Research Institute for Advanced Computer Science

Synthesizing Dynamic Programming Algorithms from Linear Temporal Logic Formulae

Grigore Roşu
Klaus Havelund

RIACS Technical Report
01.15

May 2001

Synthesizing Dynamic Programming Algorithms from Linear Temporal Logic Formulae

Grigore Roşu, RIACS
Klaus Havelund, Kestrel Technology

RIACS Technical Report
01.15

May 2001

The problem of testing a linear temporal logic (LTL) formula on a finite execution trace of events, generated by an executing program, occurs naturally in runtime analysis of software. We present an algorithm which takes an LTL formula and generates an efficient dynamic programming algorithm. The generated algorithm tests whether the LTL formula is satisfied by a finite trace of events given as input. The generated algorithm runs in linear time, its constant depending on the size of the LTL formula. The memory needed is constant, also depending on the size of the formula.

Synthesizing Dynamic Programming Algorithms from Linear Temporal Logic Formulae

Grigore Roşu[1] and Klaus Havelund[2]

[1] Research Institute for Advanced Computer Science
[2] QSS / Recom Technologies
`http://ase.arc.nasa.gov/{grosu,havelund}`
Automated Software Engineering Group
NASA Ames Research Center
Moffett Field, California, 94035, USA

Abstract. The problem of testing a linear temporal logic (LTL) formula on a finite execution trace of events, generated by an executing program, occurs naturally in runtime analysis of software. We present an algorithm which takes an LTL formula and generates an efficient dynamic programming algorithm. The generated algorithm tests whether the LTL formula is satisfied by a finite trace of events given as input. The generated algorithm runs in linear time, its constant depending on the size of the LTL formula. The memory needed is constant, also depending on the size of the formula.

1 Introduction

The work presented in this paper is part of an ambitious project at NASA Ames Research Center, called PathExplorer, that aims at developing a practical testing environment for NASA software developers. The basic idea of the project is to extract an execution trace of a concurrent program and then analyze it to detect errors. The errors we are considering at this stage are deadlocks, data races, and non-conformance with linear temporal logic specifications. Only the later issue is addressed in this paper.

Linear Temporal Logic (LTL) [17] is a logic for specifying properties of reactive and concurrent systems. The models of LTL are infinite execution traces, reflecting the behavior of such systems as ideally always being ready to respond to requests, operating systems being a typical example. LTL has been mainly used to specify properties of concurrent and interactive down-scaled models of real systems, so that fully formal correctness proofs could subsequently be carried out, for example using theorem provers or model checkers (see for example [12, 9]). However, such formal proof techniques are usually not scalable to real sized systems without a substantial effort to abstract the system more or less manually to a model which can be analyzed. Model checking of programs has received an increased attention from the formal methods community within the last couple of years, and several systems have emerged that can directly model check source code, such as Java and C [10, 21, 4, 14, 3, 16]. Stateless model checkers [20] try to avoid the abstraction process by not storing states. Although these systems provide high confidence, they scale less well because most of their internal algorithms are NP-complete or worse.

