“Next” Heuristic for On-the-fly Model Checking

Rajeev Alur* and Bow-Yaw Wang**

Abstract. We present a new heuristic for on-the-fly enumerative invariant verification. The heuristic is based on a construct for temporal scaling, called next, that compresses a sequence of transitions leading to a given target set into a single metatransition. First, we give an on-the-fly algorithm to search a process expression built using the constructs of hiding, parallel composition, and temporal scaling. Second, we show that as long the target set Θ of transitions includes all transitions that access variables shared with the environment, the process $\mathsf{next}\ \Theta\ \mathsf{for}\ P$ and P are equivalent according to the weak-simulation equivalence. As a result, to search the product of given processes, we can cluster processes into groups with as little communication among them as possible, and compose the groups only after applying appropriate hiding and temporal scaling operators. Applying this process recursively gives an expression that has multiple nested applications of next, and has potentially much fewer states than the original product. We report on an implementation, and show significant reductions for a tree-structured parity computer and a ring-structured leader-election protocol.

1 Introduction

Model checking [CE81,QS82,CK96,Hol97] has proved to be a useful technique for automatic debugging of high-level designs of hardware and protocols. Model checking requires search of the global state-space of the design, and since the number of global states increases exponentially with the size of the description, model checking tools must employ a variety of heuristics to battle this so-called *state explosion problem*. In this paper, we present a new heuristic for enumerative verification of invariant requirements. Before we describe our approach, let us briefly review two techniques, on-the-fly search of reachable state-space, as implemented in tools such as SPIN [Hol97] and Murφ [DDHY92], and reduction techniques based on process equivalences, as implemented in tools such as *Concurrency workbench* [CPS93] and CADP [FGK+96].

In on-the-fly model checking, the system is described as a collection of communicating processes. The global state-transition graph of the system is explored on demand starting from initial states using a systematic search algorithm. Only the states, and not the transitions, are stored in a global table, and as soon as

* Department of Computer and Information Science, University of Pennsylvania, and Bell Laboratories, Lucent Technologies. Email: alur@cis.upenn.edu. Supported in part by NSF CAREER award CCR-9734115 and by the DARPA grant NAG2-1214.
** Department of Computer and Information Science, University of Pennsylvania. Email: bywang@saul.cis.upenn.edu. Supported by the DARPA grant NAG2-1214.

a violation of the invariant is encountered, a counter-example is reported to the user. A variety of heuristics have been shown to be effective in restricting the search to a subset of the global state-space. Sample heuristics are reductions using symmetries [ES93,CFJ93,ID96] and partial orders [Pel94,Val92,God90].

In the so-called compositional approaches, the system is described using a richer language (e.g. CCS [Mil80]) that supports operators such as parallel composition and hiding. Compositional techniques rely on the definition of a process equivalence (e.g. weak bisimulation) and algorithms to reduce a process with respect to this equivalence [KS90,PT87,CPS93,BFH90,BRdSV87]. Starting with a complex expression, subexpressions can then be replaced by equivalent smaller ones in a bottom-up fashion. The order in which processes are composed reflects the connectivity, and a judicious use of hiding allows greater reductions by making more transitions internal at each step. For well-structured designs such as trees and rings, this approach can yield significant reductions in principle. However, it does not guarantee early detection of violations which is extremely important in typical applications of model checking. Furthermore, processing of subexpressions can be expensive. First, minimization with respect to an equivalence is typically $O(n \log n)$ or $O(n^2)$, where n is the number of states in the unreduced graph. While minimization can be replaced by less expensive heuristic reduction algorithms, as far as we know, all the tools explicitly build the state-transition graphs for the subexpressions. Second, a subexpression, when considered on its own, has lot more states than when considered in the context of the whole expression. Heuristic approaches to abstract the context by assumptions during reduction have been proposed [KM97], but offer only a partial remedy to this problem.

In this paper, we propose an on-the-fly search algorithm that takes into account the hierarchical structure of architecture of the system. The reduction performed by the algorithm on subexpressions compresses internal transitions, and preserves the weak-simulation equivalence. While the basic reduction strategy is simple and well known, the novelty lies in applying it on demand in a hierarchical manner.

We develop our search algorithm in the more general setting of explicitly structured transition relations. Our process model, in addition to the standard operations of *hiding* and *parallel composition*, employs a construct for hierarchically structuring transition relations. For a process P and a set Θ of target transitions, the process $\mathsf{next}\ \Theta\ \mathsf{for}\ P$ is obtained by compressing a sequence of transitions of P ending in a Θ-transition into a single metatransition. The next operator is a temporal scaling operator inspired by notions of multiform time in synchronous languages [BG88,Hal93], and was introduced in the language *reactive modules* [AH96]. The parallel composition and temporal scaling can be mixed freely giving complex processes. Note that applications of next can cause significant reductions in the global state-space by ruling out transient states. For instance, the expression ($\mathsf{next}\ \Theta\ \mathsf{for}\ P \,\|\, \mathsf{next}\ \Sigma\ \mathsf{for}\ Q$) can have much fewer reachable states than $P\|Q$: to explore the state-space of $P\|Q$, at every step either P or Q is allowed to take a step, while to explore the expression

($\mathsf{next}\ \Theta\ \mathsf{for}\ P \parallel \mathsf{next}\ \Sigma\ \mathsf{for}\ Q$), once P is chosen, it is executed until it takes a Θ-transition, and once Q is scheduled, it is executed until it takes a Σ-transition.

