Tracing the executions of concurrent programs

Elsa Gunter [a] Doron Peled [b]

[a] *Department of Computer Science*
New Jersey Institute of Technology
University Heights
Newark, NJ, 07102-1982 USA

[b] *Department of Elect. & Comp. Eng.*
The University of Texas at Austin
Austin, TX 78712 USA

Abstract

Checking the reliability of software is an ever growing challenge. Fully automatic tools that attempt to cover the entire state space often fail because of state explosion. We present instead a tool that employs some less-ambitious but useful methods to assist in software debugging. The tool provides an automatic translation of the code into visual flowcharts, allowing the user to interactively select execution paths. The tool assists the user by calculating path conditions and exploring the neighborhood of the paths. The tool also allow the user to interactively step through the execution of the program, directed by temporal formulas interpreted over finite sequences. We will show several different ways of using these capabilities for debugging sequential and concurrent programs.

1 Introduction

Computer programs have nowadays quite a different magnitude and complexity than two or even one decade ago. Software projects are undertaken by large teams of programmers, writing many thousands line of code. Many newly developed software systems include parallelism. Different parts of the system may reside on different machines, sometimes in different locations, and interact via some coordination mechanism such as message passing. The software development teams, like the software itself, work in parallel, interfacing with each other in order to complete a coherent product. Over the last couple of decades, there have been many attempts to develop techniques and tools for enhancing the reliability of software.

The earliest, and still most widely used method, is *software testing* [15]. This method stipulates that experienced testers (usually being highly qualified programmers) construct a collection of test cases, usually executions of the

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

program, to be checked. The test cases are a *sample* of the executions of the program, and are supposed to provide a good coverage that attempts to catch most if not all of the errors. The construction of the test cases is often based on the experience and intuition of the tester. They are often generated by having the tester first examine the code, with some commonly recurring errors, e.g., having array indexes out of bound, and trying to produce executions that will manifest such failures. Testing methods highly depend on the quality of the tester and are known to remove many of the programming errors, but not all of them.

A more recent method is *program verification* [2,6], attempting to formally prove that a program is correct, with respect to some correctness criteria and specification, and within some proof system. Deductive theorem proving is axiomatic based and tend to be quite costly. It has been mostly demonstrated on small examples, and does not scale up very well. Nevertheless, it has important contributions to the way programmers think when developing software. In particular, program verification suggests the useful idea of an *invariant*, and the related *precondition* and *postcondition*.

The automatic verification of finite state systems, called *model checking* [3,4], attempts to perform the verification task with minimal human intervention for the limited case where the system has finitely many states. A naive attempt to check all the system states by systematic enumeration often fails due to the state explosion problem. Many heuristic methods are employed to alleviate this problem. It is still true that with today's methods, a comprehensive verification of full scale software is still a long term target. One main constraint of model checking is that it can find an error only once the property to be checked is formally specified.

The approach we are taking here is to develop an easy to use software analysis tool that exploits techniques derived from the various software reliability methods mentioned above. Preliminary versions of some features of the tool are described in [11,12]. We are guided by several principles. The first is to restrict the objective to exploring execution paths, consisting of a sequence of program states or program instructions. We do not attempt to solve the entire program correctness program, nor to provide a comprehensive state space search or analysis. We deal with execution sequences and paths in the code of the program, and provide various algorithms to explore and analyze such paths. Although the tool is capable of assisting in obtaining a formal proof of a program (in the case of partial correctness of sequential programs, as will be demonstrated), it is not its main intended use.

The second principle is the use of visual and interactive methods. We believe that it is easier to illustrate software issues using a visual formalism, as demonstrated by UML tools [5]. Allowing user interaction in the process does not necessarily mean that we are not capable of automatically providing information, but rather that we are exploiting the user's experience and intuition to guide the debugging process.

2 Path Operations

Software testing is based on inspecting paths. Therefore, it is of great importance to allow convenient selection of execution paths. Different coverage techniques suggest criteria for the appropriate coverage of a program. Our tool leaves the choice of paths to the user. Once the source code is compiled into a flow chart, or a collection of flow charts, the user can choose the test path by clicking on the nodes on the flow charts.