Testing scales well, and is by far the most used technique in practice to validate software systems. The merge of testing and temporal logic specification is an attempt to achieve the benefits of both approaches, while avoiding some of the pitfalls of adhoc testing and the complexity of full-blown theorem proving and model checking. Of course there is a price to pay in order to obtain a scalable technique: the loss of coverage. The suggested framework can only be used to examine single execution traces, and can therefore not be used to prove a system correct. Our work is based on the belief that software engineers are willing to trade coverage for scalability, so our goal is to provide tools that are completely automatic, implement very efficient algorithms and find *many* errors in programs. As mentioned previously, the work presented in this paper is part of larger effort to develop a set of dynamic analysis algorithms and to integrate these into a single tool named PathExplorer. Of particular additional interest are for example algorithms that can detect deadlock and data race potentials in a program, by examining a single arbitrary execution trace of the program, even though these errors do not occur in that trace. This can be achieved by analyzing the way locks are acquired and released. A deadlock potential can for example be detected by observing that two threads take two locks in different order. A collection of commercial tools already provide this kind of analysis: Visual Threads [7], which uses the Eraser algorithm [18] for detecting data races, and which works on C and C++ programs using Pthreads; Assure [1], which works on C++ programs using Pthreads; and finally Jprobe [19] for Java. In earlier work, we implemented data race detection and deadlock detection algorithms for Java in Java PathFinder [8]. It's our intention to extend this kind of technology by identifying other error patterns that can be detected this way. A major goal is to make PathExplorer adjustable to various programming languages and thus eventually deliver a Java PathExplorer as well as a C++ PathExplorer that share the same core algorithms but have different front ends. A longer term goal is to explore the use of conformance with a formal specification to achieve fault tolerance. The idea is that the failure may trigger a recovery action in the monitored program.

Following encouraging results using rewriting-based algorithms [11] implemented in Maude [2], in this paper we investigate more efficient algorithms for testing whether finite execution traces conform to LTL formulae. The idea of using LTL in program testing is not new. It has already been pursued in commercial tools such as TempRover (TR) [5], which has admittedly motivated us in a major way to start this work. In TR, one states LTL properties as annotations of the program, these being then replaced by appropriate code, that is executed whenever reached[1]. Thus, TR can be seen as an extension of a conventional programming language with LTL instructions. Inspired by the MaC [15]

[1] The implementation details of TR are not public.

tool, we decided to rather automatically instrument the bytecode or the object code of a program to generate events of interest during the execution. Such an event-based framework is well suited for program tracing in general, and has also been used to detect race conditions and deadlocks in the Visual Threads [7, 18] and Java PathFinder [8] tools. The trace of events can then be analyzed using a different tool which can even run on a different platform. One can also save various execution traces of a program and then have someone else analyze them at a different time, in a different place. We were thus rapidly faced with the following challenge:

> Given a finite execution trace t of events and an LTL formula φ, how efficiently can one test whether t satisfies φ?

A potential solution would be to translate the formula into an automaton and then take the synchronized product of the automaton and the execution trace. This is for example how Büchi automata are used in explicit-state model checkers for representing formulae [13, 6]. A Büchi automaton is a special automaton which accepts infinite traces (words): certain states are designated as *acceptance* states, and an infinite trace is in the language of the automaton if and only if it brings the automaton through an acceptance state infinitely often. A model checker can detect such infinite traces by hashing states and detect cycles that include acceptance states. We have decided *not* to use Büchi automata for a number of reasons.

- First, the translation of LTL formulae to Büchi automata is not trivial, especially if one strives for small automata. It is worth mentioning that other similar systems like Temporal Rover [5] and MaC [15] do not use Büchi automata either.
- Second, at a semantic level, Büchi automata are interpreted over infinite traces and it is not clear how to interpret them on finite traces. Consider for example a property such as $\Box(p \rightarrow \Diamond q)$, the automaton A generated from the formula, and a finite trace t that, according to the semantics, satisfies the formula. The naive suggestion would be to drive the automaton A by t until the end of the trace, and then observe whether the automaton is in an acceptance state or not. This will, however, generally not work. In experiments made using the LTL-to-Büchi automata translator in the SPIN system [13], such a trace may bring the automaton to a state that is not an acceptance state. Hence, one can generally not conclude anything from the resulting state. A potential solution would be to pretend that an infinite sequence of stuttering transitions is appended to the trace, where a stuttering transition does not satisfy any proposition. One could then examine whether such a stuttering sequence would bring the automaton from the state(s) resulting from the finite trace, through an acceptance state infinitely often. Hence, the stuttering should be shown to "finish off" the automaton correctly. However, even though such an interpretation is possible, a different issue is that our finite trace semantics of the *always* operator $\Box$ is different from the infinite trace semantics implied by Büchi automata.

