# Automata-Based Verification of Temporal Properties on Running Programs

**Dimitra Giannakopoulou**

**Klaus Havelund**

**RIACS Technical Report 01.21**

**August 2001**

**Presented at the 16th IEEE International Conference on Automated Software Engineering, San Diego, 2001**

# Automata-Based Verification of Temporal Properties on Running Programs

**Dimitra Giannakopoulou, RIACS**

**Klaus Havelund, Kestrel Technologies**

**RIACS Technical Report 01.21**

**August 2001**

**Presented at the 16th IEEE International Conference on Automated Software Engineering, San Diego, 2001**

This paper presents an approach to checking a running program against its Linear Temporal Logic (LTL) specifications. LTL is a widely used logic for expressing properties of programs viewed as sets of executions. Our approach consists of translating LTL formulae to finite-state automata, which are used as observers of the program behavior. The translation algorithm we propose modifies standard LTL to Büchi automata conversion techniques to generate automata that check finite program traces. The algorithm has been implemented in a tool, which has been integrated with the generic JPaX framework for runtime analysis of Java programs.

This work was supported in part by the National Aeronautics and Space Administration under Cooperative Agreement NCC 2-1006 with the Universities Space Research Association (USRA).

This report is available online at http://www.riacs.edu/trs/

# Automata-Based Verification of Temporal Properties on Running Programs

## 1 Introduction

Computer program correctness has, for decades, concerned industry and been studied in academia. The formal methods research community, in particular, has studied how formal logic can be applied in the analysis of computer programs and their designs. Most effort has been spent on the development of semantic frameworks for formalizing logics and developing systems that can formally prove that a software artifact satisfies a formula in some logic. Model checking and theorem proving are examples of such attempts to mechanize proofs of correctness. Such *heavy-weight* formal methods, however, still remain to reach a state where they can be used in practice without considerable manual effort. A recent research direction is to apply model checking directly to programs written in standard programming languages such as Java and C [1, 2].

Although we find this work of great interest, and in fact have been involved in this from early on in this recent trend [1, 3], we believe that more *light-weight* use of formal techniques will be useful as well as more practically feasible in the shorter term. A light-weight formal method is here defined as a method that is completely automatic, irrespective of the size of the examined program. Hence, the main concern is scalability: the technique should be practically applicable to large systems consisting of hundreds of thousands of lines of code.

An example of such a light-weight technique is what is often referred to as program monitoring. Here, the idea is to monitor the execution of a program against a formal specification written in some formal specification logic. This kind of technique is practically feasible since only one trace is examined, and it is useful since the logic allows stating more complex properties than is normally possible in standard testing environments.

In this paper, we describe an effort to develop such a technique for monitoring program executions against high-level requirement specifications written in Linear Temporal Logic (LTL). LTL has mostly been used in the past as the logic in model checking environments, such as SPIN [4]. Such environments typically convert (automatically) the negation of an LTL requirement into a Büchi automaton that accepts all infinite words that violate this requirement (Büchi automata are finite automata on infinite words). Model checking consists of detecting accepting cycles in the synchronous product of the model/program to be checked with the automaton. The state-space of this product typically needs to be stored, which gives rise to the infamous state-explosion problem [5].

Büchi automata are designed to operate on infinite execution traces. So the question naturally arises whether Büchi automata can also be used to efficiently monitor finite traces of executing programs (since, either the program will terminate, or it will be interrupted at some stage). A typical way to check a finite trace with Büchi automata is to extend the trace by repeating the last state indefinitely. This approach would still require storing the combined state-space of the automaton and the particular program trace, in order to check for cycles.

The work presented in this paper is the result of trying to provide a more efficient alternative based on the same (and extensively polished over years of research) principles. It presents an algorithm, based on standard LTL to Büchi automata construction techniques, which generates traditional finite-state automata that can be used to monitor LTL formulae on finite program traces.

The algorithm we present has been implemented in Trace analyZer (TaZ), an observer generator tool written in Java. TaZ has been integrated in Java PathExplorer (JPaX), which is a generic tool for monitoring Java programs [6]. The result is an environment that can automatically check, on-the-fly, whether the current run of a Java program conforms to an LTL formula.

The remainder of this paper is organized as follows. Section 2 discusses background and related work, and introduces the JPaX runtime analysis tool – the context of the work presented here – under development at NASA Ames. Section 3 sets the theoretical grounds for Section 4, which presents our algorithm for generating LTL runtime observers. Section 5 proposes optimizations to the basic algorithm. We discuss the implementation of TaZ and some performance issues in Section 6. Finally, Section 7 closes the paper with discussion and conclusions.

