On-the-fly Reachability and Cycle Detection for Recursive State Machines *

Rajeev Alur[1], Swarat Chaudhuri[1], Kousha Etessami[2], and P. Madhusudan[3]

[1] University of Pennsylvania, USA
[2] University of Edinburgh, UK
[3] University of Illinois at Urbana-Champaign, USA

Abstract. Searching the state space of a system using enumerative and on-the-fly depth-first traversal is an established technique for model checking finite-state systems. In this paper, we propose algorithms for on-the-fly exploration of recursive state machines, or equivalently pushdown systems, which are suited for modeling the behavior of procedural programs. We present algorithms for reachability (is a bad state reachable?) as well as for fair cycle detection (is there a reachable cycle with progress?). We also report on an implementation of these algorithms to check safety and liveness properties of recursive boolean programs, and its performance on existing benchmarks.

1 Introduction

Recursive state machines (RSM) can model control flow in typical sequential imperative programming languages with recursive procedure calls, and are equivalent to pushdown systems [1]. Even though the state-space of an RSM is infinite due to recursion, model checking problems for RSMs are decidable [6, 7, 15, 12, 1, 5]. *Extended RSMs* (ERSM) augment RSMs with global and local variables that can be tested and updated along the edges of the control structure. Contemporary tools for software verification employ abstraction to automatically extract ERSMs from code written in languages such as C, and then use ERSM model checking algorithms to check temporal requirements [4, 17]. The complexity of the key analysis problems for ERSMs, such as reachability, is polynomial in the number of states [12, 1], where a state needs to encode the control location and the values of all the global and in-scope local variables. To cope with the state-space explosion due to the variables, existing implementations of ERSM model checkers such as Bebop [3] and Moped [12] use symbolic encoding using automata and binary decision diagrams. In this paper, we propose *on-the-fly* explicit-state search algorithms as a viable alternative.

An on-the-fly algorithm explores the reachable states starting from initial states by computing the successors of a state only when needed, typically using depth-first traversal, and terminates as soon as it finds a counterexample to the

* This research was partially supported by ARO URI award DAAD19-01-1-0473, and NSF awards ITR/SY 0121431 and CCR-0306382.

property being verified. While the effectiveness of this technique is limited by the number of states that can be stored and processed, it has its own advantages over the symbolic approach. The guards and updates on an edge can be complex, and can even include calls to library functions. It does not require an *a priori* encoding of the states, and hence, can support complex and unbounded data types and dynamic creation of data. Early termination allows discovery of shallow bugs rapidly. Finally, the performance is more predictable as more states are guaranteed to be searched with an increase in the available memory and time. Consequently, tools such as SPIN [18] and MURφ [10] that rely on on-the-fly explicit-state search algorithms have been very effective for classical model checking problems. More recent tools like ZING [2] and BANDERA [8] are also explicit-state, support complex data types, concurrency, and recursion, but do not offer any termination guarantees.

We first consider the *reachability* problem for ERSMs: starting from an initial state, can control reach one of the target locations along some execution of the ERSM? Our algorithm combines on-the-fly traversal of extended state machines with early termination used in explicit-state model checkers and a summarization algorithm used in interprocedural data-flow analysis [20].

We build on our reachability algorithm to arrive at a novel solution to the *fair cycle detection* problem for ERSMs: starting from an initial state, is there an execution of the ERSM that visits one of the target locations infinitely often? This fair cycle detection problem is central to the algorithmic verification of liveness requirements. The known solution to this problem is most naturally viewed in two phases [1]. In the first phase, all the summary edges are computed, and the second phase reduces to fair cycle detection in an ordinary graph containing these summary edges. Since we desire an on-the-fly solution with the possibility of early termination, we do not want to compute all the summary edges first, and wish to interleave the two phases. We can view this problem as fair cycle detection in a graph (second phase) in which the edges, namely, the summary edges discovered by the first phase, are inserted dynamically. For on-the-fly fair cycle detection in ordinary graphs, tools such as SPIN employ the so-called *nested depth-first-search* algorithm [9], but this algorithm relies on the ordering of states in a depth-first traversal, which fails if we allow dynamic insertion of (summary) edges. In the proposed solution, we use a path-based algorithm for computing the strongly-connected-components (SCC) of a graph [16]. Every time the first phase discovers a summary transition, the SCC discovery algorithm processes the newly reachable states. As a new SCC is discovered, early termination is possible if it contains a state with the target location or a summary transition representing a path through such a state, and if not, all vertices in the SCC can be collapsed to a single vertex for efficiency. Cycle detection (but not *fair* cycle detection) is interesting in program analysis in the context of *points-to* analysis and cycle detection in dynamic graphs has been studied [19, 14].

For analysis of worst-case time bounds, let us assume that the ERSM has k components, has no variables and has total size n (control locations plus transitions). Then, the time bounds for non-on-the-fly explicit-state algorithms for reachability and fair cycle detection are $O(n)$ [1], while the symbolic algorithms for reachability and fair cycle detection are $O(n^2)$ [13]. The newly proposed reachability algorithm is $O(n)$ and the new fair cycle detection algorithm is $O(kn)$.

To test the performance of the proposed algorithms, we implemented them in the tool VERA. The ERSM model is described in an input language that extends the boolean programs of BEBOP [3] with additional types such as bounded integers. The specifications can be written as monitors, and the tool performs on-the-fly reachability and fair cycle detection on the product of the model and the monitor. The regression test suite of SLAM contains boolean programs obtained from abstractions of real-world C code [4], and while VERA performs well on examples that contain a bug, it performs poorly compared to symbolic checkers such as MOPED [12] when forced to search the entire space. On examples such as Quicksort from MOPED's benchmarks that need manipulation of integer variables, VERA performs significantly better than MOPED. Finally, we manually abstracted a Linux driver code in which METAL had found a double locking error using static analysis [11]. VERA performs well on this example, and can also prove the liveness requirement that "every lock should eventually be released."