– Third, we think that the dynamic programming methodology that we suggest yields more efficient testing tools than ones based on Büchi automata. In fact, we claim that it is hard, if not impossible, to find more efficient algorithms than those presented in this paper.

In spite of their efficiency and elegance, the generated algorithms have a serious drawback: the execution trace needs to be visited backwards. This is a typical phenomenon in dynamic programming algorithms which, taking into account the continuously decreasing price of storage media, doesn't seem to be a practical problem if one wants to first generate the events and then analyze them. However, we admit that it can be a crucial issue when one wants to analyze the events as they are generated, warning the programmer of errors or potential errors while his/her program is being executed. We were not able to find a dynamic programming algorithm that travels the trace forwards, but we are confident that it can be done and post it as a challenge for the interested reader, mentioning that it would have a great impact on testing methodologies and tools. It is worth mentioning here that we *did* find and implement an algorithm that visits the events in the order they were generated [11], but it is not as efficient as the dynamic programming algorithms presented in this paper.

We'd like to warmly thank Rance Cleaveland, Dimitra Giannakopoulou and Willem Visser for interesting and productive technical discussions directly related to the effort in this paper, as well as Edmund Clarke, David Dill and Doron Drusinsky for general discussions on dynamic analysis of programs and its potential impact on computer aided verification.

2 Finite Trace Linear Temporal Logic

We briefly remind the reader the basic notions of finite trace linear temporal logic, including a recursive definition of the satisfaction relation between a finite trace and an LTL formula. The interested reader can check [11] for more on this subject.

We regard a trace as a finite sequence of events emitted by the program that we want to observe. Such events could indicate when variables are changed or when locks are acquired or released. Note that this view is slightly different from the traditional view where the trace is a sequence of program states, each state denoting the set of propositions that hold at that state. This view is consistent with our goal to define an LTL observer as a process that is detached from the program to be analyzed, receiving only observed events. To keep the presentation simple and our results general, we abstract away from the concrete contents of events and just define events as atoms. Similarly, we consider the basic propositions as simple as possible, also atoms, and say that a proposition a is satisfied by an event b if and only if $a = b$. In practice, one should necessarily consider appropriate notions of satisfaction of propositions by events or states generated by events. We consider that this is an interesting but too concrete problem depending upon the events that one wants to observe, so we do not approach it here.

Formulas. Assume that *Prop* is a set of atoms, called atomic propositions. Then *Formula* is the free extension of *Prop* under the standard propositional constants and operators *true*, *false*, $\neg\_$, $\_\vee\_$, $\_\wedge\_$, $\_\rightarrow\_$, $\_\leftrightarrow\_$, together with the classical temporal logic operators $\circ\_$, $\Box\_$, $\Diamond\_$, and $\_\ \mathcal{U}\ \_$ whose meaning will be given later.

Events and Traces. Suppose that *Event* is a set of events. As we previously mentioned, for the time being we consider that $Event = Prop$ is just a set of atoms. The set of finite traces is $Event^*$ which we'll denote *Trace*, where ϵ denotes the empty trace. Assume two partial functions $head : Trace \rightarrow Event$ and $tail : Trace \rightarrow Trace$ for taking the head and the tail of a trace, respectively, and a total function *length* returning the length of a finite trace. That is, $head(e\ t) = e$, $tail(e\ t) = t$, and $length(\epsilon) = 0$ and $length(e\ t) = 1 + length(t)$. Assume further for any trace $t = e_1 e_2 ... e_n$ that t_i, for some natural number $1 \leq i \leq n$, denotes the suffix trace $e_i e_{i+1} ... e_n$ that starts at position i, and that $t_{n+1} = \epsilon$; if $t = \epsilon$ then $n = 0$ and $t_1 = \epsilon$.