## 2 Program monitoring

The algorithm that we are presenting in this paper has been implemented in the internally developed runtime-monitoring tool Java PathExplorer (JPaX). JPaX is a general environment for monitoring the execution of Java programs. It consists of an instrumentation module, and an observer module. The instrumentation module performs a script-driven automated instrumentation of the

program to be observed. The instrumented program, when run, will emit relevant events to the observer. The observer may run on a different computer, in which case the events are transmitted over a socket.

The instrumentation is performed on the basis of an instrumentation script, given by the user that specifies which variables in the program shall be monitored. The automated instrumentation will then insert event-transmitting code after all updates to these variables. The updates are collected in an image state separate from the executing program state. The instrumentation script also defines a collection of Boolean valued proposition variables and an association between these and predicates over the observed program variables. When an image state change occurs, the propositional variables are re-evaluated, and what is sent to the observer is changes in these propositional variables. The Java byte code instrumentation is performed using the powerful Jtrek Java byte code-engineering tool [7] from Compaq. Jtrek makes it possible to easily read Java class files (byte code files), and traverse them as abstract syntax trees while examining their contents, and inserting new code.

Program monitoring against specifications expressed in various logics has been investigated by several researchers. In [8], for example, the authors describe an algorithm for generating test oracles from specifications written in GIL, a graphical interval logic. Similarly to our approach, the oracles are based on automata. The generation is performed in two phases. During the first phase, a hierarchical non-deterministic automaton is computed, which, during the second phase, is turned into a classical deterministic finite automaton. The authors do not mention the application of minimization techniques to the resulting automata. The automata that they generate are typically larger than the ones that our algorithm computes. The reason is that they do not attempt to collapse equivalent states during generation.

An approach based on rewriting logic is presented in [9]. The authors have implemented in Maude [10] (an efficient rewriting logic system), 8 rules that describe how an LTL formula is transformed by a new state encountered in the program, and how to decide, when the end of a trace occurs, whether the specification was satisfied or not. Naturally, their algorithm also detects if a property is satisfied or violated before the end of the trace occurs, in which case the analysis does not need to proceed. Maude is, in general, a very powerful prototyping tool, since it allows to easily define alternative types of logics. The authors have used it to also support past-time logics, for example [6].

The Temporal Rover (TR) [11] is a commercial tool for program monitoring based on temporal logic specifications. TR allows users to specify future time temporal formulae as comments in programs, which are then translated into appropriate Java code before compilation.

# 3 Preliminaries

This section describes the syntax of both the infinite-(standard) and finite-trace semantics of Linear Temporal Logic (LTL). It also defines finite-state automata as used for runtime verification.

Note that, in the context of this paper, we are only interested in the next-free variant of LTL, namely LTL-*X*. This is typical in model checking, because LTL-*X* is guaranteed to be insensitive to stuttering [5]. This property is important because it avoids the notion of an absolute next state. The next time operator (*X*) is misleading, because users naturally tend to assume some level of abstraction on the state of a running program. Additionally, it is not straightforward what the desired meaning of a next time formula would be at the last state of a program trace.

In the rest of this paper, the next-free variant of LTL is implied whenever we refer to LTL.

## 3.1 LTL – standard semantics

The set of well-formed LTL formulae is constructed from a set of atomic propositions, the standard Boolean operators, and the temporal operator *U*. Given a finite set of atomic propositions $\wp$, formulas are constructed inductively as follows:

PROPOSITIONS: Every $\varphi$ where ($\varphi \in \wp$) is a formula.

BOOLEAN OPERATORS: If $\varphi$ and $\psi$ are formulas, then so are $!\varphi$ (logical not), $\varphi\wedge\psi$ (logical and), $\varphi\vee\psi$ (logical or). Also, $\varphi\rightarrow\psi$ (logical implication) is an abbreviation for $!\varphi\vee\psi$, TRUE for $\varphi\vee!\varphi$, and FALSE for !TRUE.

TEMPORAL OPERATORS: If $\varphi$ and $\psi$ are formulas, then so is $\varphi U\psi$ (strong until). The following abbreviations are used: $<>\varphi$ (eventually) for TRUE$U\varphi$, and $[]\varphi$ (always) for $!<>!\varphi$. Finally, we also use the temporal operator *V* which is defined as the dual of *U*, i.e.: $\varphi V\psi = !(!\varphi U\, !\psi)$.

