - 1)$, then $F(x = i)$, which is exactly what we want to prove. Since this is true in the abstract model, it must be true in the concrete model. The reader might want to consider the other two cases ($i = 0, 1$), and confirm that our property holds under these abstractions as well. This suffices to prove $p(i)$ for all $i$, since all of the remaining values of $i$ yield abstractions isomorphic to the case $i = 2$.

Here is how this problem would be entered in the SMV system:

```
init(x) := 0;
next(x) := x + 1;

forall(i in T) {
    q[i] : assert F (x = i);
    using q[i-1] prove q[i];
}
```

The system automatically chooses the values $0, i-1, i$ for the abstraction of type $T$ (though this can be chosen manually as well). The three feasible equality relations among these constants are enumerated automatically, and one representative value of $i$ from each relation is chosen. The induction step is then verified by model checking for each corresponding abstraction of type $T$ (this is possible because all are finite). SMV then reports that the property $q$ has been proved. How SMV recognizes that the induction itself is valid is the subject of the next section.

## 3 Induction principle

While the above technique allows us to prove that $q(i-1)$ implies $q(i)$, for all $i$, it does not tell us that this in fact implies $q(i)$ for all $i$. For this we need an induction rule in the system. While precisely this rule could easily be added to the system, we will consider here a more general scheme, that allows mutual induction over multiple induction parameters. This will be illustrated in section 5 using the "bakery" mutual exclusion algorithm.

To present this, we must first consider how a proof is represented in the SMV system. A proof is a graph, in which the nodes are properties and an edge from property $p$ to property $q$ indicates that $p$ is used to prove $q$. Proof edges are suggested to the prover by a statement like the following:

**using** $p$ **prove** $q$;

The proof graph must be well founded, *i.e.*, have no infinite backward paths. A parameterized property is considered to be equivalent to the set of its ground instances. Thus, for example, the property

**forall**($i$ **in** $T$) $q[i]$ : **assert** $\mathbf{F}(x = i)$;

is considered to be a shorthand for the infinite set of properties

$q[0]$ : **assert** $\mathbf{F}(x = 0)$; $q[1]$ : **assert** $\mathbf{F}(x = 1)$; ...

Similarly, a parameterized proof graph statement, such as

**forall**($i$ **in** $T$) **using** $q[i\text{-}1]$ **prove** $q[i]$;

is considered to be equivalent to the infinite set of statements:

**using** $q[-1]$ **prove** $q[0]$; **using** $q[0]$ **prove** $q[1]$; ...

Note, $q[-1]$ is considered to be equivalent to "true". In the case of finitely bounded parameters, SMV can simply construct the proof graph consisting of all of the ground instances and check that there are no cycles. However, for infinite types this is clearly not possible, and for large types it is impractical. Instead, an abstract proof graph is constructed, in which the nodes are parameterized properties and an arc exists between two such properties when there is an arc between any ground instances of the properties. This graph is homomorphic to the graph of ground instances, thus well-foundedness of the former is a sufficient (but not necessary) condition for well-foundedness of the latter.

Unfortunately, this does not allow for inductive proofs. That is, the statement

**forall**($i$ **in** $T$) **using** $q[i\text{-}1]$ **prove** $q[i]$;

implies an edge from $q[i]$ to itself in the abstract proof graph, and hence a cycle. We need an abstraction of the proof graph that preserves well foundedness of inductive proofs. We can obtain this based on the following observation: an infinite path in a graph either is a *simple path* (with no repeated nodes), or it contains a cycle. Thus, if we can prove that every node in the graph is neither on a cycle, nor the root of an infinite simple (backward) path, we know the graph is well founded. The key is that we can show this using a *different* abstraction for each node.

That is, let $G = (V, E)$ be the (possibly infinite) proof graph on the set ground instances $V$, let $P = \{P_1, \ldots, P_n\}$ be a *finite* partition of $V$, where each $P_i$ is the set of ground instances of property $p_i$, and for each instance $v \in V$, let $H_v = (V_v, E_v)$ be a *finite* graph (the abstract proof graph for $v$), such that there is a homomorphism $h_v : V \to V_v$ from $G$ to $H_v$.