2 Extended recursive state machines

In this section, we introduce the formalism of extended recursive state machines (ERSMs). We start with the language we use to specify guarded commands.

Expressions and assignments Let us have a set T of types and a domain D_t associated with each type $t \in T$. In particular, we allow a *boolean* type with the domain $\{T, F\}$. Let V be a finite set of variables where each variable is associated with a type, and let $Expr(V)$ be a set of typed expressions. We refer to the set of expressions of boolean type as $BoolExp(V)$.

An *interpretation* of V is a map $\sigma : v \in V \mapsto d \in D_t$, where v is of type t. Every interpretation can be extended to a unique semantic map $\sigma : expr \in Expr \mapsto d \in D_t$, where $expr$ is of type t.

An *assignment* over V has the form $[x_1, x_2, \ldots, x_l] := [exp_1, exp_2, \ldots, exp_l]$, where $x_j \in V$ are distinct variables, and for all j, $exp_j \in Expr(V)$ is an expression of the same type as x_j. We refer to the set of assignments over V as $Assgn(V)$. The semantics of assignments are defined over pairs (σ_1, σ_2) of interpretations of V. Given an assignment α of the above form, we say $\sigma_2 = \alpha(\sigma_1)$ if (1) $\sigma_2(x_j) = \sigma_1(exp_j)$ for all x_j, and (2) $\sigma_1(y) = \sigma_2(y)$ for all variables $y \in V \setminus \{x_1, x_2, \ldots, x_l\}$.

Syntax of ERSMs An *extended recursive state machine (ERSM)* A is a tuple $\langle G, \gamma_{in}, p, (A_1, A_2, \ldots, A_k)\rangle$, where G is a finite set of global variables, γ_{in} is an *initial* interpretation of G, $p \in \{1, \ldots, k\}$ is the index of the initial component, and each *component state machine* $A_i = \langle L_i, I_i, O_i, \lambda_{i_{in}}, N_i, en_i, ex_i, \delta_i \rangle$ consists of

- a finite set L_i of local variables, a set $I_i \subseteq L_i$ of input variables, and a set $O_i \subseteq L_i$ of output variables. The sets I_i and O_i are totally ordered, the j-th variables in these orders being given by $I_i(j)$ and $O_i(j)$ respectively. Also, we require that $I_p = \emptyset$;
- an *initial* interpretation $\lambda_{i_{in}}$ of L_i;
- a finite set N_i of nodes;
- two special nodes $en_i, ex_i \in N_i$, known respectively as the *entry node* and the *exit node*; [1]
- A set δ_i of *edges*, where an edge can be one of two forms:
  - Internal edge: A tuple (u, v, g, α). Here u and v are nodes in N_i, $g \in \mathit{BoolExp}(G \cup L_i)$ is a *guard* on the edge, and $\alpha \in \mathit{Assgn}(G \cup L_i)$ is an assignment. Intuitively, such an edge will be taken only if the guard g is true, and if it is taken, the assignments will be applied to the current variables. The set of internal edges in component i is denoted by δ_i^I.
  - Call edge: A tuple $(u, v, g, m, \mathit{in}, \mathit{out})$. Here u and v are nodes in N_i, $g \in \mathit{BoolExp}(G \cup L_i)$ is an edge guard, $m \in \{1, 2, \ldots, k\}$ is the index of the *called* component, and $\mathit{in} \in L_i^r$ and $\mathit{out} \in L_i^q$, for $r = |I_m|$ and $q = |O_m|$, are two lists of local variables. Intuitively, *in* is the list of parameters passed to the call, and *out* is the list of variables where the outputs of the call are stored on return from the call. We require that all variables in *out* are distinct.
    The set of call edges in component i is denoted by δ_i^C. The function $Y_i : \delta_i^C \to \{1, 2, \ldots, k\}$ maps call edges to indices of the components they call, so that, for a call edge e such as above, $Y_i(e) = m$.

  We assume that entry nodes en_i do not have incoming edges and exit nodes ex_i do not have outgoing edges. □

We designate the component A_p as the *initial component*. This component, where runs of A begin, models the "main" procedure in procedural programs.

Example: Figure 1 shows a sample ERSM with one global variable a, and components A_1 and A_2. Component A_1 has an input variable i, and an output variable x. Component A_2, also the initial component, has no inputs and one local/output variable y. All variables are of boolean type, and initially, we have $a = F$, $x = T$, and $y = T$.

In the diagram, an internal edge (u, v, g, α) is drawn as a solid arrow from node u to node v annotated by $(g \Rightarrow \alpha)$ (we will omit the guard g and the assignment α if, respectively, g is always true and the assignment α is empty). A call edge $(u, v, g, m, \mathit{in}, \mathit{out})$ is a dashed arrow annotated by $(g \Rightarrow \mathit{out} := m(\mathit{in}))$ (we omit *out* if it is empty, and leave out the guard g if it is trivially true).

[1] The usual definition of RSMs [1] allows components to have multiple entry and exit nodes. In this paper, we model entries and exits by input and output variables, so it suffices to let each component have one entry and one exit node.