An interpretation of an LTL formula is an infinite word $w=x_0x_1\ldots$ over $2^{\wp}$ (sets of propositions), where at some time point $i\in N$, a proposition $p$ is true iff (if and only if) $p\in x_i$. We write $w_i$ for the suffix of $w$ starting at $i$. The semantics of LTL is defined as follows:

PROPOSITIONS: For $\varphi\in\wp$, $w \models \varphi$ iff $\varphi\in x_0$.

BOOLEAN OPERATORS:

- $w \models !\varphi$ iff not $w \models \varphi$;
- $w \models \varphi\wedge\psi$ iff $w \models \varphi$ and $w \models \psi$;
- $w \models \varphi\vee\psi$ iff $w \models \varphi$ or $w \models \psi$.

TEMPORAL OPERATORS:

- $w \models \varphi U \psi$ iff there exists $i \in N$ such that $w_i \models \psi$ and for all $0 \leq j < i$, $w_j \models \varphi$.

### 3.2 LTL – finite-trace semantics

An executing program defines a sequence of states; an infinite execution can therefore be viewed as an LTL interpretation, which assigns to each moment in time the set of propositions that are true at the particular program state. Model checking [5] detects infinite executions of finite-state systems through cycles in their state graphs. Runtime verification does not store the entire state-space of a program. Rather, it only observes finite program executions, on which we also need to interpret LTL formulae. We only need to modify the semantics of the *temporal* operators to accommodate this difference.

Every LTL formula may contain either a safety part, or an eventuality part (or both). The safety/eventuality part requires that something bad-never/good-eventually happens in an execution. We modify the safety requirement to mean that, *in the portion of the execution that we have observed*, nothing bad happens. Eventualities are similarly required to be satisfied *in the portion of the execution observed*. Otherwise they will have "not yet" been satisfied. We define the semantics of the temporal operators accordingly.

Let $w=x_0...x_n$ be a finite interpretation of an LTL formula over $2^{\wp}$. Then:

<u>Temporal operators:</u>

- $w \models \varphi U \psi$ iff there exists $0 \leq i \leq n$ such that $w_i \models \psi$ and for all $0 \leq j < i$, $w_j \models \varphi$.

### 3.3 Finite automata on finite words

A finite automaton *FA* is a 5-tuple $(S, A, \Delta, s_0, F)$, where $S$ is a finite set of states, $A$ is a finite set of labels which we call the alphabet of the automaton, $\Delta \subseteq S \times A \times S$ is a transition function, $s_0 \in S$ is the initial state, and $F \subseteq S$ is a set of accepting states.

An execution of *FA* is a finite sequence $\sigma = s_0\, a_0\, s_1 \ldots a_{n-1}\, s_n$, such that $(s_i\ a_i\ s_{i+1}) \in \Delta$ for each $0 \leq i < n$. An execution $\sigma$ of *FA* is accepting if $s_n \in F$. Finally, *FA* accepts a finite word $w=x_0...x_{n-1}$ over $A$, if there exists an accepting execution $\sigma = s_0\, x_0\, s_1 \ldots x_{n-1}\, s_n$ of *FA*.

In the context of this paper, automata labels will be sequences of Boolean values of propositions. A label *<false, false, true>* (or 001) for sequence *<a, b, c>* will represent, for example, that *a* and *b* are *false*, and *c* is *true*. Equivalently, we will sometimes write *!a!bc*. Finally, for an automaton with propositions *<a, b, c>*, a transition between $s_1$ and $s_2$ labeled with *a!b* is an abbreviation for two transitions between $s_1$ and $s_2$, one labeled with *a!bc*, and one labeled with *a!b!c*. In other words, the transition can be fired when *a* is *true* and *b* is *false*, irrespective of the value of *c*.

## 4 Algorithm

Our goal is to construct a finite-state automaton that accepts exactly those finite words that satisfy a given LTL formula φ. Our algorithm is based on an efficient tableau-like LTL to Büchi automata translation presented in [12]. We will briefly describe the intuition behind this construction, and then explain the modifications that our algorithm introduces in order to capture the finite-trace LTL semantics introduced in the previous section.

### 4.1 LTL to Büchi automata

The LTL formulas that we deal with are in negation normal form. This simply means that all negations are pushed inside until they precede only propositional variables. For example, the negation normal form of formula ![]φ is formula <>!φ.