The first challenge is to design an on-the-fly algorithm to search a process expression built using parallel composition and temporal scaling. Such an algorithm is presented in Section 3. It generates global states only on demand, and can do early reporting of violations of invariants. To avoid recomputation, it stores in the global hash-table, along with the global states, transient states generated during processing of next operator. It is guaranteed to visit every reachable state of the given expression, and visits no more, and typically much less, than the reachable states of the flattened expression obtained by removing the applications of next. The running time is linear in the number of states (more precisely, the transitions) visited by the algorithm, and there is basically no overhead in applying the reduction.

Then, we proceed to identify restriction under which the answer to the invariant verification problem is preserved by the application of next. Along the lines of well known results concerning compressing internal transitions (cf. [BRdSV87]), we establish that if Θ includes those transitions that read or write shared variables then the processes P and $\mathsf{next}\ \Theta\ \mathsf{for}\ P$ are equivalent according to the *weak-simulation* relation. Since weak-simulation is shown to be a congruence with respect to all the operators in our language, we can substitute freely and repeatedly any subexpression with next applied to it. Thus, given a problem to search the composition of a collection of processes, we can heuristically build an expression that clusters the given processes to limit the shared variables across clusters, and apply hiding and temporal scaling to individual clusters before composing them. This process is applied recursively to maximize the reduction. The resulting expression is then searched using the algorithm of Section 3.

The heuristic is demonstrated on a couple of benchmark examples using a prototype implementation. The first example concerns a tree-structured system which computes the parity of inputs presented at the leaves. This example is ideally suited for our reduction: while the number of reachable states of the global system increases exponentially with the size of the tree, the number of reachable states of the reduced search increases almost linearly. The second example concerns a standard leader election algorithm in a ring network. While the reduction is less dramatic, it is still significant for both space and time. The experimental results also support our analysis that the running time is proportional to the number of states visited.

The rest of the paper is organized as follows. Section 2 presents our process model. Section 3 gives the search algorithm for process expressions employing next and parallel composition. Section 4 presents the heuristic application of next and justifies its correctness using the theory of simulation equivalences. Section 5 gives experimental results. Finally, in Section 6, we conclude with comparison with other heuristics.

2 Process Model

We start with the definition of processes. Our process model uses interleaving semantics, and read-shared exclusive-write variables. The model is a simplification of *reactive modules* [AH96].

Given a set X of typed variables, a *state* over X is a function mapping variables to their values. For a state s, define $s[v/x]$ to be the state obtained from s by replacing the value of x by v.

A process is defined by the set of its variables, rules for initializing the variables, and rules for updating the variables. The variables of a process P are partitioned into three classes: *private* variables that cannot be read or written by other processes, *interface* variables that are written only by P, but can be read by other processes, and *external* variables that can only be read by P, and are written by other processes. Thus, interface and external variables are used for communication, and are called *observable* variables. The process controls its private and interface variables, and its environment controls the external variables. Once the variables of a process are specified, the state space of the process is determined. A state is also partitioned into different components as the variables are (e.g., controlled state and external state). The initialization specifies initial controlled states, and the transition relation specifies how to change the controlled state as a function of the current state.

Definition 1. *A* process P *is a tuple* (X, I, T) *where*

- $X = (X_p, X_i, X_e)$ *is the (typed) variable declaration.* X_p, X_i, X_e *represent the sets of* private variables, interface variables *and* external variables *respectively. Define the* controlled variables $X_c = X_p \cup X_i$ *and the* observable variables $X_o = X_i \cup X_e$*;*
- Q_c *is the set of* controlled states *over* X_c *and* Q_e *is the set of* external states *over* X_e. $Q = Q_c \times Q_e$ *is the set of states. We also define* Q_o *to be the set of* observable states *over* X_o*;*
- $I \subseteq Q_c$ *is the set of* initial states*;*
- $T \subseteq Q_c \times Q_e \times Q_c$ *is the* transition relation. *We use the notation* $q \xrightarrow{e} q'$ *for* $(q, e, q') \in T$. ■

A state q can be updated in two ways. If the process makes its move, it changes its controlled state according to the transition relation. If the environment makes its move, it can change the external state to any arbitrary value.

Definition 2. *Let* $P = ((X_p, X_i, X_e), I, T)$ *be a process, and* q, q' *be states. Then* q' *is a* successor *of* q*, written* $q \to q'$*, if*

- $q[X_e] = q'[X_e]$ *and* $q[X_c] \xrightarrow{q[X_e]} q'[X_c]$*; or*
- $q[X_c] = q'[X_c]$.

An execution *of* P *is a sequence of states in* Q^*, $q_0 q_1 \cdots q_n$*, where* $q_0[X_c] \in I$ *and* $q_i \to q_{i+1}$ *for* $0 \leq i < n$.

The reachable set *of* P*, denoted* $\mathcal{R}(P)$*, contains states* q *such that there is an execution ending in* q. ■

In order to support structured descriptions, we would like to build complex processes from simpler ones. Three constructs, hide H in P, $P \| P'$ and next Θ for P for building new processes are defined in our model as follows. The hide operator makes interface variables inaccessible to other processes.