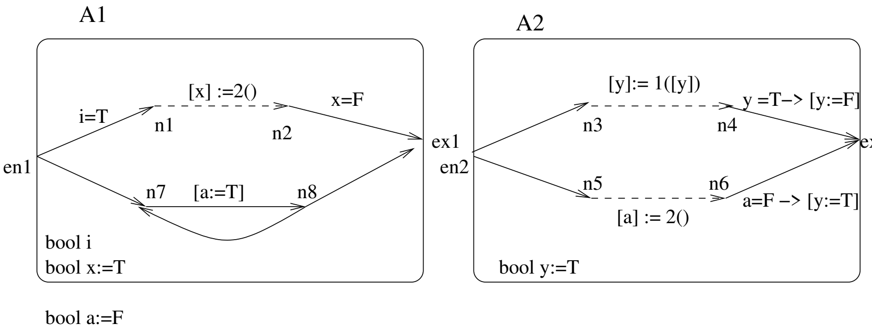

Fig. 1. A sample ERSM

Semantics of ERSMs ERSMs model procedural programs written in C-like imperative languages and resemble the latter in operational semantics. Components, nodes, internal edges, and call edges in ERSMs respectively model procedures, control locations, intraprocedural control flow, and procedure invocations in procedural programs, and *configurations* and *runs* of ERSMs are the equivalents of program states and program executions. A *configuration* of an ERSM consists of a call stack, a current node, and a current interpretation of the global and in-scope local variables. The transition relation on configurations has three kinds of transitions: *internal steps*, *calls*, and *returns*. At an internal step, control follows an internal edge, reaches a new node, and applies the assignments on the edge to the variables in scope, the stack remaining unaffected. During a call, a call edge and the current interpretation of the local variables in scope are pushed onto the call stack, control reaches the entry node of the called component, and a new set of local variables are initialized. At a return, we pop a calling context off the stack, reinstate the popped local variables (after adjusting for possible output values), and proceed to the node to which the popped call edge leads.

We now formally define the configuration space Q of an ERSM A. A *configuration* of A is a tuple $\psi = \langle \gamma, \mathit{stack}, u, \lambda \rangle$, where γ is an interpretation of G, and *stack* is either of the form $\langle (e_1, \lambda_1), (e_2, \lambda_2), \ldots, (e_r, \lambda_r) \rangle$ or the empty list. Here, $e_1, e_2, \ldots, e_r$ are call edges of A, λ_1 is an interpretation of L_p, e_1 is a call edge in A_p, and, for every $i > 1$, λ_i is an interpretation of L_c and e_i is a call edge in A_c, where $c = Y(e_{i-1})$. Finally, u is a node in N_j and λ is an interpretation of L_j, where j equals p if *stack* is empty, and $Y(e_r)$ otherwise.

In a configuration of the above form, we define the node u to be the *current node* in ψ. We refer to this node as $\mathit{Currnode}(\psi)$.

We need some more notation before we can define the transition relation on these configurations. Let σ_1 and σ_2 be interpretations of disjoint sets of variables V_1 and V_2. Then $\sigma_1 \sqcup \sigma_2$ is the interpretation of $(V_1 \cup V_2)$ that agrees with σ_1 and σ_2 on variables from V_1 and V_2 respectively.

Now let $l_1 \in U_1^q$ and $l_2 \in U_2^q$ be two lists of variables such that $U_1 \cap U_2 = \emptyset$ and members of l_2 are all distinct. Let us also have interpretations σ_1 and σ_2 of U_1 and U_2 respectively. Then $\tau = \mathit{borrowValues}(\sigma_2, l_2, \sigma_1, l_1)$ is an interpretation of U_2 obtained by (1) setting $\tau(l_2(i)) = \sigma_1(l_1(i))$ for all i, and (2) for all other

v, setting $\tau(v) = \sigma_2(v)$. Intuitively, *borrowValues* replaces the values of those variables in σ_2 that occur in l_2, the i-th variable in l_2 getting the value that interpretation σ_1 gives to the i-th variable in l_1.

The global transition relation Δ of an ERSM is then defined as follows. Let $\psi = \langle \gamma, \mathit{stack}, u, \lambda \rangle$ be a configuration with $u \in N_j$. Then $(\psi, \psi') \in \Delta$ iff one of the following sets of conditions holds:

1. Internal step
   (a) $(u, u', g, \alpha) \in \delta_j^I$ for a node u' of A_j,
   (b) $\gamma \sqcup \lambda$ satisfies g, and
   (c) $\psi' = \langle \gamma', \mathit{stack}, u', \lambda' \rangle$, where $\gamma' \sqcup \lambda' = \alpha(\gamma \sqcup \lambda)$.
2. Call
   (a) $e = (u, u', g, m, \mathit{in}_m, \mathit{out}_m) \in \delta_j^C$ for a node u' of A_j,
   (b) $\gamma \sqcup \lambda$ satisfies g, and
   (c) $\psi' = \langle \gamma, \langle \mathit{stack}, (e, \lambda) \rangle, \mathit{en}_m, \lambda' \rangle \rangle$, where $\lambda' = \mathit{borrowValues}(\lambda_{m_{in}}, I_m, \lambda, \mathit{in}_m)$.
3. Return
   (a) u is the exit node ex_j of A_j,
   (b) stack is of the form $\langle \mathit{stack}', (e_r, \lambda_r) \rangle$
   (c) $e_r = (v, u', g, j, \mathit{in}_j, \mathit{out}_j)$ for some v, u', g, in_j, and out_j, and
   (d) $\psi' = \langle \gamma, \mathit{stack}', u', \lambda' \rangle \rangle$, where $\lambda' = \mathit{borrowValues}(\lambda_r, \mathit{out}_j, \lambda, O_j)$.

Note that the configurations Q and the transition relation Δ define an ordinary (and in general infinite) transition system T_A. A *run* of A is a (finite or infinite) sequence $\rho = \psi_0\psi_1\psi_2\ldots$, where $\psi_0 = \langle \gamma_{in}, \perp, \mathit{en}_p, \lambda_{p_{in}} \rangle$ ($\perp$ being the empty stack sequence), and for all i, $\psi_i \in Q$ and $(\psi_i, \psi_{i+1}) \in \Delta$. The semantics of A are defined by its set of runs.