The selected path appears also in a separate window, where each line lists the selected node, the process and the shape (the lines are also indented according to the number of the process to which they belong). In order to make the connection between the code, the flow chart and the selected path clear, again sensitive highlighting is used. For example, when the cursor points to some node in the path window, the corresponding node in the flow chart is highlighted, as is the corresponding text of the process.

Once a path is fixed, the condition to execute it is calculated. The tool allows altering the path by removing nodes from the end, in reverse order. This allows, for example, the selection of an alternative choice for a condition, after the nodes that were chosen past that condition are removed. Another way to alter a path is to use the same transitions but allow a different interleaving of them. When dealing with concurrent programs, the way the execution of transitions from different nodes are interleaved is perhaps the most important source of problems. The tool allows the user to flip the order of adjacent transitions on the path, if they belong to different processes.

An important information that is provided is the condition to execute a selected path. An important point to note is that an execution path in a set of flow charts is really a sequence of edges, which when restricted to each of the processes involved, forms a contiguous sequence. Selecting the node does not always tell us how it executed: a condition may be true or it may be false. The execution of a condition node, corresponding to an if-the-else condition, a while condition, or similar, is determined by whether its "true" or the "false" labeled edge was selected, which we can know by fixing the successor node to the test in the process. Thus, if a condition node is the last node of some process in the selected path, it would not contribute to the path condition, as the information about how it is executed is not given.

Let $\xi = s_1 s_2 \ldots s_n$ be a sequence of nodes. For each node s_i on the path, we define:

$type(s_i)$ is the type of the transition in s_i. This can be one of the following: *begin, end, condition, wait, assign.* A *condition* node is obtained from an *if* or *while* statement. A *wait* node is similar to a *condition*, but has only a *true* exit. It is used for synchronization of concurrent processes, as a process cannot continue from this node unless the condition holds.

$proc(s_i)$ is the process to which s_i belongs.

$cond(s_i)$ is the condition on s_i, in case that s_i is either a *condition* or a *wait* node.

$branch(s_i)$ is the label on a node s_i which is a *condition* if it has a successor in the path that belongs to the same process, and is "undefined" otherwise.

$expr(s_i)$ is the expression assigned to some variable, in case that s_i is an *assign* statement.

$var(s_i)$ is the variable assigned, in case s_i is an *assign* statement.

$p[v/e]$ is the predicate p where all the (free) occurrences of the variable v are replaced by the expression e.

The following is the algorithm used to calculate the path condition. It works backwards, from tail to head of the sequence.

```
current_pred := 'true';
for i := n to 1 do
    begin case type(s_i) do
        condition⇒
            case branch(s_i) do
                'true'⇒
                    current_pred := current_pred∧cond(s_i)
                'false'⇒
                    current_pred := current_pred ∧¬cond(s_i)
                'undefined'⇒
                    current_pred := current_pred
            end case
        wait⇒
            current_pred := current_pred∧cond(s_i)
        assign⇒
            current_pred := current_pred [ var(s_i)/expr(s_i) ]
      end case
      simplify(current_pred)
    end
```

The meaning of the calculated path condition is different for sequential and concurrent programs. In a sequential program, consisting of one process, the precondition expresses all the possible assignments that would *ensure* executing the selected path, starting from the first selected node. When concurrency is allowed, the precondition expresses the assignments that would make the execution of the selected path *possible*. Thus, when concurrency is present, the path precondition does not guarantee that the selected path is executed, as there might be alternatives paths with the same variable assignments.

Simplifying expressions is a hard task. For one thing, it is not clear that there is a good measure in which one expression is simpler than the other. Another reason is that in general, deciding the satisfiability or the validity

of first order formulas is undecidable. However, such limitations should not discard heuristic attempts to simplify formulas, and for some smaller classes of formulas such decision procedures do exist.

The approach for simplifying first order formulas is first to try to apply several simple term-rewriting rules in order to perform some common-sense and general purpose simplifications. In addition, it is checked whether the formula is of the special form of *Presburger arithmetic*, i.e., allowing addition, multiplication by a constant, and comparison. If this is the case, one can use some decision procedures to simplify the formula.