Satisfaction. The satisfaction relation $\models\ \subseteq Trace \times Formula$ defines when a trace t satisfies a formula φ, written $t \models \varphi$, and is defined inductively over the structure of the formulae as follows, where $p \in Prop$ is any atomic proposition and φ and ψ are any formulae:

$t \models true$		is always true,
$t \models false$		is always false,
$t \models p$	iff	$t \neq \epsilon$ and $head(t)$ is p,
$t \models \varphi \vee (\wedge, \rightarrow, \leftrightarrow)\ \psi$	iff	$t \models \varphi$ and (or, implies, iff) $t \models \psi$,
$t \models \circ\varphi$	iff	$t \neq \epsilon$ and $tail(t) \models \varphi$,
$t \models \Box\varphi$	iff	$(\forall\ 1 \leq i \leq length(t))\ t_i \models \varphi$,
$t \models \Diamond\varphi$	iff	$(\exists\ 1 \leq i \leq length(t) + 1)\ t_i \models \varphi$,
$t \models \varphi\ \mathcal{U}\ \psi$	iff	$(\exists\ 1 \leq i \leq length(t) + 1)\ t_i \models \psi$ and $(\forall\ 1 \leq j < i)\ t_j \models \varphi$.

The LTL operators have a slightly different interpretation in the context of finite traces, though similar in spirit to their standard semantics in classical LTL with infinite traces. The formula $\circ\varphi$ (next φ) holds for a finite trace iff the trace is nonempty and φ holds in the suffix trace starting in the next (the second) time point. The formula $\Box\varphi$ (always φ) holds if φ holds in all time points, while $\Diamond\varphi$ (eventually φ) holds if φ holds in present or in some future time point. The formula $\varphi\ \mathcal{U}\ \psi$ (φ until ψ) holds if ψ holds in present or in some future time point, and until then φ holds. As an example illustrating the semantics, the formula $\Box(\varphi \rightarrow \Diamond\psi)$ holds for a finite trace iff for any time point in the trace it holds that if φ is true then eventually ψ is true.

The reader probably noticed that i ranges from 1 to $length(n)$ in the definition of the semantics of $\Box$, while it ranges from 1 to $length(n) + 1$ in the case of $\Diamond$. This discrepancy is not a typo; it is because of the intended semantics of the two operators on the empty trace, that is, $\epsilon \models \Box\varphi$ for any formula φ while $\epsilon \models \Diamond\varphi$ if and only if $\epsilon \models \varphi$. We still don't know exactly if this is the most appropriate semantics of the two operators; it should be taken just as a subjective choice at this incipient stage, but we are certainly going to clarify this issue soon as we get more practical experience with this new technology. However, the algorithms presented in this paper do not essentially depend on this choice.

An important observation which is crucial to the development of the dynamic programming generic algorithms presented later is that the relation $\models$ can be defined recursively in the context of finite traces. We only need to consider the temporal operators:

$$\begin{array}{llll}
\epsilon \models \circ\varphi & \text{is false} & , e\ t \models \circ\varphi & \text{iff } t \models \varphi, \\
\epsilon \models \Box\varphi & \text{is true} & , e\ t \models \Box\varphi & \text{iff } e\ t \models \varphi \text{ and } t \models \Box\varphi, \\
\epsilon \models \Diamond\varphi & \text{iff } \epsilon \models \varphi & , e\ t \models \Diamond\varphi & \text{iff } e\ t \models \varphi \text{ or } t \models \Box\varphi, \\
\epsilon \models \varphi\ \mathcal{U}\ \psi & \text{iff } \epsilon \models \psi & , e\ t \models \varphi\ \mathcal{U}\ \psi & \text{iff } e\ t \models \psi \text{ or } (e\ t \models \varphi \text{ and } t \models \varphi\ \mathcal{U}\ \psi).
\end{array}$$

3 An Example

In this section we show how to generate dynamic programming code for a concrete LTL formula. We think that this example would practically be sufficient for the reader to foresee our general algorithm presented in the next section.