Given an ERSM A, we are interested in two central algorithmic questions:

1. *Reachability*: Given an ERSM A and a set of *target nodes* T of A, does A have a run $\rho = \psi_0\psi_1\psi_2\ldots$ such that $\mathit{Currnode}(\psi_j) \in T$ for some j?
2. *Fair cycle detection*: Given an ERSM A and a set R of *repeating nodes* of A, does A have a run $\rho = \psi_0\psi_1\psi_2\ldots$ such that $\mathit{Currnode}(\psi_j) \in R$ for *infinitely many* $j \in \mathbb{N}$?

In this paper, we present two algorithms that search the set of local states enumeratively to solve these problems. These algorithms differ from previous work in two important respects:

1. *On-the-fly search:* We generate states "on demand", as we explore the state space, and only store the visited states.
2. *Early termination:* Our algorithms terminate as soon as a reachability witness or a cycle containing a repeating state occurs in the visited state space. Consequently, our algorithms do not necessarily have to generate the entire state space in order to terminate.

3 Reachability

We now describe an on-the-fly, early-terminating algorithm to check if a given set T of target nodes is reachable in an ERSM A.

A *state* of an ERSM A is a tuple of the form $s = \langle v, \gamma, \lambda \rangle$, where v is a node, and γ and λ are interpretations of global and local variables. Note that a state is different from a configuration in that it does not include the stack. An *entry state* for component A_i is a state $s = \langle v, \gamma, \lambda \rangle$ where $v = en_i$. Likewise, $s = \langle v, \gamma, \lambda \rangle$ is an *exit state* if $v = ex_i$. A *summary* is a pair (s_{en}, s_{ex}), where s_{en} is an entry state and s_{ex} is an exit state in the same component.

Let us now define a state graph $\mathcal{S}$ corresponding to A. The vertices of $\mathcal{S}$ are the states of A and the set of transitions of $\mathcal{S}$ is the smallest set E of transitions satisfying the following conditions:

- **Internal transitions** Let $s = \langle u, \gamma, \lambda \rangle$ be a state. If A_i has an internal edge (u, v, g, α) and the interpretation $\gamma \sqcup \lambda$ satisfies g, then E has a internal transition (s, s'), where $s' = \langle v, \gamma', \lambda' \rangle$, where $\gamma' \sqcup \lambda' = \alpha(\gamma \sqcup \lambda)$.
- **Call and summary transitions** Let $s = \langle u, \gamma, \lambda \rangle$ be a state. Assume A_i has a call edge $(u, v, g, m, in_m, out_m)$ and the interpretation $\gamma \sqcup \lambda$ satisfies g. Let $s_{en} = \langle en_m, \gamma, \lambda_{en} \rangle$, where $\lambda_{en} = \mathit{borrowValues}(\lambda_{m_{in}}, I_m, \lambda, in_m)$. Then (s, s_{en}) is a call transition in E.
  If $s_{ex} = \langle ex_m, \gamma', \lambda_{ex} \rangle$ is some exit state in A_m and s_{ex} is reachable from s_{en} using only internal and summary transitions, then (s, s') is a *summary transition* in E where $s' = \langle v, \gamma', \lambda' \rangle$ and $\lambda' = \mathit{borrowValues}(\lambda, out_m, \lambda_{ex}, O_m)$.

For a set of repeating nodes R, let us also define $\mathcal{S}_R$, which is defined exactly as S is defined above except that summary transitions can be of two kinds, *fair* or *not fair*. When a summary transition is added, it is set to be fair if the run from s_{en} to s_{ex} goes through a state involving R or uses a fair summary transition.

The key to checking reachability and cycles in an ERSM is given by the following lemma:

Lemma 1 ([1]). *Let A be an ERSM and let $\mathcal{S}$ be its associated state-graph. For a given set of target nodes T, T is reachable in A iff there is a node of the form (u, γ, λ) with $u \in T$ reachable in $\mathcal{S}$. Similarly, given a set of repeating nodes R, there is run of A that visits R infinitely often iff there is a path in $\mathcal{S}_R$ that visits the set $\{(u, \gamma, \lambda) | u \in R\}$ infinitely often or uses fair summary edges infinitely often.*

Note that if local and global variables are finite-domain, then $\mathcal{S}$ is finite as well, and the above lemma shows checking reachability and fair cycles are decidable. We refer to the subgraph of $\mathcal{S}$ induced by the nodes belonging to A_i, i.e. nodes of the form (u, γ, λ) where $u \in N_i$, as S_i; the graphs S_i contain only internal and summary transitions. Our reachability algorithm explores $\mathcal{S}$ on-the-fly looking for a state of the form (u, γ, λ), where $u \in T$. It can be in

```
Reachability(s, s_en)
Visited ← Visited ∪ {(s, s_en)}
if Currnode(s) ∈ T
  then print ("Target reached") ;  break
if s is an exit state in component i
  then VisitedExits[i, s_en] ← VisitedExits[i, s_en] ∪ {s}
       for (s', s'_en) ∈ VisitedCalls[i, s_en]
       do s_ret = GetReturnState(s', s)
          if (s_ret, s'_en) ∉ Visited
            then Reachability(s_ret, s'_en)
  else for e ∈ Edges_I(s)
       do if s satisfies guard of e
            then s' ← Apply(e, s)
                 if (s', s_en) ∉ Visited
                   then Reachability(s', s_en)
       for e ∈ Edges_C(s)
       do if s satisfies guard of e
            then m ← Y(e); s' ← GetEntryState(e, s)
                 VisitedCalls[m, s'] ← VisitedCalls[m, s'] ∪ {(s, s_en)}
                 if (s', s_en) ∉ Visited
                   then Reachability(s', s_en)
                   else for s_ex ∈ VisitedExits[Y(e), s']
                        do s_ret = GetReturnState(s, s_ex)
                           if (s_ret, s_en)) ∉ Visited
                             then Reachability(s_ret, s_en)
```

Fig. 2. On-the-fly reachability in ERSMs

fact viewed as an interleaving of k separate depth-first searches, the i-th search taking place in the transition system S_i.