The simplification that is performed includes the following rewriting:

- Boolean simplification, e.g., $\varphi \wedge \mathit{true}$ is converted into φ, and $\varphi \wedge \mathit{false}$ is converted into *false*.
- Eliminating constant comparison, e.g., replacing $1 > 2$ by *false*.
- Constant substitution. For example, in the formula $(x = 5) \wedge \varphi$, every (free) occurrence of x in φ is replaced by 5.
- Arithmetic cancellation. For example, the expression $(x+2)-3$ is simplified into $x - 1$, and $x * 0$ is replaced by 0. However, notice that $(x/2) * 2$ is not simplified, as integer division is not the inverse of integer multiplication.

In case the formula is in Presburger arithmetic, we can decide if the formula φ is unsatisfiable, i.e., is constantly *false*, or if it is valid, i.e., constantly *true*. The first case is done by deciding on $\neg\exists x_1 \exists x_2 \ldots \exists x_n\, \varphi$, and the second case is done by deciding on $\forall x_1 \forall x_2 \ldots \forall x_n \varphi$, where $x_1 \ldots x_n$ are the variables that appear in φ. If the formula is not of Presburger arithmetic, one can still try to decide whether each maximal Presburger subformula of it is equivalent to *true* or *false*.

A prior tool that used symbolic evaluation of paths of programs is described in [13]. This early tool (1976) could calculate path conditions of a selected path in a sequential PL/I program. The path was selected from the text of the program. The calculation used forward symbolic execution.

3 Examples

Consider the simple protocol in Figure 1, intended to obtain mutual exclusion. In this protocol, a process can enter the critical section if the value of a shared variable `turn` does not have the value of the other process. The code for the first process is as follows:

```
begin
  while true do
  begin
    while turn=1 do begin (* no-op *) end;
    (* critical section *)
    turn:=1
```

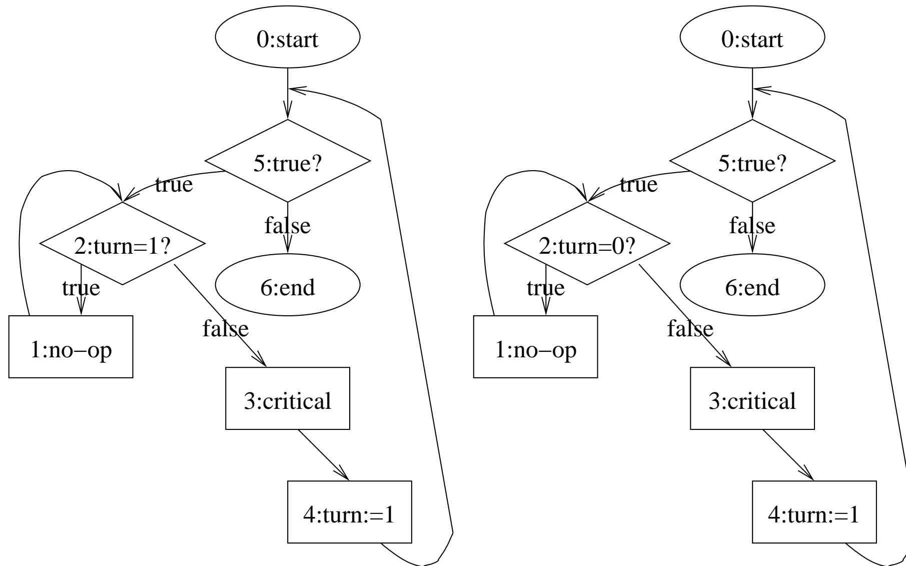

Fig. 1. A Mutual exclusion example

```
  end
end.
```

The second process is similar, with constant values 1 changed to 0.

When we select the following path, which admits the second process *mutex1*, while the first process *mutex0* is busy waiting as follows:

```
(mutex0 : 0)
  (mutex1 : 0)
  <mutex1 : 5>
<mutex0 : 5>
  <mutex1 : 2>
<mutex0 : 2>
  [mutex1 : 3]
[mutex0 : 1]
```

we get the path condition *turn = 1*, namely that the second process will get first into its critical section if initially the value of the variable *turn* is 1. When we check a path that gets immediately into both critical sections, namely:

```
(mutex0 : 0)
  (mutex1 : 0)
  <mutex1 : 5>
<mutex0 : 5>
<mutex0 : 2>
  <mutex1 : 2>
[mutex0 : 3]
  [mutex1 : 3]
```

we get a path condition $turn \neq 1 \wedge turn \neq 0$. This condition suggests that we will not get a mutual exclusion if the initial value would be, say, 3. This indicates an error in the design of the protocol. The problem is that a process enters its critical section if *turn* is *not* set to the value of the other process. This can be fixed by allowing a process to enter the critical section if *turn* is set to its own value.