Let $\Box((p\ \mathcal{U}\ q) \rightarrow \Diamond(q\ \rightarrow\ \circ r))$ be an LTL formula and let $\varphi_1, \varphi_2, ..., \varphi_{10}$ be its subformulae, in breadth-first order:

$$\begin{array}{ll}
\varphi_1 & = \Box((p\ \mathcal{U}\ q) \rightarrow \Diamond(q\ \rightarrow\ \circ r)), \\
\varphi_2 & = (p\ \mathcal{U}\ q) \rightarrow \Diamond(q\ \rightarrow\ \circ r), \\
\varphi_3 & = p\ \mathcal{U}\ q, \\
\varphi_4 & = \Diamond(q\ \rightarrow\ \circ r), \\
\varphi_5 & = p, \\
\varphi_6 & = q, \\
\varphi_7 & = q\ \rightarrow\ \circ r, \\
\varphi_8 & = q, \\
\varphi_9 & = \circ r, \\
\varphi_{10} & = r.
\end{array}$$

Given any finite trace $t = e_1 e_2 ... e_n$ of n events, one can recursively define a matrix $s[1..n{+}1, 1..10]$ of boolean values $\{0, 1\}$, with the meaning that $s[i, j] = 1$ iff $t_i \models \varphi_j$ as follows:

$$\begin{array}{ll}
s[i, 10] & = (e_i == r) \\
s[i, 9] & = s[i + 1, 10] \\
s[i, 8] & = (e_i == q) \\
s[i, 7] & = s[i, 8]\ \mathit{implies}\ s[i, 9] \\
s[i, 6] & = (e_i == q) \\
s[i, 5] & = (e_i == p) \\
s[i, 4] & = s[i, 7]\ \mathit{or}\ s[i + 1, 4] \\
s[i, 3] & = s[i, 6]\ \mathit{or}\ (s[i, 5]\ \mathit{and}\ s[i + 1, 3]) \\
s[i, 2] & = s[i, 3]\ \mathit{implies}\ s[i, 4] \\
s[i, 1] & = s[i, 2]\ \mathit{and}\ s[i + 1, 1],
\end{array}$$

for all $i \leq n$, where *and*, *or*, *implies* are ordinary boolean operations and $==$ is the equality predicate, where $s[n + 1, 1..10]$ are defined as below:

$$
\begin{aligned}
s[n+1,10] &= 0\\
s[n+1,9] &= 0\\
s[n+1,8] &= 0\\
s[n+1,7] &= s[n+1,8] \textit{ implies } s[n+1,9]\\
s[n+1,6] &= 0\\
s[n+1,5] &= 0\\
s[n+1,4] &= s[n+1,7]\\
s[n+1,3] &= s[n+1,6]\\
s[n+1,2] &= s[n+1,3] \textit{ implies } s[n+1,4]\\
s[n+1,1] &= 1.
\end{aligned}
$$

An important observation is that, like in many other dynamic programming algorithms, one doesn't have to store all the table $s[1..n+1, 1..10]$, which would be quite large in practice; in this case, one needs only two lines, $s[i, 1..10]$ and $s[i+1, 1..10]$, which we'll write *now* and *next* from now on, respectively. It is now only a simple exercise to write up the following algorithm:

INPUT: trace $t = e_1 e_2 ... e_n$
$next[10] \leftarrow 0$;
$next[9] \leftarrow 0$;
$next[8] \leftarrow 0$;
$next[7] \leftarrow next[8]$ *implies* $next[9]$;
$next[6] \leftarrow 0$;
$next[5] \leftarrow 0$;
$next[4] \leftarrow next[7]$;
$next[3] \leftarrow next[6]$;
$next[2] \leftarrow next[3]$ *implies* $next[4]$;
$next[1] \leftarrow 1$;
for $i = n$ **downto** 1 **do** {
    $now[10] \leftarrow (e_i == r)$;
    $now[9] \leftarrow next[10]$;
    $now[8] \leftarrow (e_i == q)$;
    $now[7] \leftarrow now[8]$ *implies* $now[9]$;
    $now[6] \leftarrow (e_i == q)$;
    $now[5] \leftarrow (e_i == p)$;
    $now[4] \leftarrow now[7]$ *or* $next[4]$;
    $now[3] \leftarrow now[6]$ *or* $(now[5]$ *and* $next[3])$;
    $now[2] \leftarrow now[3]$ *implies* $now[4]$;
    $now[1] \leftarrow now[2]$ *and* $next[1]$;
    $next \leftarrow now$ }
output$(next[1])$;