Our search begins from the initial state $\langle en_p, \gamma_{in}, \lambda_{p_{in}} \rangle$, in the initial component A_p. The search proceeds depth-first following edges in A_p. If, during this search, we are at a state s_1 in S_p and find a call edge calling component A_q, we would need to search along a summary transition in S_p. To discover this transition, however, we would need to know the reachability relation between the corresponding entry and exit states in A_q, and to compute this relation, we must search S_q.

The crux of the algorithm is to view S_p as an *incompletely specified* transition system and *suspend* the search in S_p when such a situation occurs. Given s_1 and the call edge in question, we can compute the entry state s_2 in S_q reached following the corresponding call transition. If s_2 has not been visited so far, we search S_q starting from s_2; if a search from s_2 has previously been started, we simply suspend searching and wait for future "updates". As we learn more about reachability between entry and exit states in S_q, we may add corresponding summary transitions in S_p and resume the search in S_p. If all local searches terminate, then we have explored all of the reachable part of $\mathcal{S}$, and can terminate.

Figure 2 describes the algorithm more formally. Given a state $s = \langle v, \gamma, \lambda \rangle$, we refer to the set of internal and call edges going out of v as $Edges_I(s)$ and

$Edges_C(s)$ respectively. If the variables in s satisfy the guard g on an internal edge e, the function $Apply(e, s)$ returns the state s' to which the corresponding internal transition leads. If s satisfies the guard g on a call edge e, the function $GetEntryState(e, s)$ returns the entry state s' that is the target of the corresponding call transition. Finally, suppose s is an exit state in component A_m and s' is a state with a call transition to component A_m; also suppose there exists a summary transition (s', s'') corresponding to s', e and s. Then the function $GetReturnState(s', s)$ returns the state s''.

The function *Reachability* has two inputs: a state s in component A_i and an entry state s_{en}. The pair of states (s, s_{en}) forms a *context* if s is reachable from s_{en}. The set $VisitedCalls[i, s_{en}]$ stores the set of "calling contexts": contexts (s', s'_{en}) where control switched to component A_i and entry state s_{en}. Then, if an exit state s_{ex} in A_i is reachable from state s_{en}, a summary transition between states s' and $s'' = GetReturnState(s', s_{ex})$ has been discovered.

To solve the reachability problem, we call $Reachability(s_{init}, s_{init})$, where $s_{init} = \langle en_p, \gamma_{in}, \lambda_{p_{in}} \rangle$. Termination of this algorithm is guaranteed if the set of states reachable from the initial states is finite; one such special case is when all types are finite-domain. We omit the detailed proof of correctness.

Theorem 1. *Let A be an ERSM, T be a set of target nodes, and $s_{init} = \langle en_p, \gamma_{in}, \lambda_{p_{in}} \rangle$. Then if the algorithm $Reachability(s_{init}, s_{init})$ halts, it prints "Target reached" iff there is a run of A that reaches a configuration ψ with $Currnode(\psi) \in T$. Moreover, if the set of states reachable from s_{en} in $\mathcal{S}$ is finite, then $Reachability(s_{init}, s_{init})$ is guaranteed to halt.* □

This algorithm has some of the nicer, "on-the-fly" properties of DFS. We start with an initial state, only store the "visited" state space, make a switch to a different component only when a call edge requires it, and, even when such a switch is made, "discover" entry states only when necessary. Moreover, we can terminate as soon as we encounter a target state.

If s'_{en} is an entry state of S_i reachable in $\mathcal{S}$ and s' is reachable from s'_{en} using edges in S_i only, then when calling $Reachability(s_{init}, s_{init})$, the recursive procedure $Reachability(s', s'_{en})$ will be called at most once. This observation leads to the following complexity of the reachability algorithm in terms of the number of discovered states and transitions in $\mathcal{S}$:

Theorem 2. *Let Reachability terminate on a given ERSM A. Let n and m be the number of states and edges in the explored part of $\mathcal{S}$. Let β be a bound on the maximum number of reachable entry or exit states in any component S_i. Then $Reachability(s_{init}, s_{init})$ takes $O(m\beta + n\beta^2)$ time to terminate and space $O(n\beta)$.* □

4 Fair Cycle Detection

Let us fix an ERSM A and a set of repeating nodes R of A. Let $\mathcal{S}_R$ be the associated state-graph of A and R as defined in the previous section. In this section, we present an on-the-fly fair cycle detection algorithm for ERSMs that searches the transition system $\mathcal{S}_R$ for a cycle containing a repeating state or a fair summary edge. If the domains of the types are finite, Lemma 1 guarantees that A has a run visiting infinitely many repeating states if and only if such a cycle exists. Our core idea is to view $\mathcal{S}_R$ as an incomplete transition system to which edges are added dynamically, and to use an *online cycle detection algorithm for dynamically presented graphs* to find such a cycle.