Definition 3. *Let $P = ((X_p, X_i, X_e), I, T)$ be a process and $H \subseteq X_i$. Define the process* hide H in P *to be $((X_p \cup H, X_i \setminus H, X_e), I, T)$.* ∎

The parallel composition operator allows to combine two processes into a single one. The composition is defined only when the controlled variables of the two processes are disjoint. Intuitively, a step of $P \| Q$ is taken by either P or Q but not both.

Definition 4. *Let $P' = ((X'_p, X'_i, X'_e), I', T')$ and $P'' = ((X''_p, X''_i, X''_e), I'', T'')$ be processes where $X'_c \cap X''_c = \emptyset$. The* composition *$P$ of P' and P'', denoted $P' \| P''$, is defined as follows.*

- $X_p = X'_p \cup X''_p$;
- $X_i = X'_i \cup X''_i$;
- $X_e = (X'_e \cup X''_e) \setminus X_i$;
- $I = I' \times I''$;
- $T \subseteq Q_c \times Q_e \times Q_c$ *where* $(q, e, r) \in T$ *if*
  - $q[X'_c] \stackrel{(e \cup q)[X'_e]}{\longrightarrow} r[X'_c]$ *in* T' *and* $q[X''_c] = r[X''_c]$; *or*
  - $q[X''_c] \stackrel{(e \cup q)[X''_e]}{\longrightarrow} r[X''_c]$ *in* T'' *and* $q[X'_c] = r[X'_c]$. ∎

Finally, we introduce the next operator. The next operator is a *temporal* construct where new transitions are created by grouping a number of lower-level transitions. A transition in next Θ for P is a sequence of transitions in P where the last transition is in Θ. The external state is assumed to stay fixed during the sequence. By applying the next operator, a sequence of small steps becomes a big step. The next operator is inspired by the notions of multi-form time in synchronous languages [Hal93], and was introduced in the language *reactive modules* [AH96].

Definition 5. *Let $P = (X, I, T)$ be a process and Θ a subset of T. Define the process* next Θ for P *to be (X, I, T') where $(q, e, r) \in T'$ if there are states $q_0 = q, q_1, \ldots, q_n = r$ in Q_c such that $(q_i, e, q_{i+1}) \in T \setminus \Theta$ for $0 \leq i < n-1$ and $(q_{n-1}, e, q_n) \in \Theta$.* ∎

3 Search Algorithm

In this section, we will develop an on-the-fly search algorithm to explore reachable states of processes formed using the applications of parallel composition, hiding, and next. The applications of next nested with parallel composition lead to hierarchically structured transitions. The search algorithm proceeds recursively in the structure of the process. In order to present the algorithm, we distinguish atomic processes from composite ones. A process is called a *composite process* if it uses composition, hide or next operator. Otherwise, it is called an *atomic process.*

Definition 6. *A* process expression *is defined recursively as follows.*

$$M \equiv P \mid M \| M \mid \mathsf{hide}\ H\ \mathsf{in}\ M \mid \mathsf{next}\ \Theta\ \mathsf{for}\ M$$

where P is an atomic process, H is a set of variables and Θ is a set of transitions. ■

We will assume that a suitable syntax for atomic processes, such as guarded commands, has been chosen. We also assume that any atomic process appears in a process expression at most once. The algorithm uses the following pre-defined functions:

- The function *GetInitialStates(M)* returns the set of initial states of any process expression M.
- For an atomic process P and a state q in P, *AtomicGetNext(P, q)* returns the set of successors of q in P.
- The function *BelongsTo(s, t, Θ)* tests whether the transition from state s to state t is in Θ.

Since initial states are preserved by hide and next constructs, they can be computed by ignoring these operators. The function *AtomicGetNext(P, q)* is used to handle the basic case and is straightforward to implement as well.

Given a process expression, the algorithm of Figure 1 traverses reachable states in a depth-first manner. Since initial states are reachable, it calls the depth-first search subroutine on initial states iteratively. The algorithm uses a global hash table to store states. The depth-first search routine then uses the key function $GetNext(M, s)$ to compute the set of successors of s in process M. The function $GetNext(M, s)$ proceeds in a top-down fashion. Given a global state s, $GetNext(M, s)$ will return global states that are unvisited successors to s. It uses the pre-defined function $AtomicGetNext(P, s)$ for the atomic case. For the composition, it computes the successors of each component and returns their union. When it encounters a process defined by next operator, it generates the successors of the current state and checks the transition. If a target transition is found, the function adds new states to its returned result. Otherwise, it generates the successors of the new state repeatedly until a target transition is found.