**Theorem 1.** *If there is an infinite backward path in* $G$*, then for some* $1 \leq j \leq n$ *and* $v \in P_j$*,*

- $h_v(v)$ *is on a cycle in* $H_v$, or
- *there is a backward path in* $H_v$ *from* $h_v(v)$ *to some* $w \in V_v$ *such that for infinitely many* $w' \in P_j$, $h_v(w') = w$.

**Proof.** Let $\sigma = s_0, s_1, \ldots$ be an infinite backward path in $G$. This path uses either a finite or an infinite subset of $V$. In the first case, let $v$ be some repeated element of $\sigma$. There is a cycle in $G$ containing $v$, hence a cycle in $H_v$ containing $h_v(v)$ (since $h_v$ is a homomorphism from $G$ to $H_v$). In the second case, $\sigma$ must use an infinite number of elements of some $P_j$. Let $v$ be some element of $P_j$ on $\sigma$, and let $\sigma'$ be the tail of $\sigma$ beginning with $v$. Since $V_v$ is finite, it follows that $h_v$ maps an infinite number of elements $w' \in P_j$ of $\sigma'$ to some $w \in V_v$. □

By the theorem, it suffices to check each ground instance of a property in one abstract proof graph, verifying that its image in the abstract graph is neither on a cycle, nor on a backward path to an node abstracting an infinite set of instances of the same property. In practice, we can use the same abstraction of the natural numbers to generate the abstract proof graph $H_v$ that we use to verify instance $v$. This means that the abstract proof graphs within an inequality class are isomorphic, hence we need only check one representative of each class.

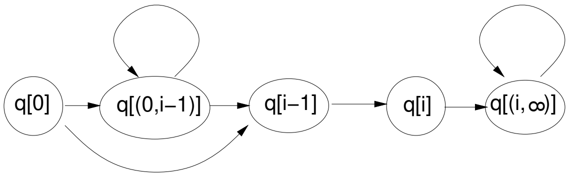


**Fig. 5.** An abstract proof graph

**Example** Consider our proof from the previous section. There are three cases of feasible inequality relations to consider. In the case $0 < i-1 < i$, we abstract type $T$ to the set $\{0, (0, i-1), i-1, i, (i, \infty)\}$. Substituting these abstract values into the proof graph statement, we obtain the abstract proof graph of figure 5. We must check two properties of this graph:

1. The node $q[i]$ is not on a cycle.
2. No backward path rooted at $q[i]$ reaches an instance of $q$ containing an infinite interval.

Since the abstract graphs are isomorphic for all values of $i$ satisfying $0 < i-1 < i$, we need check only one instance in this class (say, $i = 2$). In fact, we can verify the above properties for one representative of each inequality class, so we conclude that the proof graph is acyclic.

On the other hand, suppose we had written instead:

**forall**($i$ **in** $T$) **using** $q[i{+}1]$ **prove** $q[i]$;

In the abstract proof graph, we would find a backward path $q[i]$, $q[i+1]$, $q[(i+1,\infty)]$. Since the last of these contains an infinite interval, we reject this graph. In fact, there is an infinite backward chain in the proof, so the proof is not valid.

Using this technique, we can handle more general induction schema than the simple example above. For example, we can use mutual induction over two invariants:

... **using** $p[i]$ **prove** $q[i]$;
... **using** $q[i\text{-}1]$ **prove** $p[i]$;

Or, we can use induction simultaneously over two parameters:

... **using** $p[i\text{-}1][j]$, $p[i][j\text{-}1]$ **prove** $p[i][j]$;

(here, imagine a grid in which each cell is proved using its neighbors to the left and below). In section 5, we will see an example of induction over multiple parameters.

## 4 A FIFO buffer

As a simple example of proof by induction using model checking, we verify a hardware implementation of a FIFO buffer. We can decompose this problem into two parts. The first is to show that each data item output by the FIFO is correct, and the second is to show that the data items are output in the correct order. This separation is done by tagging each data item with an index number indicating its order at the input. Since only the ordering problem requires induction, we consider only that part of the proof here.

The input *inp* and output *out* of our FIFO have three fields: *valid*, a boolean indicating valid data at the interface, *data*, the actual data item, and *idx*, the index number of the data item. To specify ordering at either interface, we simply introduce a counter *cnt* that counts the number of items received or transmitted thus far. If there is valid data at the interface, we specify that the index of that data is equal to *cnt*. This counter ranges over an *ordset* type called *INDEX*.