The following are a few ways in which this can be implemented:

- The most naive algorithm would be to search the state-space of A using Reachability, postponing cycle detection until we know all states and transitions in $\mathcal{S}_R$. At that point, we may detect cycles in $\mathcal{S}_R$ using an algorithm (such as nested DFS [9]) for cycle detection in finite graphs. This algorithm, however, is inherently a two-phase algorithm and does not have the early termination property.
- Another possibility is to adapt the nested DFS algorithm [9] to a setting where summary transitions are dynamically presented and early termination is required. This turns out to be difficult. The problem is that in the nested DFS algorithm, the secondary search follows the DFS order computed by the primary search: if s and s' are two states such that s is an ancestor of s' in the primary DFS tree, the secondary search from s' must terminate *before* the secondary search from s may start. However, in our context, we may discover a summary transition from s' (that can possibly introduce cycles) while searching a different branch of the primary DFS tree rooted at s.
  A conceivable way of adapting this algorithm to our setting would be to start a new instance of Reachability each time we reach a repeating state s_r, to check if s_r is reachable from itself. However, the time complexity of such an algorithm would be N_F times the size of $\mathcal{S}_R$, where N_F is the number of repeating states; note that due to data interpretations, N_F can be very large.
- A third option, which is what we follow, is to maintain strongly connected components (SCCs) in $\mathcal{S}_R$ dynamically using an incremental algorithm. We terminate, reporting a cycle, as soon as the explored part of $\mathcal{S}_R$ starts containing a non-trivial SCC with a repeating state or a fair summary transition in it. However, linear time incremental algorithms for maintaining SCCs are not known. While we could use heuristically tuned online algorithms such as in [19], we have chosen instead to use an adaptation of Gabow's algorithm as it uses simpler data-structures.

Our algorithm Fair-Cycle-Detect consists of two subroutines: one explores the state space of the ERSM and discovers new transitions in $\mathcal{S}_R$ (including summary transitions), while the other updates the SCCs in the discovered graph. The former algorithm is essentially the algorithm Reachability of the previous section while the latter is an adaptation of a path-based DFS algorithm by Gabow [16] that finds SCCs in a graph.

Gabow's algorithm finds SCCs in a graph via a DFS on it. As soon as a back edge is identified, it contracts the cycle formed by it into an SCC, and, finally, outputs the SCCs in a topological order. This algorithm has an early termination property, because if an SCC introduced by a back edge contains a repeating state or a fair summary transition, we can terminate immediately.

Let us now describe an optimization that changes the structure of our algorithm. Since Gabow's algorithm essentially explores its graph using a depth-first search and the state-space exploration of the ERSM done by REACHABILITY is also essentially a DFS, these two can be combined easily. However, in REACHABILITY, when a new summary is discovered, the control shifts to the returns corresponding to this summary, which can be in an entirely different part of the graph. Since such a 'jump' requires us to restart our SCC algorithm, we prefer to process the summary transitions later, after the current DFS is over.

Consequently, our search-space exploration algorithm is the same as REACHABILITY, except that when a summary is discovered, the returns corresponding to it are not pursued and instead the new summary transitions are recorded in a set *Summ*. The dynamic SCC algorithm processes these transitions and updates the SCCs. When the exploration stops (or the current *search phase* ends), we add the summary transitions in *Summ* and run the dynamic SCC algorithm once more to effect the changes. Then we call the exploration algorithm again and ask it to proceed from the return states corresponding to the newly discovered summary transitions in *Summ*. We terminate, concluding that there is no fair cycle, if no new summary is found at the end of a search phase. Figure 3 and Figure 4 give the pseudocode of the entire algorithm FAIR-CYCLE-DETECT.

SCC-SEARCH explores the transition system $\mathcal{S}_R$ recursively, feeding every new transition to the procedure UPDATE-SCCS, which uses two stacks [16] to update the data structures it uses to remember the SCCs, and halts if the new transition introduces a fair cycle. To perform this update, it may have to do a DFS on the graph of discovered SCCs; however, since the edges fed to it in a phase are in DFS order, it only needs one cache of "visited" SCCs per phase.

While backtracking from the search, the procedure CREATE-COMPONENT, which marks an SCC to be used in the next phase, is called. Finally, the procedure COLLAPSE-SCCS takes in a set of summary edges found in the previous search phase (fair summaries are kept track of using a special bit b) and updates the current graph G of SCCs with them, terminating if it finds a fair cycle.

The correctness of this algorithm is guaranteed by the following theorem:

Theorem 3. *Given an ERSM A, a set R of repeating nodes, and the entry state $s_{init} = \langle en_p, \gamma_{in}, \lambda_{p_{in}} \rangle$, if the algorithm* FAIR-CYCLE-DETECT *halts, then it prints "Fair cycle found" iff there is a run of A that has infinitely many configurations ψ' with $Currnode(\psi') \in R$. Furthermore, if the state-graph $\mathcal{S}_R$ corresponding to A is finite, then* FAIR-CYCLE-DETECT *always halts.* □

Recall that in every search phase, we need to perform a search on the graph of SCCs. The total number of search phases in FAIR-CYCLE-DETECT is bounded by the number of possible summaries in $\mathcal{S}_R$. Let N be the maximum, over all component graphs S_i, of the number of pairs (s_{en}, s_{ex}), where s_{ex} is an exit state in S_i and is reachable from entry state s_{en} of S_i. Then $\mathcal{S}_R$ can have at most kN search phases, where k is the number of components in A. Hence, we have:

Fair-Cycle-detect()

graph $G \leftarrow (\{s_{in}\}, \emptyset)$; *Visited* $\leftarrow \emptyset$; $S_{in} = \{(s_{in}, s_{in}, (s_{in} \in R))\}$
repeat
    Summ $\leftarrow \emptyset$; Initialize-SCC-Update()
    for $(s, s_{en}, b) \in S_{in}$
    do SCC-Search (s, s_{en}, b)
    Collapse-SCCs (*Summ*)
    if Collapse-SCCs finds a fair nontrivial SCC
        then print ("Fair cycle found") ; *break*
    $S_{in} \leftarrow \{(s_{ret}, s', b) : \exists s.(s'_{en}, (s, s_{ret}), b) \in Summ\}$
until *Summ* $= \emptyset$
print ("No fair cycle")