To illustrate how the algorithm works, consider the processes P and Q shown in figure 2. Let P' and Q' denote $\mathsf{next}\ \Theta\ \mathsf{for}\ P$ and $\mathsf{next}\ \Sigma\ \mathsf{for}\ Q$ respectively, where Θ and Σ consist of thick transitions (i.e. $\Theta = \{(0,1),(3,4)\}$ and $\Sigma = \{(B,C)\}$). The product $P\|Q$ is also shown in the figure, and has 15 states. The product $P'\|Q'$ has much fewer states, namely, 6, shown as filled circles. This illustrates that the application of next rules out many transient states, and can potentially save space. The challenge is to explore the expression in a top-down manner computing metatransitions only on demand.

```
proc Main(M) ≡
  tbl := NewHashTable();
  foreach s ∈ GetInitialStates(M) do
    if s ∉ tbl then Insert(tbl, s); DFS(M, s); fi
  od.
proc DFS(M, s) ≡
  foreach s' ∈ GetNext(M, s) do DFS(M, s') od.
funct GetNext(M, s) ≡
  if M ≡ P
    then result := ∅;
         foreach s' ∈ AtomicGetNext(P, s[P]) do
           if s[s'/X_P] ∉ tbl
             then Insert(result, s[s'/X_P]);
                  Insert(tbl, s[s'/X_P])
           fi
         od;
         return result
  elsif M ≡ hide H in M'
       then return GetNext(M', s)
  elsif M ≡ M_1||M_2
       then return GetNext(M_1, s) ∪ GetNext(M_2, s);
  elsif M ≡ next Θ for M'
       then result := ∅;
            foreach s' ∈ GetNext(M', s) do
              if BelongsTo(s, s', Θ)
                then Insert(result, s')
                 else Insert(result, GetNext(M, s'))
              fi;
            od
            return result
  fi.
```

Fig. 1. Search Algorithm

Suppose the algorithm is exploring reachable states of $P'\|Q'$. First, it visits the initial state $0A$. After the first invocation of $GetNext(P'\|Q', 0A)$, it returns the states $1A$, $4A$ and $0C$ and the hash table contains these states as well as light gray states such as 2A and 3A. It is crucial that the transient states such as 2A are stored in the hash-table to avoid recomputation.

Next, the algorithm searches from $1A$. It ignores successor state $2A$ since it is in the hash table, returns state $1C$ and puts state $1B$ in the hash table. Then, it tries to explore state $4A$. It in turn returns state $4C$ and puts state $4B$ in the table. When the algorithm processes the state $0C$, it adds states $2C$ and $3C$ to the hash table. However, since states $1C$ and $4C$ have been visited, it will ignore them. At the end, the algorithm returns black states, puts shaded states in the hash table. Note that the algorithm never visits the states 2B and 3B, and thus, does less work than required to search $P\|Q$.

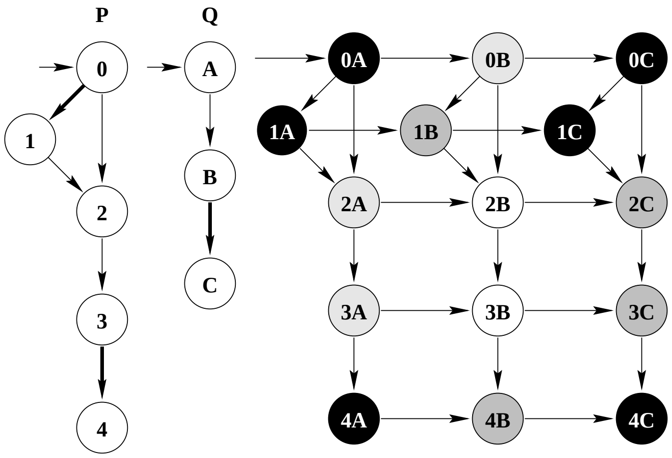

Fig. 2. Traversal Example

To analyze the algorithm, given a process expression M, define $Strip(M)$ to be the process expression obtained by removing all hide and next operators in M. More formally, we have

Definition 7. *Let M be a process expression. Define $Strip(M)$ recursively as follows.*

$$\begin{aligned} Strip(P) &= P \\ Strip(\textsf{hide } H \textsf{ in } M) &= Strip(M) \\ Strip(\textsf{next } \Theta \textsf{ for } M) &= Strip(M) \\ Strip(M_1 \| M_2) &= Strip(M_1) \| Strip(M_2) \end{aligned}$$

where P is an atomic process, H is a set of variables and Θ is a set of transitions.

For any process expression M, consider two reachable sets $\mathcal{R}(M)$ and $\mathcal{R}(Strip(M))$. Clearly, for each state $s \in \mathcal{R}(M)$, $s \in \mathcal{R}(Strip(M))$ because $Strip(M)$ contains all transitions of atomic processes and hence each transition in M is a sequence of transitions in $Strip(M)$. On the other hand, if M contains the next construct, some of the intermediate states may be reachable in $Strip(M)$ but not M. Hence $\mathcal{R}(M)$ and $\mathcal{R}(Strip(M))$ are the two extremes of the state spaces one can hope to search. Since we are interested in checking invariants of the process, we would like the algorithm to visit all reachable states of M. Besides, the number of states visited by the algorithm shouldn't exceed those of $Strip(M)$. In figure 2, one can see that $\mathcal{R}(P' \| Q')$, *tbl* and $\mathcal{R}(Strip(P' \| Q'))$ are black states, shaded states and all states respectively.

The following theorem thus shows the correctness and the space requirements of our algorithm.

Theorem 1. *Let M be an arbitrary process expression. Then at the end of the procedure $Main(M)$,*

- $\mathcal{R}(M) \subseteq tbl$;
- $tbl \subseteq \mathcal{R}(Strip(M))$.

To prove theorem 1, we begin by defining an auxiliary transition relation parameterized by process expressions.