Applications

The combination of a graphical interactive tool, the ability to calculate path conditions and simplifying them has several uses for gaining intuition about programs and debugging them.

Proving partial correctness

The *partial correctness* of a sequential program is proved with respect to some initial condition φ and final assertion ψ. The notation $\{\varphi\}P\{\psi\}$ means that if φ holds when the execution begins, and the program terminates (partial correctness does not include the assertion that the program terminates), then upon termination ψ holds.

In order to verify the partial correctness of a program, it is sufficient to do the following. We first annotate edges of the flowchart with invariants that hold while control passes these edges. The edge from the *begin* node is annotated with the initial condition, and the edge into the *end* node is annotated with the final assertion. We need to find edges that will cut all the executions of the program into finitely many finite paths σ_1, σ_2, ... Each such path includes an enter and an exit edge. The enter edge to path σ_i is annotated with an invariant μ_i, and the exit edge is annotated with an invariant ν_i. We have to prove the following:

If σ_i is executed from a state satisfying μ_i, it ends in a state satisfying ν_i.

In order to prove that, we can add interactively a new process, consisting of the following code:

begin wait ν_i end.

We let the system compile the code into a trivial three nodes flowchart. We select the path σ_i and then select the node for ν_i and the corresponding end node. The tool generates a path condition δ_i. This is the condition for executing the path σ_i and reaching a state satisfying ν_i. We have to prove the logical implication $\mu_i \rightarrow \delta_i$. (By building another process with this implication as a *wait* condition, similar to the above created process, we can apply the system simplification over this formula, but it is not guaranteed that the system will succeed in simplifying it to *true* even if the formula is valid.)

Generate test cases

In white-box testing, we are often interested in finding test cases for covering various executions of the program. In order to cover a path in the execution, we may select the path in our system and let the system calculate the path condition. In order to generate a test case that will pass through that path, we need to instantiate the path conditions with a satisfying assignment.

Depending on the required program coverage, we may want to specialize the path beyond the execution of its nodes. For example, passing a node s labeled with the condition $x = 3$ *or* $y = 4$, we may want to test the path when $x = 3 \wedge y \neq 4$ and again when $x \neq 3 \wedge y = 4$ (when both $x \neq 3 \wedge y \neq 4$, the path will not be selected). We can create, e.g., the first case, by interactively creating a new process with the code:

```
begin wait x=3 and not y=4 end.
```

We select the desired path, but add the new node for the `wait` statement (and the corresponding *end*) after selecting the node s. Then we continue with the original path.

Playing "what if" games

The interactive symbolic manipulation approach is very flexible and allows us to study the connection between the program variables. It does not limit us to start the program at the beginning or to initialize all the variables. By adding additional processes that interact and interfere with our code, we can simulate many mental activities that are related to code inspection in a formal way.

For example, we can check what happens if the program executes from some point with $x = 0$, $y = 3$ and $z = 4$. We do that by generating a new process:

```
begin x:=0; y:=3; z:=4 end.
```

We select these nodes followed by the nodes of the original path. The path condition can still be a formula, where the other program variables appear as free variables. If there are no other program variables, selecting the path with amount to simulating it; the automatic simplification of formulas involving constants will result in that in this case the path condition will be limited to *true* and *false*. The former means, for sequential programs, that 'this path will be executed as selected when started with these values'. For concurrent programs it means 'this path may be executed, provided appropriate nondeterministic scheduling'. A *false* means (in both cases) that the path cannot be executed starting with these values.

Exploring the neighborhood of an execution

Given an execution path in which an error occurs, it is important to be able to inspect similar paths, its *neighborhood* [17]. Looking at related paths can

help us in pinpointing the exact location of the error (it does not have to be the first point where we start to obtain wrong values). Looking at related paths can also help us to suggest the fix for the bug and to check whether the fix will only solve the problem locally. The tool allows us to look at related paths of two kinds:

(i) Paths with mutual prefix. Assuming that some error occurs in some prefix of the paths, we can easily check other paths with the same prefix and see whether they have the same problem.