$\mathit{cnt} : \mathit{INDEX};$
$\mathbf{init}(\mathit{cnt}) := 0;$
$\mathbf{if}(\mathit{valid})\ \mathbf{next}(\mathit{cnt}) := \mathit{cnt} + 1;$

$\mathit{ordered}$: $\mathbf{assert\ G}\ (\mathit{valid} \Rightarrow \mathit{idx} = \mathit{cnt});$

We create an instance of this specification for both input and output of the FIFO, and would like to prove that the property *ordered* at the input implies *ordered* at the output.

Let us implement the FIFO as a circular buffer, with a large number of entries (say, 1024) so that it is too large to verify temporal properties of it directly by model checking. To solve this problem, we need to abstract the type of buffer addresses as well. Thus, we create another *ordset* type and declare the type of buffer addresses *ADDR* to be the subrange $0 \ldots 1023$ of this type. Note that there are two fixed constants of this type that appear in the code: 0 and 1023. For example, to increment the head pointer in the circular buffer, the following code is used:

$\mathbf{next}(\mathit{head}) := (\mathit{head} = 1023)\ ?\ 0 : \mathit{head} + 1;$

Now, we can prove the *ordered* property at the output by induction on the value of *cnt*. We prove that if the last item output was numbered $i-1$, then the current one is numbered $i$. The natural abstraction needed to prove this would use the values 0, $i-1$ and $i$ (we need 0, since the initial value must be zero). In fact, SMV chooses this abstraction by default. Note that if the item indexed $i$ gets stored in address $j$ of the circular buffer, then item $i-1$ must have been stored at address $j-1 \bmod 1024$. Thus, if we analyze cases on $j$, we only have to consider two addresses in the array to prove the inductive step. To do the case analysis in SMV, we say:

$\mathbf{forall}(i\ \mathbf{in}\ \mathit{INDEX})\ \mathbf{forall}(j\ \mathbf{in}\ \mathit{ADDR})$
$\quad \mathbf{subcase}\ \mathit{out.ordered}[i][j]\ \mathbf{of}\ \mathit{out.ordered}$
$\quad\quad \mathbf{for}\ \mathit{out.cnt} = i \wedge \mathit{tail} = j;$

The property *ordered*[*i*][*j*] says that, if the current *cnt* at the output is $i$, and the current value of the tail pointer (*i.e.*, the address of the current output) is $j$, then the current index at the output must be *cnt*. Now, we use induction over $i$ in the following way:

**forall**(*i* **in** *INDEX*) **forall**(*j* **in** *ADDR*)
  **using** *ADDR*$\Rightarrow\{0,j\text{-}1,j,1023\}$, *inp.ordered*, *out.ordered*[*i*-1]
    **prove** *out.ordered*[*i*][*j*];

In the abstracted models, there are maximally four elements in the buffer, the remainder being abstracted to $\perp$. With this abstraction, the buffer implementation can be model checked without difficulty. Note, we are not actually using induction over the type *ADDR* here. We are simply using the ordered set abstraction to make the model checking of a 1024 element buffer tractable. The use of this abstraction does increase the number of cases that must be checked, however. We must check three cases of $i$ times five cases of $j$ (the additional fixed constant of type *ADDR* increases the number of feasible ordering relations) for a total of fifteen. Nonetheless, all of these cases can be verified by SMV in less than two seconds.

We can also prove our buffer implementation for arbitrary depth by using an uninterpreted constant to represent the buffer depth. This would be of secondary interest in hardware verification, however, where resources are fixed and finite.

## 5 Bakery mutual exclusion algorithm

Leslie Lamport's "bakery" algorithm [Lam74] is a mutual exclusion algorithm for a set of $N$ processes, each running a program with two parts, a *critical section* and a *noncritical section*. We have adapted from Lamport's original presentation with minor changes for the present exposition.[2] For each process $i$, the algorithm uses a boolean variable $choosing[i]$, natural number variables $number[i]$ and $max[i]$, and a variable $count[i]$ that ranges over process id's $1, \ldots, N$. We denote the type of natural numbers by `NAT`, and the type of process id's by `PID`. These can all be read and written by process $i$. In addition, $choosing[i]$, and $number[i]$ are readable, but not writable by other processes. Each process $i$ starts execution in its noncritical section with $choosing[i]$ and $number[i]$ initially equal to 0. The following pseudo-code (not SMV input) gives the program for process $i$:

L1: {noncritical section; nondeterministically **goto** L1 or L2;}
L2: *choosing*[*i*] := 1;
L3: {*count*[*i*] := 1; *max*[*i*] := 0;}
L4:   *max*[*i*] := maximum(*max*[*i*], *number*[*count*[*i*]]);
L5:   **if** *count*[*i*] < N **then** {*count*[*i*] := *count*[*i*]+1; **goto** L4;}

[2] The proof described here disregards two important features of the algorithm: the non-atomicity of reads and writes, and the possibility of crashes. SMV files for this version and the general algorithm can be found at
`http://www-cad.eecs.berkeley.edu/~kenmcmil/bakery.html`.

```
L6:    number[i] := max[i]+1;
L7:    choosing[i] := 0;
L8:    count[i] := 1;
L9:        if choosing[count[i]] = 1 then goto L9;
L10:       if number[count[i]] ≠ 0 and
               (number[count[i]] < number[i] or
               (number[count[i]] = number[i] and count[i] < i))
           then goto L10;
L11:       if count[i] < N then {count[i] := count[i]+1; goto L9;}
L12:   critical section;
L13:   {number[i] := 0; goto L1;}
```

In lines L3–L6, process $i$ chooses a positive number *number*$[i]$; the ordered pair (*number*$[i], i$) serves as process $i$'s "ticket" to enter the critical section. In lines L8–L11, process $i$ loops over all processes $j$, waiting at L9 until process $j$ is not in the act of choosing a ticket, and waiting at L10 until process $j$ either does not have a ticket (*number*$[j] = 0$) or has a ticket that is greater (in lexicographic order) than process $i$'s ticket.

The algorithm has two key properties. First, it ensures *safety*: no two processes can ever be in the critical section (at L12) at the same time. Second, it guarantees *liveness* under an assumption of *fairness*: if every process continues to execute instructions (the fairness assumption), then any process that reaches L2 eventually gets to the critical section (the liveness guarantee). The fairness assumption includes the assumption that a process in the critical section will eventually leave the critical section. However, as indicated by the pseudo-code "nondeterministically goto L1 or L2", a process may remain in the noncritical section forever (and this does not interfere with the liveness of any other process).

To encode the bakery algorithm in SMV we introduce some additional variables. For each process $i$ the variable *pc*$[i]$ ($i$'s "program counter") ranges over the values $\{L1, L2, \ldots, L13\}$ and designates the next line to be executed by process $i$. The variable *act* takes on an arbitrary value in the range $1, \ldots, N$ at each time step and designates the process that executes a line at that time step. Thus the action taken at any time step depends on the value of *pc*[*act*]:

```
switch (pc[act]) {
    L1: { next(pc[act]) := {L1, L2}; }
    L2: { next(choosing[act]) := 1; next(pc[act]) := L3; }
    ...
    L13: { next(number[act]) := 0; next(pc[act]) := L1; } }
```

The code also includes initial conditions for the processes (not shown here) and the fairness condition:

```
forall (i in PID) fair[i]: assert G(F(act = i));
```

The safety and liveness properties of the algorithm can now be defined as follows:

**forall** ($i$ **in** *PID*) **forall** ($j$ **in** *PID*) {
  *safe*[$i$][$j$]: **assert** $\mathbf{G}(i \neq j \Rightarrow \neg(pc[i] = L12 \wedge pc[j] = L12))$;
  *live*[$i$]: **assert** $\mathbf{G}(pc[i] = L2 \Rightarrow \mathbf{F}(pc[i] = L12))$; }

In the limited remaining space, we concentrate on sketching the liveness proof, which is more interesting and difficult than the safety proof. The liveness property is easily proved using the following two lemmas, the first saying that any process that starts the loop in lines L3–L5 eventually completes it, and the second saying the same for the loop in lines L8–L11:

**forall** ($i$ **in** *PID*) {
  *reaches_L6*[$i$]: **assert** $\mathbf{G}(pc[i] = L3 \Rightarrow \mathbf{F}(pc[i] = L6))$;
  *reaches_L12*[$i$]: **assert** $\mathbf{G}(pc[i] = L8 \Rightarrow \mathbf{F}(pc[i] = L12))$;
  **using** *reaches_L6*[$i$], *reaches_L12*[$i$], *fair*[$i$] **prove** *live*[$i$]; }

Note that the proof of *live*[$i$] relies on the fairness assumption *fair*[$i$] to ensure that process $i$ eventually gets from *L2* to *L3* and from *L6* to *L8*. Of course, the assumption of fairness is also essential for the proofs of the lemmas *reaches_L6* and *reaches_L12*.