Definition 8. *Let M be a process expression, s, s' be states of M, and N be a subexpression of M. Define $s \longrightarrow_N s'$ as follows.*

$$\frac{s[X^M \setminus X^P] = s'[X^M \setminus X^P] \quad s[X_c^P] \xrightarrow{s[X_e^P]} s'[X_c^P] \in T^P}{s \longrightarrow_P s'}$$

$$\frac{s \longrightarrow_{M_1} s'}{s \longrightarrow_{M_1 \| M_2} s'} \qquad \frac{s \longrightarrow_{M_2} s'}{s \longrightarrow_{M_1 \| M_2} s'}$$

$$\frac{s \longrightarrow_M s'}{s \longrightarrow_{\mathsf{hide}\ H\ \mathsf{in}\ M} s'}$$

$$\frac{s \longrightarrow_N s' \quad s[X_c^N] \xrightarrow{s[X_e^N]} s'[X_c^N] \in \Theta}{s \longrightarrow_{\mathsf{next}\ \Theta\ \mathsf{for}\ N} s'}$$

$$\frac{s \longrightarrow_N s'' \quad s[X_c^N] \xrightarrow{s[X_e^N]} s''[X_c^N] \notin \Theta \quad s'' \longrightarrow_{\mathsf{next}\ \Theta\ \mathsf{for}\ N} s'}{s \longrightarrow_{\mathsf{next}\ \Theta\ \mathsf{for}\ N} s'}$$

■

Intuitively, $s \longrightarrow_N s'$ denotes the state changes from s to s' by taking a transition of the subexpression N.

Definition 9. *Let M be a process expression, $s, s' \in Q^M$, N be a subexpression of M, and $s \longrightarrow_N s'$. Define* $\mathrm{Expand}(s \longrightarrow_N s')$ *to be the set of paths*

$$\{s_0 = s \longrightarrow_{P_0} s_1 \cdots s_{n-1} \longrightarrow_{P_{n-1}} s_n = s' \mid$$
$$s_i \longrightarrow_{P_i} s_{i+1} \textit{ appears at the leaves of a derivation for } s \longrightarrow_N s'\}$$

■

Intuitively, $\mathrm{Expand}(s \longrightarrow_N s')$ contains paths of atomic steps from s to s'.

Lemma 1. *Let M be a process expression, N be a subexpression of M, t, t' be states of M, and tbl be the content of the table before $GetNext(N, t)$ is invoked.*

If $t \longrightarrow_N t'$, then

- $t' \in GetNext(N, t)$; *or*
- *For all paths $s_0 = t \longrightarrow_{P_0} s_1 \cdots s_{n-1} \longrightarrow_{P_{n-1}} s_n \in Expand(t \longrightarrow_N t')$, there is an i, $0 < i \leq n$ such that $s_i \in tbl$.*

The above lemma is used to establish that if $t \longrightarrow_N t'$ then t' is added to the hash-table eventually, and allows us to conclude that the algorithm visits all the reachable states of the input expression.

Since the hash table is used, each state in *tbl* is processed precisely once. The cost of processing a state is proportional to its outdegree, and this gives a bound on the running time of the algorithm. Thus, the saving in space comes at no extra overhead in time.

It is straightforward to modify the search algorithm to check invariants of process expressions. However, it should be noted that our algorithm cannot be used to check path properties. For instance, in the example of Figure 2, the global product $P' \| Q'$ has a transition from 1A to 4A, but this transition will not be discovered by the algorithm.

4 Applying Next as a heuristic

Suppose we wish to check an invariant of the process expression $M = P \| Q$. Can we find transition sets Θ and Σ, and search instead, the process expression $M' = \textsf{next}\ \Theta\ \textsf{for}\ P \,\|\, \textsf{next}\ \Sigma\ \textsf{for}\ Q$. As seen already, searching M' can be more efficient than searching M, and we can use the algorithm from previous section. In this section, we would like to find conditions under which searching M' suffices to solve the original problem.

Notice that application of next compresses a sequence of transitions into a single transition. Intuitively, as long as the compressed transitions are *local*, and do not access shared variables, such a transformation should be transparent to the remaining processes. The fact that sequences of invisible transitions can be compressed preserving weak equivalences is well known in theory and tools based on standard process algebras (see, for instance, [BRdSV87]). We adopt this concept to our framework in the sequel. In particular, we show that under appropriate restrictions, $\textsf{next}\ \Theta\ \textsf{for}\ P$ is equivalent to P under the definition of Milner's *weak-simulation equivalence*. This allows us, then, to apply next to subexpressions repeatedly in a recursive manner.

We start with the definition of observable transition. If a transition doesn't change the values of interface variables, it is called *unobservable*. From other processes' point of view, unobservable transitions are not interesting. This motivates the following definition of observable transition relation.

Definition 10. *Let $P = ((X_p, X_i, X_e), I, T)$ be a process, $q, q' \in Q_c$ and $e \in Q_e$. Then $q \xrightarrow{e}_w q'$ is an* observable transition *if there are $q_0 = q, q_1, \ldots, q_n = q' \in Q_c$ such that*

- *for all $0 \leq i < n$. $q_i[X_i] = q_0[X_i]$;*
- *for all $0 \leq i < n$. $q_i \xrightarrow{e} q_{i+1}$.* ■

Now we define the simulation equivalence.