(ii) Path with the same instructions, but with a different interleaved order. In concurrent systems, the possibility of ordering events in different order is a major contributor to programming errors.

4 A Temporal Debuger

We extend the use of a temporal specification logic for interactively controlling the debugging of systems. We allow specifying temporal properties of *finite sequences.* Our debugger is enriched with the ability to progress from one step to another via a finite sequence of states that satisfy a temporal property.

The usual mode of debugging involves stepping through the states of a system (program) by executing one or several transitions (with different granularities, e.g., a transition can involve the the execution of a procedure). Debugging concurrent systems is harder, since there are several cooperating processes that need to be monitored. Stepping through the different transitions can be applied in many different ways. Instead, we allow applying a temporal property that describes a finite sequence of concurrent events that need to be executed from the current state, leaping into the next state.

We interpret linear temporal logic (LTL) on finite sequences. The automatic translation from LTL to finite state automata in [8] is adapted to include the finite case.

One of the most popular specification formalisms for concurrent and reactive systems is Linear Temporal Logic (LTL) [18]. Its syntax is as follows:

$$\varphi ::= (\varphi) \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid \overline{\bigcirc}\varphi$$
$$\mid \Box\varphi \mid \Diamond\varphi \mid \varphi\, \mathcal{U}\, \varphi \mid \varphi\, \mathcal{V}\, \varphi \mid p$$

where $p \in \mathcal{P}$, with $\mathcal{P}$ a set of propositional letters. We denote a propositional sequence over $2^{\mathcal{P}}$ by σ, its ith state (where the first state is numbered 0) by $\sigma(i)$, and its suffix starting from the ith state by $\sigma^{(i)}$. Let $|\sigma|$ be the length of the sequence Σ, which is a natural number. The semantic interpretation of LTL is as follows:

- $\sigma \models \bigcirc\varphi$ iff $|\sigma| > 1$ and $\sigma^{(1)} \models \varphi$.
- $\sigma \models \varphi\, \mathcal{U}\, \psi$ iff $\sigma^{(j)} \models \psi$ for some $0 \leq j < |\sigma|$ so that for each $0 \leq i < j$, $\sigma^{(i)} \models \varphi$.
- $\sigma \models \neg\varphi$ iff it is not the case that $\sigma \models \varphi$.

- $\sigma \models \varphi \vee \psi$ iff either $\sigma \models \varphi$ or $\sigma \models \psi$.
- $\sigma \models p$ iff $|\sigma| > 0$ and $\sigma(0) \models p$.

The rest of the operators can be defined using the above operators. In particular, $\overline{\bigcirc}\varphi = \neg \bigcirc \neg\varphi$, $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$, $\varphi \,\mathcal{V}\, \psi = \neg((\neg\varphi)\,\mathcal{U}\,(\neg\psi))$, $\mathit{true} = p \vee \neg p$, $\mathit{false} = p \wedge \neg p$, $\Box\varphi = \mathit{false}\,\mathcal{V}\,\varphi$, and $\Diamond\varphi = \mathit{true}\,\mathcal{U}\,\varphi$. The operator $\overline{\bigcirc}$ is a 'weak' version of the $\bigcirc$ operator. Whereas $\bigcirc\varphi$ means that φ holds in the suffix of the sequence starting from the next state, $\overline{\bigcirc}\varphi$ means that *if* the current state is not the last one in the sequence, *then* the suffix starting from the next state satisfies φ.

We distinguish between the operator $\bigcirc$, which we call *strong nexttime*, and $\overline{\bigcirc}$, which we call *weak nexttime*. Notice that

$$(\bigcirc\varphi) \wedge (\overline{\bigcirc}\psi) = \bigcirc(\varphi \wedge \psi), \tag{1}$$

since $\bigcirc\varphi$ already requires that there will be a next state. Another interesting observation is that the formula $\overline{\bigcirc}\mathit{false}$ holds in a state that is in deadlock or termination.