To prove that the loop in *L3*–*L5* terminates (once it is started), we use a helper lemma, stating that the $j$-th iteration is eventually completed, for any process id $i$. The helper lemma is proved by straightforward induction: eventual completion of the $j$-th iteration follows from completion of the $(j-1)$-th iteration, together with the fairness assumption. Completion of the entire loop then follows from fairness and completion of the last iteration. Here is the SMV code:

**forall** ($i$ **in** *PID*) **forall** ($j$ **in** *PID*) {
  *reaches_L5*[$i$][$j$]: **assert** $\mathbf{G}(pc[i] = L3 \Rightarrow \mathbf{F}(pc[i] = L5 \wedge count[i] = j))$;
  **using** *reaches_L5*[$i$][$j$-1], *fair*[$i$] **prove** *reaches_L5*[$i$][$j$];
  **using** *reaches_L5*[$i$][$N$], *fair*[$i$] **prove** *reaches_L6*[$i$]; }

The termination proof for the loop in lines L8–L11 (that is, for the lemma *reaches_L12*) is more difficult in that it involves induction not only over iterations of the loop, but also over tickets. To prove that a process $i$ completes the loop, we use the induction hypothesis that any process $j$ with a lower ticket eventually reaches (and by fairness, eventually leaves) the critical section, and so cannot cause process $i$ to wait forever at *L10*. We use the following three lemmas:

**forall** ($n$ **in** *NAT*) **forall** ($i$ **in** *PID*) **forall** ($j$ **in** *PID*) {
  *reaches_L11*[$n$][$i$][$j$]:
    **assert** $\mathbf{G}(pc[i] = L8 \wedge number[i] = n \Rightarrow \mathbf{F}(pc[i] = L11 \wedge count[i] = j))$;
  *lower_number_reaches_L12*[$n$][$j$]:
    **assert** $\mathbf{G}(number[j] < n \wedge pc[j] = L8 \Rightarrow \mathbf{F}(pc[j] = L12))$;
  *lower_pid_reaches_L12*[$n$][$i$][$j$]:
    **assert** $\mathbf{G}(number[j] = n \wedge j < i \wedge pc[j] = L8 \Rightarrow \mathbf{F}(pc[j] = L12))$; }

The first states that a process with ticket $(n, i)$ eventually completes the $j$-th iteration of the loop. The other two together state the induction hypothesis that any process $j$ with a ticket lexicographically less than $(n, i)$ eventually completes the entire loop. We prove these three lemmas by mutual induction, over three parameters $n$, $i$ and $j$, where $(n, i)$ is the ticket and $j$ is the value of the loop counter. For example, induction over $j$ can be seen in the SMV command to prove *reaches_L11*:

**using** *fair*[*i*], *fair*[*j*], *reaches_L11*[*n*][*i*][*j*-1],
  *lower_pid_reaches_L12*[*n*][*i*][*j*], *lower_number_reaches_L12*[*n*][*j*],
  *reaches_L6*[*j*], *NAT*⇒{0,*n*}, ...
**prove** *reaches_L11*[*n*][*i*][*j*];

To prove that process $i$ eventually finishes waiting for process $j$, we assume that if process $j$ has a lower ticket it will eventually reach *L12*. The next time around the loop, process $j$ must choose a larger ticket. Note we also need *reaches_L6*, to ensure that process $j$ cannot make process $i$ wait forever at *L9*. By writing $NAT \rightarrow \{0, n\}$, we specify that the abstraction for type *NAT* keep 0 as a distinguished value, lest we get an abstraction too coarse to model the interaction of the assignment at *L13* with the test at line *L10*.