The core of the algorithm is based on *expanding* a *graph node*. A graph node is a data structure that contains the following fields:

NAME: a unique name for the node.

INCOMING: the set of nodes that lead to this node, i.e., which have incoming edges to this node.

NEW: the set of LTL formulae that must hold on the current state but have not yet been processed.

OLD: the set of LTL formulae that have already been processed. Each formula in NEW that gets processed is transferred to OLD.

NEXT: the set of LTL formulae that must hold at all immediate successors of this node.

**Table 1: Formula expansions used in Node splitting**

| $f$ | NEW1($f$) | NEXT1 ($f$) | NEW2 ($f$) |
|---|---|---|---|
| $\varphi\ U\ \psi$ | $\{\varphi\}$ | $\{\varphi\ U\ \psi\}$ | $\{\psi\}$ |
| $\varphi\ V\ \psi$ | $\{\psi\}$ | $\{\varphi\ V\ \psi\}$ | $\{\varphi, \psi\}$ |
| $\varphi \vee \psi$ | $\{\varphi\}$ | $\varnothing$ | $\{\psi\}$ |

The automaton states that we construct are stored in STATES, which is a set of graph nodes.

Field NEW in a graph node represents all the formulae that the node must make true. The idea of the expansion algorithm is to remove formulae in NEW one by one by processing them in the following way. Each formula is broken down until we get to the literals (propositions or negated propositions) that must hold to make it true. For example, $\varphi \wedge \psi$ is broken down by adding both φ and ψ to the NEW field of the node. If there are alternative ways to make a formula true (if it is an $\vee$ or $U$ formula, for

example) the node is split in two nodes, where each of these nodes represents one way of making the formula true. For example, to make $\varphi \vee \psi$ true, we split the node into a node that needs to make $\varphi$ true, and one that needs to make $\varphi$ true.

To satisfy temporal operator formulae, a node needs to also push obligations to its immediate successors. These obligations are added to the NEXT field of a node. The way obligations are moved to successors is based on the following identities:

- $\varphi U \psi \equiv \psi \vee (\varphi \wedge X(\varphi U \psi))$
- $\varphi V \psi \equiv \psi \wedge (\varphi \vee X(\varphi V \psi))$

Table 1 illustrates, for the types of formulas *f* that cause a node to split, the formulas that are added to various fields of the resulting nodes. The two resulting nodes contain the same INCOMING and OLD fields as the original node, whereas their NEW and NEXT fields are the union of the ones of the original node with the fields illustrated in Table 1 (where 1 and 2 refer to the two resulting nodes).

Note that, the old field of a node cannot contain contradicting formulae (e.g. both $\varphi$ and $!\varphi$). If contradictions are obtained during processing, then the node is discarded.

The algorithm starts with a node, which contains the LTL formula *f* (for which an automaton is being build), in its NEW field, *INIT* in its INCOMING field, and all other fields are empty. *INIT* represents the initial node, a node that has all its fields empty, and which represents the initial state of the automaton. It is the only node that is initially in STATES.

When all formulae of a node $N_C$ have been processed (i.e., the NEW field becomes empty), the node represents a node of the automaton. Before, however, we add $N_C$ to STATES, we check whether an equivalent node is already contained there. An equivalent node is one that has the same OLD and NEXT fields as the one that has just been processed. If an equivalent node *N* exists in STATES, then the INCOMING field of $N_C$ is added (by set union) to the INCOMING field of *N*. If no equivalent node exists in states, then $N_C$ is added to STATES, and a new node $N_N$ is added to the ones that need to be processed. $N_N$ represents the immediate successors of $N_C$. Its INCOMING field is set to $N_C$, and its NEW field is set to the NEXT field of $N_C$. All other fields are initially empty.

When this process is completed, an automaton can be built from the nodes in STATES in the following fashion. Each node represents a state of the automaton. Edges are defined by the INCOMING fields of the nodes. The initial node is the one that has no incoming edges (*INIT*). All edges that lead into a node *N*, are labeled with the literals that must hold at *N*, i.e. the literals that belong to the OLD field of *N*. We do not discuss accepting states here, because our algorithm assigns these in a completely different fashion.