The operators $\mathcal{U}$ and $\mathcal{V}$ can be characterized using a recursive equation, which is useful for understanding the transformation algorithm, presented in the next section. Accordingly, $\varphi \,\mathcal{U}\, \psi = \psi \vee (\varphi \wedge \bigcirc\varphi \,\mathcal{U}\, \psi)$ and $\varphi \,\mathcal{V}\, \psi = \psi \wedge (\varphi \vee \overline{\bigcirc}(\varphi \,\mathcal{V}\, \psi))$.

We exploit temporal specification to control stepping through different states of a concurrent system. The basic operation of a debugger is to step between different states of a system in an effective way. While doing so, one can obtain further information about the behavior of the system.

Given the current global state of the system s, we are searching for a sequence $\xi = s_0 s_1 \ldots s_n$ such that

- $s_0 = s$.
- n is smaller than some limit given (perhaps as a default).
- $\xi \models \varphi$.

The *temporal stack* consists of the different sequences, used in the simulation or debugging obtained so far. It contains several *temporal steps*, each corresponding to some temporal formula that was satisfied. The end state of a temporal step is also the start state of the next step. We search for a temporal step that satisfies a current temporal formula. When such a step is found, it is added to the temporal stack. We can then have several options of how to continue the search, as detailed below.

Searching a path can be done using search on pairs: a state from the joint state space of the system, and a state of the property automaton. Furthermore, each new temporal formula requires a new copy of the search space. Recursion is handled within that space. Thus, when starting the search for formula φ_1, we use one copy of the state space. When seeking a new temporal step for φ_2, we start a fresh copy. If we backtrack on the second step, we backtrack the second search, looking for a new finite sequence that satisfies φ_2. If we remove

the last step, going back to the formula φ_1, we remove the second state space information, and backtrack the first state space search. For this reason, we need to keep enough information that will enable us to resume a search after other temporal steps where exercised and backtracked.

The debugging session consists of searching the system through the temporal stack. At each point we may do one of the following:

- Introduce a new temporal formula and attempt to search for a temporal step from the current state. The new temporal step is added to the search stack. A new automaton for the temporal formula is created, and the product of that automaton with the system automaton with new initial state of the current state is formed. The temporal step is found by finding a path to an accepting state in this product automaton.
- Remove a step. In this case, we are back one step in the stack. We forget about the most recent temporal formula given, and can replace it by a new one in order to continue the search. We also discard the temporal automaton and product automaton generated for that temporal step.
- Backtrack the most recent step. The search process of the latest step resumes from the place it was stopped using the automaton originally created for this temporal step. This is an attempt to find another way of satisfying the last given formula. We either find a new temporal step that replaces the previous one, or report that no such step exists (in this case, we are back one step in the stack and discard the automata created for this step).
- We allow also simple debugger steps, e.g., executing one statement in one process. Such steps can be described as trivial temporal steps (using the nexttime temporal operator).

There are further parameters for the choice of temporal steps, besides the minimality and maximality of the step.

- Allowing or disallowing a different step that ends with the same system state as before. In the former case, we may request an alternative step and reach exactly the same system state, but passes through a different path on the way. The latter case is easily obtained by adding a special flag to each system state that was found during the search.
- Allowing or disallowing the same sequence of system states to repeat. Such a repetition can happen, for example, in the following situation. The specification is of the form $(\Diamond p) \vee (\Diamond q)$. Consider a sequence of system states in which $(\neg p) \wedge (\neg q)$ holds until some state in which both p and q start to hold, simultaneously. Such a sequence can be paired up with different property automaton states to generate two different paths.
- Allowing *all* possible paths with sequence of system states that satisfy the temporal step formula φ or only a *subset* of them. Typical searches like depth first or breadth first search do not pass through all possible paths that satisfy a given formula φ. If a state (in our case, a pair) participated

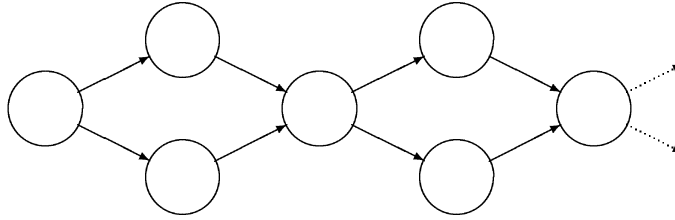

Fig. 2. Exponential number of sequences

before in the search, we do not continue the search in that direction. For this reason, the number of paths that can be obtained in this way is limited, and, on the other hand, the search is efficient. There are topological cases where requiring all the paths results in exponentially more paths than obtained with the above mentioned search strategies, see e.g., the case in Figure 2.