We omit here the similar, but simpler, proofs of the other two lemmas, which contain the inductive steps over $n$ and $i$ respectively. We remark simply that the three lemmas are mutually dependent, and that well-foundedness of the entire mutual induction is automatically checked by SMV, as described in section 3. We also omit the command to prove that *reaches_L12* follows from *lower_number_reaches_L12*.

Note that we could have combined lemma *lower_number_reaches_L12* and lemma *lower_pid_reaches_L12* (or even these two and *reaches_L11*) into a single lemma. However, the proof would then have involved temporal case-splitting over four (or more) variables at once, rather than three. Increasing the number of simultaneous case splits increases the number of concrete values in the abstract types, thus making the proof inefficient in two ways. First, the resulting finer abstractions tend to be individually more costly to model-check. Second, more distinct abstractions must be checked in order to check a representative of every equivalence class. In general, it is desirable to structure proofs so that the abstractions used are as coarse as possible (but no coarser).

## 6 Conclusion

By using an appropriate parameterized abstraction of the natural numbers, we can reduce a proof by induction over the natural numbers to a finite number of finite state verification problems. This is because, on the one hand, each abstracted type is finite, and on the other hand, the infinite set of abstractions falls into a finite number of isomorphism classes. As a result, we can check proofs by model checking that would ordinarily be considered to be only in the domain of theorem provers. The advantage is that, in a proof by model checking, we need not consider the details of the control of a system, since these are handled by state enumeration. This technique has been implemented in the SMV proof assistant, and integrated with a variety of techniques for reducing infinite state problems to finite state problems.

This technique is more general than previous techniques that use model checking as part of an induction proof [KM89,WL89] in that variables may range over unbounded types, and the proof is fully mechanically checked. Further, a novel induction scheme based on proof graph abstractions makes it possible to do fairly complex proofs using mutual induction over multiple variables, without explicit recourse to an induction rule, as the induction scheme is inferred by analyzing the abstract proof graphs.

The method is not fully automated, in that it relies on the user to supply inductive properties, which may have to be stronger than the property being proved. Note, however, that it is not required to provide an inductive *temporal* invariant, since the model checker can, in effect, compute the strongest invariants of the abstract models. Since linear temporal logic with natural number variables is undecidable (by a trivial reduction from termination of a two-counter machine), the method described here is necessarily incomplete.

A practical problem with the method is that the number of cases (*i.e.*, feasible inequality relations) expands very rapidly with the number of parameters and fixed constants of a given type. Thus, in the proof of the bakery algorithm, for example, considerable care had to be used to minimize the number of parameters in the lemmas. Clearly, techniques of reducing this case explosion (perhaps using weaker abstractions) would make the technique easier to apply.

Finally, we note that the proof of the bakery algorithm, which relies substantially on properties of the natural numbers, was more difficult than proofs of considerably more complex systems (*e.g.* [McM99]) that are control dominated. Nevertheless, we think this example shows that model checking can be used to advantage even in an area where finite control plays a relatively small part. This suggests that there may be many areas in which model checking can be applied that previously have not been considered amenable to finite state methods.

## References


[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In A.J. Hu and M.Y. Vardi, editors, *Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 319–331. Springer-Verlag, 1998.

[DDP99] S. Das, D.L. Dill, and S. Park. Experience with predicate abstraction. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 160–171. Springer-Verlag, 1999.

[HQR99] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 301–315. Springer-Verlag, 1999.

[KM89] R. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 239–247, 1989.

[Lam74] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Comm. ACM*, 17:453–455, August 1974.

[McM97] K. L. McMillan. A compositional rule for hardware design refinement. In O. Grumberg, editor, *Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 24–35. Springer-Verlag, 1997.

[McM98] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 110–121. Springer-Verlag, 1998.

[McM99] K. L. McMillan. Verification of infinite state systems by compositional model checking. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *LNCS*, pages 219–233. Springer-Verlag, 1999.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Computer Aided Deduction (CADE'92)*, volume 607 of *LNAI*, pages 748–752. Springer-Verlag, 1992.

[RSS95] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In P. Wolper, editor, *Computer Aided Verification (CAV'95)*, volume 939 of *LNCS*, pages 84–97. Springer-Verlag, 1995.

[SG97] H. Saïdi and S. Graf. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, 1997.

[SS99] H. Saïdi and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 443–454. Springer-Verlag, 1999.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 68–80. Springer-Verlag, 1989.