Given a fixed LTL formula, the analysis of this algorithm is straightforward. Its time complexity is $\Theta(n)$ where n is the length of the input trace, the constant being given by the size of the LTL formula. The memory required is constant, since the length of the two arrays is the size of the LTL formula. However, one may want to also include the size of the formula, say m, into the analysis; then the time complexity is obviously $\Theta(n \cdot m)$ while the memory required is $2 \cdot m$ bits. The authors think that it's hard to find an algorithm running faster than the above in practical situations.

4 The Main Algorithm

We now formally describe our algorithm that synthesizes a dynamic programming algorithm from an LTL formula. Our synthesizer is generic, the potential user being expected to adapt it to his/her desired target language. The algorithm consists of three main steps:

Breadth First Search. The LTL formula should be first visited in breadth-first order to assign increasing numbers to subformulae as they are visited. Let $\varphi_1, \varphi_2, ..., \varphi_m$ be the list of all subformulae in BFS order. Because of the semantics of finite trace LTL, this step insures us that the truth value of $t_i \models \varphi_j$ can be completely determined from the truth values of $t_i \models \varphi_{j'}$ for all $j < j' \leq m$ and the truth values of $t_{i+1} \models \varphi_{j'}$ for all $j \leq j' \leq m$. This recurrence gives the order in which one should generate the code.

Loop Initialization. Before we generate the "for" loop, we should first initialize the vector $\mathit{next}[1..m]$, which basically gives the truth values of the subformulae on the empty trace. According to the semantics of LTL, one should fill the vector *next* backwards. For a given $m \geq j \geq 1$, $\mathit{next}[j]$ is calculated as follows:

- If φ_j is a variable then $\mathit{next}[j] = 0$. Notice that φ_m is always a variable. In a more complex setting of LTL, containing more complex propositions than just propositional variables, one would have to evaluate φ_j in the context of the empty trace or of the final state generated by the trace of events;
- If φ_j is $\neg\varphi_{j'}$ for some $j < j' \leq m$, then $\mathit{next}[j] = \mathit{not}\ \mathit{next}[j']$, where *not* is the negation operation on booleans (bits);
- If φ_j is $\varphi_{j_1}\ \mathit{Op}\ \varphi_{j_2}$ for some $j < j_1, j_2 \leq m$, then $\mathit{next}[j] = \mathit{next}[j_1]\ \mathit{op}\ \mathit{next}[j_2]$, where *Op* is any propositional operation and *op* is its corresponding boolean operation;
- If φ_j is $\circ\varphi_{j'}$ then clearly $\mathit{next}[j] = 0$ according to the semantics of finite trace LTL;
- If φ_j is $\Box\varphi_{j'}$ then $\mathit{next}[j] = 1$ because the empty trace satisfies "always" everything;
- If φ_j is $\Diamond\varphi_{j'}$ then $\mathit{next}[j] = \mathit{next}[j']$ because there are no further events that could make $\varphi_{j'}$ hold in the future: it must hold now;
- If φ_j is $\varphi_{j_1}\ \mathcal{U}\ \varphi_{j_2}$ for some $j < j_1, j_2 \leq m$, then $\mathit{next}[j] = \mathit{next}[j_2]$ for the same reason as above.