The case where similar sequences are generated as a result of repeated backtracking may seem at first to be less useful for debugging. Intuitively, we may give up exhaustiveness for the possibility of stepping through quite different sequences. However, there is a very practical case in which we may have less choice in selecting the kind of search and the effect of backtracking. Specifically, in many cases keeping several states in memory at the same time and comparing different states may be impractical. In this case, we may want to perform memoryless search, as developed for the Verisoft system [9]. In this case, we may perform breadth first search with increasingly higher depth (up to some user defined limit). We keep in the search stack only information that allows us to generate different sequences according to some order, and to regenerate a state. Such information may include the identification of the transitions that were executed from the initial states.

Consider for example the two processes in Figure 1. We can try to check the property

$$mutex1\ at\ o\,\mathcal{U}\,mutex0\ at\ 3$$

This checks whether process $mutex0$ can get into its critical section while process $mutex1$ has not yet moved. Since the system initializes all the variables to 0, we obtain a path:

```
(mutex0:0)
  (mutex1:0)
<mutex0:5>
<mutex0:2>
[mutex0:3]
```

We can now add another temporal step, by checking:

$$\Diamond mutex1\ at\ 3$$

We may obtain the following temporal step:

```
<mutex0:4>
  <mutex1:5>
```

```
  <mutex1:2>
  [mutex1:3]
```

Alternatively (depending on the search order and search option used) we can obtain the following step:

```
  <mutex1:5>
  <mutex1:2>
  [mutex1:1]
<mutex0:4>
  <mutex1:2>
  <mutex1:3>
```

References

[1] R. Alur, G. Holzmann, D. Peled, An anlyzer for message sequence charts, Software: Concepts and Tools, 1(1996), 70–77.

[2] K. R. Apt, E. R. Olderog, *Verification of Sequential and Concurrent Programs*, Springer-Verlag, 1991 (second edition, 1997).

[3] E. M. Clarke, E. A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic. Workshop on Logic of Programs, Yorktown Heights, NY, Lecture Notes in Computer Science 131, Springer-Verlag, 1981, 52–71.

[4] E. A. Emerson, E. M. Clarke, Characterizing correctness properties of parallel programs using fixpoints, International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 85, Springer-Verlag, July 1980, 169–181.

[5] M. Fowler, K. Scott, *UML Distilled : Applying the Standard Object Modeling Language*, Addison-Wesley, 1997.

[6] N. Francez, *Program Verification*, Addison Wesley, 1992.

[7] E.R. Gansner, S.C. North, An open graph visualization system and its applications to software engineering, Software – Practice and Experience, 30(2000), 1203–1233.

[8] R. Gerth, D. Peled, M.Y. Vardi, P. Wolper, Simple On-the-fly Automatic Verification of Linear Temporal Logic, *PSTV95, Protocol Specification Testing and Verification*, 3–18, Chapman & Hall, 1995, Warsaw, Poland.

[9] P. Godefroid, Model checking for programming languages using Verisoft, POPL 1997, 174–186.

[10] G. Holzmann, Design and Validation of Computer Protocol, *Prentice Hall.*

[11] E. Gunter, D. Peled, Path Exploration Tool, TACAS 1999, LNCS 1579, Springer, 405–419.

[12] E. Gunter, D. Peled, Temporal debuging for concurrent systems, TACAS 2002, LNCS 2280, Springer, 431-444.

[13] J. C. King, Symbolic execution and program testing, Communication of the ACM, 1976, 385–394.

[14] ITU-T Recommendation Z.120, Message Sequence Chart (MSC), March 1993.

[15] G.J. Myers, The Art of Software Testing, John Wiley and Sons, 1979.

[16] B. Selic, G. Gullekson, P. T. Ward, Real-Time Object-Oriented Modeling, Wiley, 1993.

[17] N. Sharygina, D. Peled, A combined testing and verification approach for software reliability, FME 2001, LNCS 2021, 611–628.

[18] A. Pnueli, The temporal logic of programs, 18th IEEE symposium on Foundation of Computer Science, 1977, 46–57.