Definition 11. *Let $P = ((X_p, X_i, X_e), I, T)$ and $P' = ((X'_p, X_i, X_e), I', T')$ be two processes, $q \in Q_c$, and $q' \in Q'_c$. A* (weak) simulation *relation $\preceq \subseteq Q_c \times Q'_c$ satisfies*

$$q \preceq q' \text{ implies } q[X_i] = q'[X_i] \text{ and}$$
$$\text{for all } r \in Q_c,\ \text{for all } e \in Q_e,\ \text{if } q \xrightarrow{e} r \text{ then there exists } r' \in Q'_c$$
$$\text{such that } q' \xrightarrow{e}_w r' \text{ and } r \preceq r'.$$

P is (weakly) simulated by P' *if there is a simulation* $\preceq$ *such that for all* $q \in I$, *there exists* $q' \in I'$ *with* $q \preceq q'$. $P \cong P'$ *if* $P \preceq P'$ *and* $P' \preceq P$. ■

Intuitively, if two processes are simulated by each other, one can perform the other's visible behaviors and vice versa. More specifically, if one is able to change its interface variables under certain environment, the other is able to do the same. Therefore, if an observable state is reachable for one process, it is reachable in the other too. Observe that if P and Q are simulation-equivalent, then to check invariant concerning observable states of P, it suffices to check that invariant of Q. Now we proceed to establish simulation-equivalence between a process and its next-abstraction provided the compressed transitions do not access shared variables.

Definition 12. *Let* $P = ((X_p, X_i, X_e), I, T)$ *be a process,* $q, q' \in Q_c$. *Then* $(q, e, q') \in T$ *is called a* write-visible transition *if* $q[X_i] \neq q'[X_i]$. $(q, e, q') \in T$ *is called a* read-visible transition *if there exists* $e' \in Q_e$, $(q, e', q') \notin T$.

A transition in T *is called* visible *if it is write-visible or read-visible. The set of visible transitions in* T *is denoted by* T_v. ■

The next technical lemma is needed when we prove the theorem.

Lemma 2. *Let* $P = (X, I, T)$ *be a process and* $T_v \subseteq \Theta \subseteq T$. *Define the relation* $S \subseteq Q_c \times Q_c \times Q_e$ *as follows.*

$$S = \{\ (q, q', e) \mid \exists q_0, \ldots, q_n . q_0 = q' \wedge q_n = q \wedge \forall 0 \leq i < n . (q_i, e, q_{i+1}) \in T \setminus \Theta\ \}.$$

Then for any $q \in \mathcal{R}(P)$, *there exists* $q' \in \mathcal{R}(\textsf{next}\ \Theta\ \textsf{for}\ P)$ *with* $(q, q', q[X_e]) \in S$.

Theorem 2. *Let* $P = (X, I, T)$ *be a process and* $T_v \subseteq \Theta \subseteq T$. *Then* $P \cong \textsf{next}\ \Theta\ \textsf{for}\ P$.

Given a process P, we now define its reduced version $\textsf{NEXT}\ P$ to be $\textsf{next}\ T_v\ \textsf{for}\ P$. By theorem 2, $P \cong \textsf{NEXT}\ P$. To be able to replace a process subexpression by application of $\textsf{NEXT}$ freely and repeatedly, we must establish that the simulation equivalence is a congruence for the operators in our language:

Theorem 3. *Let* P, P' *be processes and* $P \preceq P'$. *Then*

- $P \| Q \preceq P' \| Q$ *for any* Q;
- $\textsf{hide}\ H\ \textsf{in}\ P \preceq \textsf{hide}\ H\ \textsf{in}\ P'$;
- $\textsf{NEXT}\ P \preceq \textsf{NEXT}\ P'$.

Using the search algorithm shown in Section 3, one can check invariants φ on observable variables more efficiently as follows. Consider any process expression M, we construct a new process M' by substituting subexpressions E in M by $\textsf{NEXT}\ E$. This process can be applied repeatedly. Then every observable state of M will be visited by $Main(M')$. Thus, invariant verification of M reduces to the invariant verification of M'.

Consider a process expression $M \equiv M_1 \| \cdots \| M_n$ where each M_i is an atomic process P. We want to construct in a heuristic fashion an expression M' such

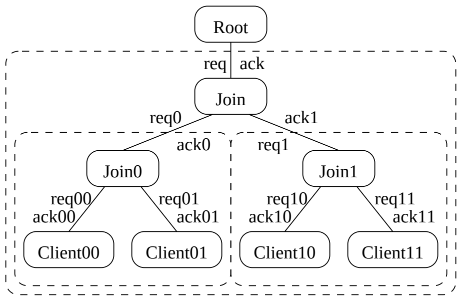

Fig. 3. Tree-structured Parity Computer

that M' involves applications of hide and NEXT to subexpressions of M with $Strip(M') = M$. By the previous sections, any substitution will yield the same answer but the effectiveness of reduction may differ by cases. How to find a suitable partition of M to apply this technique becomes an interesting problem on its own.