## 4.2 Applying finite-trace semantics

The main aspect of the LTL to Büchi automata construction algorithm that we modify is the selection of accepting conditions. Any infinite execution of an automaton generated for a formula *f* as described in Section 4.1 satisfies the safety conditions of *f*. As mentioned in [12], this is guaranteed by construction of the automaton. Accepting conditions have then got to be imposed, to make sure that eventualities are also satisfied. More precisely, we need to make sure that whenever some node contains $\varphi U \psi$, some successor node will contain $\psi$.

Given the modified semantics presented in Section 3.2, we similarly need to impose accepting conditions to make sure that any accepting finite execution of the automaton (since we deal with finite traces) satisfies all the required eventualities. The eventualities that remain to be satisfied after any finite execution of the automata we generate, are reflected by the formulae contained in the NEXT field of the last state of this execution. This means that, unless there exist *U* formulae in the next field of the state, then this state has satisfied its potential eventuality requirements (as can be seen in Table 1, if the right-hand formula of an *U* operator is added to the NEW field of a state, then no formula is added to its NEXT field).

The initial state is non-accepting. Therefore, our construction assumes that automata will only accept traces that contain at least one state (since the initial state is non-accepting). In the optimizations section (Section 5) we discuss how to raise this assumption in order to simplify the automata we obtain, in some cases.

It can be seen from our construction, that field OLD is used by our algorithm only to generate the labels of the automaton, but plays no role in the identification of accepting conditions. Therefore, two states are basically equivalent when they have the same NEXT fields. Although the original construction would still generate correct automata, we can make our construction more efficient based on this observation.

To allow for more equivalent states to be collapsed during construction, when the algorithm compares a newly expanded node to the nodes that have already been entered in STATES, we only compare the NEXT fields of the nodes. This, of course, does not make the OLD field redundant; the literals contained in it are needed in order to determine labels of the automata edges. Therefore, we decide to store in old only literals, during construction.

One needs to be careful during the process of collapsing equivalent nodes. If two nodes are collapsed, the resulting node must record the information of each component's INCOMING field and its associated OLD field.

This is for it to remember that it is obtained from alternative parent nodes by different sets of literals. In the simple case where the literals in the OLD fields are the same, the OLD field of the resulting node is the same as the corresponding field of either of its components, and its INCOMING field is obtained as the union of their corresponding INCOMING fields.

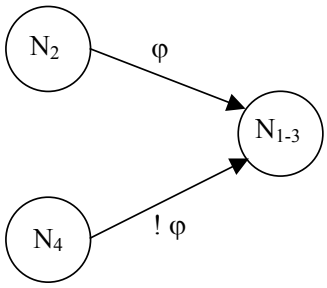


**Figure 1: Collapsing nodes with different OLD fields**

For example, assume that a node $N_1$ with OLD={φ} (where φ is a proposition) and INCOMING={$N_2$}, is collapsed with node $N_3$ with OLD={!φ}, and INCOMING={$N_4$}. Let us call the resulting node $N_{1\text{-}3}$. This information is kept appropriately in node $N_{1\text{-}3}$, so that in the generated automaton, it will look as in Figure 1.

### 4.3 Proof of correctness

We briefly sketch the proof of correctness of the algorithm presented here. Our proof is based on the corresponding proof of the [12] algorithm, so the interested reader is referred to that paper for details.

What we need to show additionally is that indeed, our accepting conditions guarantee that a finite sequence (of length ≥1) is accepted iff it satisfies the LTL formula for which the automaton was built. Intuitively, the construction that we follow ensures that. The reason is that, when a node is processed, all the possibilities of making its requirements true are examined.

As far as eventualities are concerned (derived from *U* formulae) our construction ensures that any *U* formula that must be satisfied remains in the next field of the corresponding node and all of its successors, until its right-hand side formula becomes true. Therefore, the existence of a *U* formula in the next field of a node reflects the fact that, in the path followed to this node, there are eventualities that remain to be checked.

## 5 Optimizations

DETERMINISTIC MINIMAL AUTOMATA. The automata we generate are finite automata on finite words. We can therefore, with standard algorithms, both make them deterministic, and minimize them. A finite automaton on finite words can be made deterministic by using the subset construction [13]. The algorithm is theoretically exponential in the number of states of the automaton, but works well in practice for the sizes of automata typically needed for verification. Efficient ($O(n\ log(n))$, where $n$ is the number of states in the automaton) minimization algorithms also exist for finite automata [14, 15].

Before we apply these algorithms to the automata we generate, we need to make their labels typical of such automata. We perform this by applying the following transformation to the labels of the automata we generate.