Loop Generation. Because of the dependences in the recursive definition of finite trace LTL satisfaction relation, one is expected to visit the trace backwards, so the loop index will vary from n downto 1. The loop body will update/calculate the vector *now* and in the end will move it into the vector *next* to serve as basis for the next iteration. At a certain iteration i, the vector *now* is updated also backwards as follows:

- If φ_j is a variable then *now*$[j]$ only depends on the event e_i. In our simplified version of LTL, *now*$[j]$ is 1 if and only if $e_i = \varphi_j$. In a more complex finite trace LTL where φ_j was a proposition, one would be expected to evaluate φ_j in a state at moment i.
- If φ_j is $\neg\varphi_{j'}$ for some $j < j' \leq m$, then *now*$[j]$ = *not now*$[j']$;
- If φ_j is φ_{j_1} *Op* φ_{j_2} for $j < j_1, j_2 \leq m$, then *now*$[j]$ = *now*$[j_1]$ *op now*$[j_2]$, where *Op* is any propositional operation and *op* is its corresponding boolean operation;
- If φ_j is $\circ\varphi_{j'}$ then *now*$[j]$ = *next*$[j']$ since φ_j holds now if and only if $\varphi_{j'}$ hold at the previous step (which treated the next event, the $i+1$-th);
- If φ_j is $\Box\varphi_{j'}$ then *now*$[j]$ = *now*$[j']$ *and next*$[j]$ because φ_j holds now if and only if $\varphi_{j'}$ holds now and φ_j hold at the previous iteration;
- If φ_j is $\Diamond\varphi_{j'}$ then *now*$[j]$ = *now*$[j']$ *or next*$[j]$ because of similar reasons as above;
- If φ_j is $\varphi_{j_1}\ \mathcal{U}\ \varphi_{j_2}$ for some $j < j_1, j_2 \leq m$, then because of the recursion at the end of Section 2, *now*$[j]$ = *now*$[j_2]$ *or (now*$[j_1]$ *and next*$[j]$).

After each iteration i, *next*$[1]$ tells whether the initial LTL formula is validated by the trace $e_i e_{i+1} \ldots e_n$. Therefore, the desired output is *next*$[1]$ after the last iteration. Putting all the above together, one can now write up the generic pseudocode presented in the appendix which can be implemented very efficiently on any current platform. Since the BFS procedure is linear, the algorithm synthesizes a dynamic programming algorithm from an LTL formula in linear time with the size of the formula.

References

1. Kuck & Associates. Assure, 2000. `http://www.kai.com/parallel/assuret`.
2. Manuel Clavel, Francisco J. Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and Programming in Rewriting Logic, March 1999. Maude System documentation at `http://maude.csl.sri.com/papers`.
3. James Corbett, Matthew B. Dwyer, John Hatcliff, Corina S. Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, Limerich, Ireland, June 2000. ACM Press.
4. Claudio Demartini, Radu Iosif, and Riccardo Sisto. A Deadlock Detection Tool for Concurrent Java Programs. *Software Practice and Experience*, 29(7):577–603, July 1999.
5. Doron Drusinsky. The Temporal Rover and the ATG Rover. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.
6. Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the 15th Workshop on Protocol Specification, Testing, and Verification*. North-Holland, 1995.