In principle, one would like to divide the system into parts so that the communication between any two parts is minimal. When a system is built with certain interconnection structure (like trees, rings, or grids etc), these structures should be exploited in verification. For instance, one can reduce the number of states by applying the next construct to subtrees in a tree-like structure. One can use the technique recursively on each subtrees and obtain even more efficient result. Note that hiding interface variables whenever possible is crucial since our reduction cannot ignore write-visible transitions.

5 Implementation and Experiments

We report two case-studies to test our theory. In our experiments, we use a syntax similar to MOCHA to specify modules [AHM+98]. The experiments were done using a prototype implementation in O'Caml. Details of the examples are available to the readers at `www.cis.upenn.edu/~bywang/concur99.html`.

The first example models a parity-computer with a binary tree structure. The system consists of several *Client* modules as leaves and *Join* modules as nodes. The interconnection between these modules and *Root* module is shown in Figure 3 for 4-client configuration.

Instead of searching

$$\text{Root} \parallel \text{Join} \parallel \text{Join0} \parallel \text{Join1} \parallel \text{Client00} \parallel \text{Client01} \parallel \text{Client10} \parallel \text{Client11}$$

we can naturally cluster modules as shown in Figure 3. The expression we search is

$$\text{Root} \parallel \text{NEXT hide } [\ \text{Join} \parallel \begin{matrix}\text{NEXT hide (Join0} \parallel \text{Client00} \parallel \text{Client01)} \\ \text{NEXT hide (Join1} \parallel \text{Client10} \parallel \text{Client11)}\end{matrix}].$$

Num. of clients	Without next		With next	
	States	Time	States	Time
3	202	0.37s	189	0.36s
4	1748	1.05s	515	0.39s
5	6994	4.46s	2017	0.77s
6	69614	55.22s	2633	0.94s
7	320094	324.73s	3531	1.32s
8	2782634	4449.12s	4751	1.79s

Fig. 4. Reduction for the parity tree computer

For readability the argument variables to hide are not shown above, but should be obvious from context: only the variables communicating with the parent need to be observable at each level of the subtree. Notice that for the top-level expression, a step is either a step of the module *Root*, or it is a sequence of steps of the tree rooted at *Join* until either *req* is updated or *ack* is read. A step of the tree rooted at *Join* is either a step of the module *Join*, or a sequence of steps of the subtree rooted at *Join0* until *req0* or *ack0* is accessed, or a sequence of steps of the subtree rooted at *Join1* until *req1* or *ack1* is accessed. Since each *Join* node communicates with parent only after it has received and processed requests from its children, we get excellent reduction. In the table of Figure 4, we show the number of states visited, together with the CPU time required by the algorithm for different number of clients. Observe that while the number of states grows exponentially without any reduction, our heuristic scales almost linearly.

Our second example concerns the standard leader election protocol in an asynchronous ring of cells. The protocol used here is taken from [Lyn96], and has been used as a benchmark to study how model checking algorithms scale. Figure 5 shows four cells connected together and the partition used for applications of next. The results are reported in the table of Figure 6. While not as spectacular as the parity-tree-computer, observe the significant reduction in space and time.

6 Conclusions

We have presented a new heuristic for model checking based hierarchically structured transition relations. The heuristic potentially combines advantages of the on-the-fly search with early detection of violations and compositional reduction techniques based on process equivalence.

While the construct next has been studied previously in the context of symbolic search [AHR98], this is the first study that establishes conditions under which next construct can be introduced without changing the answer to the original verification problem. The transformation of Section 3 can be used for improving efficiency of symbolic model checking as well. The reduction offered by our method differs with examples. The reduction is maximum when processes

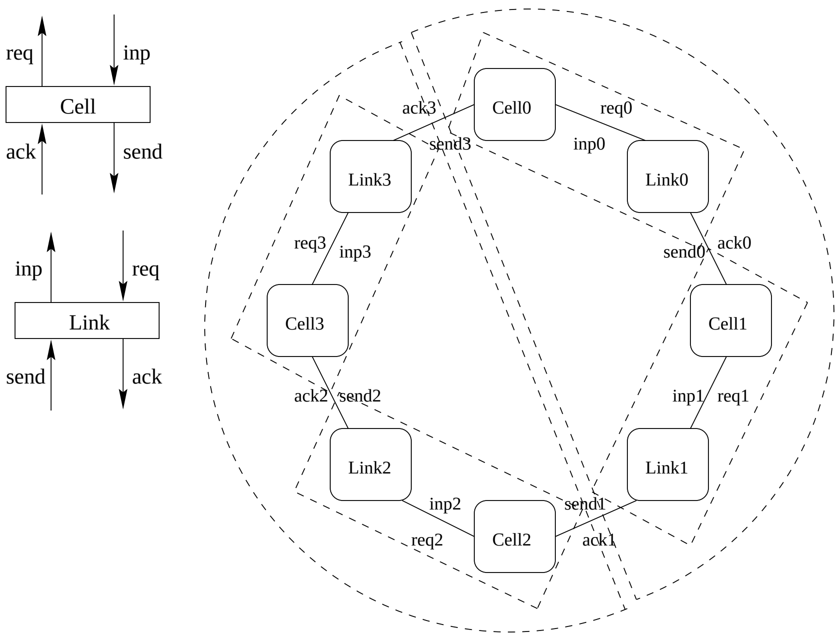

Fig. 5. Leader Module

Num. of clients	Without next		With next	
	States	Time	States	Time
3	10847	7.64s	4401	1.71s
4	195017	210.15s	26430	10.79s
5	2235421	4351.20s	230569	115.21s

Fig. 6. Reduction for the leader election in ring

can be clustered so that when one cluster communicates with another, it is only in a small number of different internal states.