Assume that $A = \{a_1 ... a_n\}$ is the set of propositions in the alphabet of an automaton. Then the labels of the transformed automaton will be arrays of length $n$, where position $i$ contains the value that $a_i$ needs to have to make a transition. We therefore transform the transitions in our original automata into transitions labeled with such arrays.

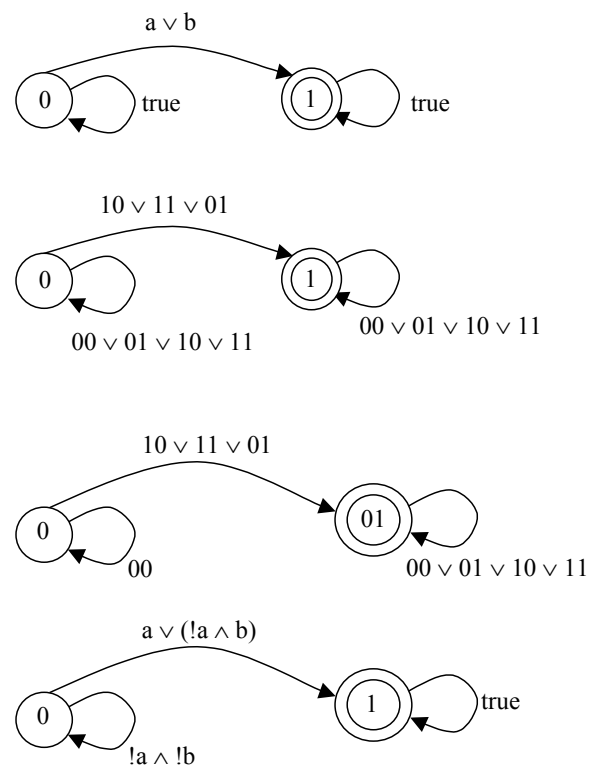


**Figure 2: Creating deterministic, minimal automata**

We illustrate this procedure with an example. The automaton generated by our algorithm for LTL formula $\Diamond(a \vee b)$ is the first automaton illustrated in Figure 2. By transforming the labels of that automaton as described above, we obtain the second automaton illustrated in the same figure. For example, label $a \vee b$ (this is how we represent two transitions from state 0 to state 1, one labeled with $a$, and one labeled with $b$) is transformed

into label *10∨11∨01*, where the first/second number of each sub-label represents the value that *a*/*b* must have for this transition to be triggered. The third automaton results from applying subset construction to the second one. Minimization of this automaton does not result in fewer states.

The last automaton returns the labels to their original form. In performing this, optimizations that concern the simplification of edge terms in Büchi automata [16] can be applied. Formulae on label edges can thus be simplified based on propositional logic rules that are standard; we will therefore not elaborate further on those.

TRUE LOOP. A true loop around an accepting state means that, as soon as a prefix of any trace reaches that state, the property is satisfied by that trace. Any outgoing edges from such a state can therefore be removed. Moreover, this state can bear a label that indicates the fact that, when it is reached, there is no need for further exploration.

INITIAL STATE. Our approach to dealing with the initial state of the automata we generate reflects the fact that we do not design our automata to deal with empty traces. Although this assumption makes sense, we have ways of raising it in some cases. The solutions we propose here are only partial, that is, they deal with empty traces only in specific cases. In all other cases, they simply do not accept empty traces. This is obviously not a problem in practice, since we do not expect users to wish to test empty traces of their programs.

If the formula for which we generate an automaton contains no *U* sub-formulas, then the initial state of the automaton is set to accepting. This expresses the fact that, purely safety properties are trivially satisfied by empty traces (along the lines of the fact that any program that does nothing is safe…).

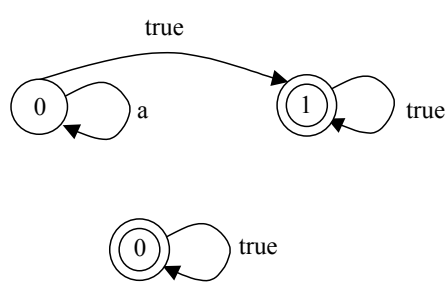


**Figure 3: Dealing with the initial state**

Similarly, if an accepting state is reachable from the initial state with a *true* transition, then we can safely set the initial state to accepting. This is best applied after the "determinization" phase of the construction, and before minimization (this may increase the reduction achieved). For example, the first automaton illustrated in Figure 3 represents formula (*aUtrue*). The initial state is non-accepting, but, through the procedure just described, can be set to accepting. The deterministic minimal automaton thus obtained is depicted by the second automaton illustrated in the same figure.