Fig. 3. Fair cycle detection algorithm

Theorem 4. *Let A be an ERSM, R be a set of repeating nodes and $\mathcal{S}_R$ be the associated state graph. Let n and m be the number of states and edges, β be a bound on the maximum number of reachable entry states* and *reachable exit states in any S_i, in the reachable part of $\mathcal{S}_R$. Then* Fair-Cycle-Detect *takes $O(kN(m\beta + n\beta^2))$ time to terminate and uses space $O(n\beta^2 + m)$.* □

Note that N is bounded by β^2. While Fair-Cycle-Detect does not run in time linear in the size of $\mathcal{S}_R$, it has the early-termination property and some "on-the-fly" properties.

5 Vera

Vera is a Java implementation of the algorithms for reachability and fair cycle detection presented in this paper. In this section, we highlight its main features and compare it with Moped [13], a popular BDD-based LTL model checker for pushdown systems.

Input language *Boolean programs*, introduced in [3] and used in the Slam verification process [4], are abstractions of imperative programs that retain most of the control structures available in a C-like language but only allow variables and expressions of boolean type. These abstractions permit procedure calls with call-by-value parameter passing and recursion; procedures can return vectors of expressions. Global and local declarations of variables are permitted. Allowed statements include parallel assignment (where a list of variables may be assigned in parallel, either by a list of expressions or by a vector returned by a procedure), "goto" jumps, "if-else" branches, and "while" loops. Non-determinism is permitted both in branches and loops.

Vera accepts boolean programs as inputs; it also admits a bounded-integer data type and arithmetic expressions on variables declared as such. These abstractions are translated into ERSMs internally before the algorithms for reachability and fair cycle detection are applied.

```
SCC-Search(s, s_en, b)
Visited ← Visited ∪ {(s, s_en, b)}
if  s is an exit state in component i
   then VisitedExits[i, s_en] ← VisitedExits[i, s_en] ∪ {(s, b)}
        for (s', s'_en, b') ∈ VisitedCalls[i, s_en]
        do s_ret = GetReturnState(s', s);  b_ret = b ∨ b' ∨ (s_ret ∈ R)
           if (s_ret, s'_en, b_ret) ∉ Visited
              then Summ ← Summ ∪ {((s', s_ret), s'_en, b_ret)}
   else for e ∈ Edges_I(s)
        do if s satisfies guard of e
              then s' ← Apply(e, s);  b' = b ∨ (s' ∈ R);  Update-SCCs(s, s', b');
                   if Update-SCCs finds a fair nontrivial SCC
                      then print ("Fair cycle found") ; break
                   if (s', s_en, b') ∉ Visited
                      then SCC-Search(s', s_en, b')
        for e ∈ Edges_C(s)
        do if s satisfies guard of e
              then m ← Y(e);  s' ← GetEntryState(e, s);
                   b' = (s' ∈ R);  Update-SCCs(s, s', b')
                   VisitedCalls[m, s'] ← VisitedCalls[m, s'] ∪ {(s, s_en, b)}
                   if (s', s_en, b') ∉ Visited
                      then SCC-Search(s', s_en, b')
                      else for (s_ex, b_ex) ∈ VisitedExits[Y(e), s']
                           do s_ret = GetReturnState(s, s_ex)
                              b_ret = b_ex ∨ b' ∨ (s_ret ∈ R)
                              if (s_ret, s_en, b_ret) ∉ Visited
                                 then SCC-Search(s_ret, s_en, b_ret)
Create-Component ()
```

Fig. 4. Procedure SCC-Search

Specifying properties One way to specify target (or repeating) nodes in Vera is to list a set of *target (or repeating) labels* along with the input. Any control location marked by such a label translates into a target or repeating node. The target (repeating) set may also be specified by a *monitor*.

A *monitor*, in our context, is a finite automaton M with edges labeled by guards on global and local variables in A, and a set of states identified as target states. The definition of the product P of M and A is standard: a configuration of P consists of a configuration of A and the current state of M, and progress along a monitor is allowed only if the current variables satisfy the guard on it. A target (or repeating) node in P is one where the current state of M is a target. Given a monitor and the ERSM underlying an input program, Vera can perform reachability (cycle) analysis for the product ERSM P.

5.1 Experiments

Slam regression testing examples We ran Vera on the regression test suite for Slam: a collection of 64 C programs which, after abstraction in Slam,

Example	Lines	Globals	Locals	Reachable	Visited	VERA time(s)	MOPED time (s)
n-mutex1	439	3	13	Yes	274	0.06	0.04
p-mutex33	460	6	21	Yes	702	0.21	0.14
n-i2o-simple	347	2	3	Yes	94	0.08	0.01
n-list-22	305	0	15	Yes	316	0.07	0.02
p-mutex34	466	6	21	No	14144	6.17	0.08
p-farray	306	0	8	No	1304	0.18	0.01
p-nbebop-test	239	0	16	No	75524	151.85	0.04
p-srdriver	1454	10	36	No	-	-	0.29

Fig. 5. Experiments on the SLAM regression test suite

give boolean programs whose lengths range between 80 and 1450 lines. In each case, the query was: *is the control location labeled as SLIC_ERROR reachable?* The experiments were run on a machine with 2GB of RAM and two 1.4 GHz CPUs. Measurements on a few representative examples are tabulated in Figure 5. The first three columns show the number of lines of code, the number of global variables, and the maximum number of local variables in a procedure (recall that the number of ERSM states is exponential in the last two parameters). The next column gives the answer to the query. The next two columns give the number of visited states at termination, and total runtime in seconds. The final column shows the runtime of the MOPED model checker on the same example.