Partial-order reduction has been successfully combined with on-the-fly search (see, for example, [Pel94,Val92,God90]). It may appear that our reduction is a simple form of partial-order reduction, but on more scrutiny, the two techniques are incomparable, and we believe them to be compatible. Partial-order reduction does not exploit the hierarchical structure, but uses independence among transitions to restrict the number of interleavings. It is easy to construct examples in which our method works better than the partial-order reduction method, and vice versa. On the examples reported in this paper, the software Spin gives much more impressive reductions on the leader election protocol, but our method performs better for the parity computer. The modeling languages of Spin and Mocha are quite different to make any meaningful comparison in the absolute number of states for each configuration, so the comparison is based on the rates of growth with the scaling parameter. More information about the comparison can be found along with the code of our examples (`www.cis.upenn.edu/~bywang/concur99.html`).

Another popular heuristic for improving on-the-fly search is reductions using symmetry [ES93,CFJ93,ID96]. Again, our method is incomparable to and compatible with symmetry reduction. In particular, note that in our examples, if we change the individual clients so that they behave slightly differently from each other, or the different join nodes compute different functions, there will be no significant impact on the reduction obtained. In the parity computer example, if we change $Join1$ to its complement, the experiment shows that our heuristic can still achieve the same reduction for the eight clients system.

The process-algebraic tools such as *Concurrency workbench* [CPS93] and CADP [FGK+96] implement the reductions based on weak-simulation (or even the less expensive reduction to compress internal transitions). We have already discussed our motivation to explore on-the-fly algorithms that implement the reduction in a top-down manner.

In this paper, we have used next only to compress invisible transitions. However, our search algorithm can be used effectively in other contexts. First, next can be supplied an argument that contains only a subset of the visible transitions. This would perform a more efficient search at a coarser granularity of atomicity. Note that, while this is similar in spirit to the *atomic* construct in the language of SPIN to cluster a sequence of atomic transitions, it gives a high-level and hierarchical construct. Second, next can be used to implement the so-called "run-to-completion" semantics in which transitions are executed until there are no more enabled transitions.

References

[AH96] R. Alur and T.A. Henzinger. Reactive modules. In *Proceedings of the 11th IEEE Symposium on Logic in Computer Science*, pages 207–218, 1996.

[AHM+98] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proceedings of the 10th International Conference on Computer Aided Verification*, LNCS 1427, pages 516–520. Springer-Verlag, 1998.

[AHR98] R. Alur, T. Henzinger, and S. Rajamani. Symbolic exploration of transition hierarchies. In *Proceedings of the Fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1384, pages 330–344. Springer-Verlag, 1998.

[BFH90] A. Bouajjani, J.C. Fernandez, and N. Halbwachs. Minimal model generation. In *Computer-Aided Verification, 2nd International Conference, CAV'90*, LNCS 531, pages 197–203. Springer-Verlag, 1990.

[BG88] G. Berry and G. Gonthier. The synchronous programming language ESTEREL: design, semantics, implementation. Technical Report 842, INRIA, 1988.

[BRdSV87] G. Boudol, V. Roy, R. de Simone, and D. Vergamini. Process calculi, from theory to practice: verification tools. In *Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 1–10. 1987.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.

[CFJ93] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal-logic model checking. In *Computer-Aided Verification, Fifth International Conference, CAV'95*, LNCS 697, pages 450–462. Springer-Verlag, 1993.

[CK96] E.M. Clarke and R.P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of finite-state systems. *ACM Trans. on Programming Languages and Systems*, 15(1):36–72, 1993.

[DDHY92] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[ES93] E.A. Emerson and A.P. Sistla. Symmetry and model checking. In *Computer-Aided Verification, Fifth International Conference, CAV'95*, LNCS 697, pages 463–478. Springer-Verlag, 1993.

[FGK+96] J. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In *Proceedings of the Eighth International Conference on Computer-Aided Verification*, LNCS 1102. Springer-Verlag, 1996.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. In E.M. Clarke and R.P. Kurshan, editors, *Computer-Aided Verification, 2nd International Conference, CAV'90*, LNCS 531, pages 176–185. Springer-Verlag, 1990.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[Hol97] G.J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[ID96] C.N. Ip and D.L. Dill. Verifying systems with replicated components in murφ. In *Proceedings of the Eighth International Conference on Computer Aided Verification*, LNCS 1102. Springer-Verlag, 1996.

[KM97] J.-P. Krimm and L. Mounier. Compositional state space generation of lotos programs. 1997.

[KS90] P. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, 1990.

[Lyn96] N.A. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. LNCS 92. Springer-Verlag, 1980.

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Computer Aided Verification, Proc. 6th Int. Conference*, LNCS 818. Springer-Verlag, 1994.

[PT87] R. Paige and R.E. Tarjan. Three partition-refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent programs in CESAR. In *Proceedings of the Fifth International Symposium on Programming*, LNCS 137, pages 195–220. Springer-Verlag, 1982.

[Val92] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1:297–322, 1992.