## 6 Implementation

We have developed a tool, the trace analyser (TaZ), which receives as input an LTL formula, and generates an observer for traces of running programs, using the algorithm presented.

### 6.1 Automata generation

TaZ does not yet implement the algorithms for making the automata generated deterministic and minimal. These features will be implemented in the near future. In addition to generating observers, TaZ currently outputs the corresponding automata in FSP, the input language of the LTSA model-checking tool [17], in order to allow their graphical illustration. LTSA also supports determinization and minimization, which we can apply to the automata generated for experimental purposes.

In TaZ, any string can be used to represent a proposition, and the operators are entered as follows: [], <>, *U*, *V*, !, ∧, ∨, ->. As an example, assume that TaZ is given as input the LTL formula [](*a*→ <>*b*). The FSP output it produces is the following, and the automaton depicted by the LTSA tool is as illustrated in Figure 4 (note that the LTSA tool always names states with integers):

```
RES = S0,
S0=(true->S2 |b->S7 |na -> S7),
S2=(true->S2|na->S2 |b->S7 |b_AND_na->S7),
S7 @ =(true -> S2|b -> S7 |na -> S7).
```

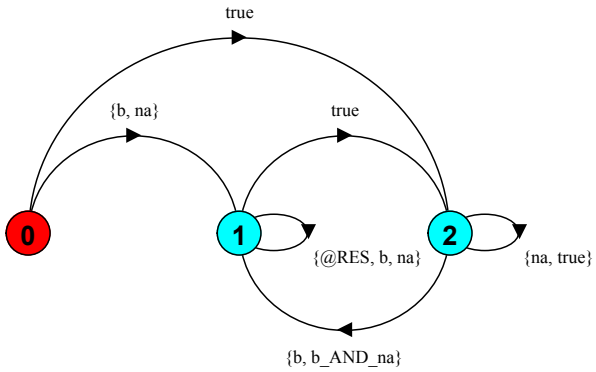


**Figure 4: Automaton for property [](a → <>b)**

The "@" character is used to denote the accepting state in FSP, which is converted by the tool in a looping

transition labeled with "@automaton_name" [18] (so state 1 is the accepting state in the automaton of Figure 4). Moreover, $\{a, b\}$ is used to denote "$a \vee b$". We use the feature of the tool that supports labeling of transitions with sets to redefine transition labels in order to make the automaton deterministic. The deterministic version obtained is illustrated in Figure 5.

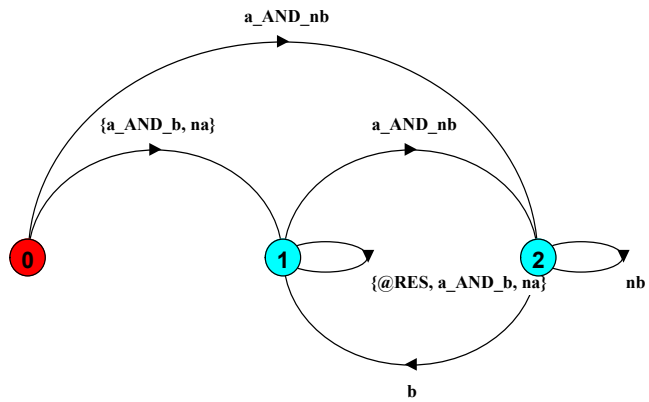


**Figure 5: Deterministic automaton for [](a→ <>b)**

We have used TaZ to generate automata for large formulae (more than 20 operators, mostly *U, V* and \/s, which cause nodes to split), and it produces results instantaneously.

## 6.2 Using the automata for runtime analysis

TaZ turns any automaton that it generates into an observer of program traces. An observer is a data structure that consists of the following fields:

- The automaton for the formula to be checked.
- The current states of the automaton. These may be multiple if the automaton is non-deterministic (this is the case with our current implementation). Initially, the automaton is in its initial state.
- A hash-table that records the values, at the program state that is being verified, of the propositions involved in the formula;

Our observer class implements the following interface, required by PaX:

```
interface LTL{
  void init(STATEINIT init);
  void next(STATECHANGE change);
  void end();
}
```

Method *init* is called by JPaX to pass the observer the values of propositions at the initial program state. Then, each time the proposition values change, JPaX calls the *next* method of the observer to pass it information about the state change. This is provided as a list of propositions that have changed value since the previous state.