In the first four examples, where the target set is reachable, VERA seems to find a reachability witness easily. In the next four cases, where it has to generate the entire reachable state space, it performs much worse than MOPED. Particularly, in the last case, where there there may be as many as 10 uninitialized globals and 36 uninitialized locals in any procedure, the state space is too large for our procedure to terminate. On the other hand, in examples such as `p-mutex34` where VERA works better, there are complex conditions on edges but the number of uninitialized variables is not very high.

Quicksort Among the examples that come with MOPED is an abstraction of a buggy quicksort routine (`quicksort_error.pds`). The routine has two nondeterministically chosen integer inputs and can run into an infinite loop for some input values. While there exists a short witness to this error, it is by no means trivial.

We use MOPED and VERA to find this witness. To do this in VERA, we write a simple monitor and run the fair cycle detection module. We find that VERA's early termination capability lets it identify a cycle very fast, even when inputs have large ranges and, consequently, the set of reachable states is very large. The symbolic algorithm for MOPED, however, becomes prohibitively expensive as the number of bits in an integer (N) is increases, and does not terminate for $N = 10$ or above (see Figure 6).

We also compared VERA and MOPED on a trivial reachability property: whether the program has *some* terminating run. VERA identifies a witness immediately, whereas in MOPED, an effect similar to Figure 6 is observed.

N	Vera runtime(s)	Vera visited states	Moped runtime(s)	Moped BDD nodes
4	0.08	95	0.16	40880
6	0.08	95	4.01	1.91×10^5
8	0.08	95	260.35	2.12×10^6
10	0.10	95	-	-
32	0.15	95	-	-

Fig. 6. Buggy quicksort

Abstraction of a Linux driver Finally, we ran Vera's reachability and cycle detection algorithms on a manual abstraction of the Perle Specialix RIO driver for Linux. This driver, 1100 lines long and previously identified as buggy by the Stanford metacompilation project [11], contains a double locking error. We abstract it manually into a 220-line Vera input file, keeping the basic control structure intact, modeling locks and process id-s by Vera variables, and replacing many of the control-flow conditions by nondeterminism. We write simple monitors to answer the following questions:

(1) Is there an execution where the same lock is acquired twice in a row?
(2) Is every lock that is acquired also released?

In the former case, there exists a reachability witness to an error state. For 4-bit integers, Vera detects the error in 0.18s after visiting 15 states (this figure stays more or less the same even as the size of the integer type is made larger). In the second case, our abstraction satisfies the property, and Vera has to generate the entire state space before it terminates. Because of a few uninitialized integer variables, this space is quite large. For $N = 2$, it takes 50.92s. For higher values of N, Vera does not terminate.

6 Conclusions

We have presented algorithms for on-the-fly reachability and fair cycle detection for extended recursive state machines. Algorithmically, on-the-fly detection of cycles deserves further exploration. It is closely related to the problem of dynamic data structures for graphs where insertions are allowed, and queries check existence of cycles containing repeating nodes. It is open whether the worst-case quadratic bound of our cycle-detection algorithms can be improved. It would be interesting to know whether online SCC algorithms are essential to detect fair cycles in ERSMs on-the-fly, i.e. whether faster algorithms for on-the-fly traversal of ERSMs would necessarily imply faster online algorithms for cycle detection. Our implementation in Vera and experimentation support the hypothesis that on-the-fly model checking is a viable, and sometimes more effective, alternative to symbolic checkers for verifying ERSMs. Future work will focus on optimizations, alternative strategies for cycle detection, and applications to program analysis problems.

Acknowledgements: We thank Mihalis Yannakakis for useful discussions, Sriram Rajamani and Stefan Schwoon for the Slam regression test suite, and an anonymous referee for several relevant references in program analysis.

References

1. R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proc. Computer-Aided Verification*, LNCS 2102: 207–220, 2001.
2. T. Andrews, S. Qadeer, S.K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Proc. CAV'04*, LNCS 3114: 484–487, 2004.
3. T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN Workshop on Model Checking of Software*, LNCS 1885: 113–130, 2000.
4. T. Ball and S. Rajamani. The SLAM toolkit. In *Proc. CAV'01*, LNCS 2102, 2001.
5. M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestricted hierarchical state machines. In *Proc. ICALP '01*, LNCS 2076: 652–666, 2001.
6. A. Boujjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Applications to model checking. In *CONCUR'97*, LNCS 1243, 1997.
7. O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. *Theoretical Computer Science*, 221:251–270, 1999.
8. J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. of Intl. Conf. on Software Engg.*, pages 439–448. 2000.
9. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.
10. 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.
11. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. 4th USENIX OSDI*, pages 1–16, 2000.
12. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification, 12th International Conference*, LNCS 1855, pages 232–247. Springer, 2000.
13. J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. *Lecture Notes in Computer Science*, 2102:324+, 2001.
14. M. Fähndrich, J.S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. *Proc. PLDI '98*, pages 85–96, 1998.
15. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *Proc. Workshop on Verification of Infinite State Systems*, volume 9 of *Electronic Notes in Theor. Comp. Sci.* Elsevier, 1997.
16. H. Gabow. Path-based depth-first search for strong and biconnected components. *Inf. Process. Lett.*, 74(3-4):107–114, 2000.
17. T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV 02: Proc. of 14th Conf. on Computer Aided Verification*, LNCS 2404, pages 526–538, 2002.
18. G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
19. D.J. Pearce, P.H.J. Kelly, and C. Hankin. Online cycle detection and difference propagation for pointer analysis. *Software Quality Journal*, 12(4):311–337, 2004.
20. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. *POPL '95*, pages 49–61, 1995.