7. Jerry Harrow. Runtime Checking of Multithreaded Applications with Visual Threads. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 331–342. Springer, 2000. http://www5.compaq.com/products/software/visualthreads.
8. Klaus Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 245–264. Springer, 2000.
9. Klaus Havelund, Michael R. Lowry, and John Penix. Formal Analysis of a Space Craft Controller using SPIN. In *Proceedings of the 4th SPIN workshop*, Paris, France, November 1998. To appear in IEEE Transactions of Software Engineering.
10. Klaus Havelund and Thomas Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, April 2000. Special issue of STTT containing selected submissions to the 4th SPIN workshop, Paris, France, 1998.
11. Klaus Havelund and Grigore Roşu. Testing linear temporal logic formulae on finite execution traces, 2000. Submitted for publication. http://ase.arc.nasa.gov/people/grosu.
12. Klaus Havelund and Natarajan Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In Marie Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681. Springer, 1996.
13. Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
14. Gerard J. Holzmann and Margaret H. Smith. A Practical Method for Verifying Event-Driven Software. In *Proceedings of ICSE'99, International Conference on Software Engineering*, Los Angeles, California, USA, May 1999. IEEE/ACM.
15. Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime Assurance Based on Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
16. David Y.W. Park, Urlich Stern, and David L. Dill. Java Model Checking. In *Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification, Limerick, Ireland*, June 2000.
17. Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
18. Stefan Savage, Michael Burrows, Greg Nelson, Patrik Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.
19. Sitraka Software. Jprobe, 2000. http://www.sitraka.com/software/jprobe.
20. Scott D. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 224–244. Springer, 2000.
21. Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model Checking Programs. In *Proceedings of ASE'2000: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, September 2000.

A Generic Pseudocode for the Synthesizer

The following generic program implements the technique discussed in the paper. It takes as input an LTL formula and generates a for loop which traverses the trace of events backwards, thus validating or invalidating the formula.

INPUT: LTL formula φ
output("INPUT: trace $t = e_1 e_2 ... e_n$");
let $\varphi_1, \varphi_2, ..., \varphi_m$ be all the subformulae of φ is BFS order
for $j = m$ **downto** 1 **do** **{**
  output("*next*[", j, "] $\leftarrow$ ");
  if φ_j is a variable **then output**("0;");
  if $\varphi_j = \neg\varphi_{j'}$ **then output**("*not next*[",j', "];");
  if $\varphi_j = \varphi_{j_1}\ Op\ \varphi_{j_2}$ **then output**("*next*[",j_1, "] *op next*[", j_2, "];");
  if $\varphi_j = \circ\varphi_{j'}$ **then output**("0;");
  if $\varphi_j = \Box\varphi_{j'}$ **then output**("1;");
  if $\varphi_j = \Diamond\varphi_{j'}$ **then output**("*next*[",j', "];");
  if $\varphi_j = \varphi_{j_1}\ \mathcal{U}\ \varphi_{j_2}$ **then output**("*next*[",j_2, "];"); **}**
output("**for** $i = n$ **downto** 1 **do {**");
for $j = m$ **downto** 1 **do** **{**
  output("   *now*[", j, "] $\leftarrow$ ");
  if φ_j is a variable **then output**("(e_i ==", φ_j, ");");
  if $\varphi_j = \neg\varphi_{j'}$ **then output**("*not now*[",j', "];");
  if $\varphi_j = \varphi_{j_1}\ Op\ \varphi_{j_2}$ **then output**("*now*[",j_1, "] *op now*[", j_2, "];");
  if $\varphi_j = \circ\varphi_{j'}$ **then output**("*next*[", j', "];");
  if $\varphi_j = \Box\varphi_{j'}$ **then output**("*now*[", j', "] *and next*[", j, "];");
  if $\varphi_j = \Diamond\varphi_{j'}$ **then output**("*now*[", j', "] *or next*[", j, "];");
  if $\varphi_j = \varphi_{j_1}\ \mathcal{U}\ \varphi_{j_2}$ **then output**("*now*[", j_2, "] *or (now*[", j_1, "] *and next*[", j, "]);"); **}**
output("   *next* $\leftarrow$ *now*; **}**");
output("**output** *next*[1];");

where *Op* is any propositional connective and *op* is its corresponding boolean operator.

The boolean operations used above are usually very efficiently implemented on any microprocessor and the vectors of bits *next* and *now* are small enough to be kept in cache. Moreover, the dependencies between instructions in the generated "for" loop are simple to analyze, so a reasonable compiler can easily unfold or/and parallelize it to take advantage of machine's resources. Consequently, the generated code is expected to run very fast.