Every time *next* is called, the observer performs the following steps. It updates the values of propositions in its local hash-table of the program state. It then checks which transitions rooting at its current states are enabled. To do this, it checks if the state of the program is compatible with the literals labelling these transitions. If, for example, *!a* labels a transition *trans*, and *a* is false in the current program state, then *trans* is enabled. The current states of the automaton are then updated to be the set of states that are reached through enabled transitions. If this set is empty, it means that the automaton cannot make a step, which reflects the fact that the property is violated by the specific trace of the program. This information is reported, which concludes the observer's job.

When the program is stopped, and if the observer is still running (i.e. it did not yet detect a violation or the fact that the property is satisfied), the program calls the *end* method of the observer. At this stage, the observer checks its set of current states. If there exists/does not exist at least one accepting state within this set, then the observer reports the fact that the property is satisfied/violated by the specific program trace, respectively.

**Stuttering:** As mentioned, the LTL-X variant of LTL is insensitive to stuttering. Therefore, the observer only needs to be notified whenever propositions in its alphabet change value. This can be implemented by the observer initially informing JPaX about the particular state attributes it is interested in observing.

**Experimental results:** Checking program traces with the observers we generate is very efficient; it is linear in the program trace.

We have applied our tools to artificially generated traces for early testing purposes. For properties that require checking the entire trace before a result is produced (e.g. <>[]φ), it takes our approach less than 5 minutes on a Pentium 4, 1.3 GHz processor, to process a trace 100 Million state changes long. We expect that when we produce deterministic and minimal automata, this performance will be further improved.

Another issue that we are interested in is ways of minimizing the effort required to compute enabled transitions. When observers are based on deterministic automata, an obvious improvement would be that, when an enabled transition is discovered, other possibilities do not need to be checked (since a single transition can be enabled at a time). Another optimization example would be the following. When the proposition values that two transitions depend on overlap, we should need to check those only once.

An exponential but useful in practice algorithm has been developed, which is briefly discussed in [6], but will be fully documented in the near future.

## 7 Conclusions

We presented an approach to generate deterministic and minimal finite-state automata used to check running programs against LTL specifications. The core of the algorithm modifies standard LTL to Büchi automata construction techniques. These techniques have been polished for efficiency over years of research. It has therefore been important for us to use these as a foundation. Moreover, we have been able to exploit standard algorithms for determinization and minimization of the automata we generate.

This approach is clearly more efficient than using Büchi automata for the same purpose. A benefit of our approach is that it does not require the detection of cycles in the product of the automaton with the program trace. Rather, all that is needed in terms of storage is the current state of the program, and the current state of the automaton. There are, therefore, no scalability issues involved. Additionally, we are able to generate minimal deterministic automata. Büchi automata provide full expressiveness only when they are non-deterministic. Moreover, finding the optimal (or approximately optimal) sized automaton for an LTL formula is PSPACE-hard [16].

An issue that occurs is whether LTL is the most appropriate language for expressing properties of running programs. LTL is a logic that has been widely used for expressing properties of reactive systems. This is particularly so in the domain of model checking. We believe that runtime monitoring and model checking will form components of extended debugging environments. It is therefore crucial to allow users to specify properties that are supported by both approaches.

From our experiments, the generation of observers is very efficient. So is their behavior during runtime analysis; specifications are checked in time linear in the length of the program trace that is examined. The core of our future research will therefore concentrate on how to improve the interaction of the running program with the observer so as to allow maximal independence between the two, but minimal disruption to the running program.

Another topic that this research will involve is the specification of the relationship between program attributes and propositions involved in the formulae, and the instrumentation of a program to emit the relevant information for use by its observers.

## 8 References


[1] Visser, W., Havelund, K., Brat, G., and Park, S. "Model Checking Programs", in *Proc. of the 15th IEEE International Conference on Automated Software Engineering (ASE'2000)*. 11-15 September 2000, Grenoble, France. IEEE Computer Society, pp. 3-11. Y. Ledru, P. Alexander, and P. Flener, Eds.

[2] Holzmann, G.J. and Smith, M.H., *Software model checking - Extracting verification models from source code.* Formal Methods for Protocol Engineering and Distributed Systems, Kluwer Academic Publishers, October 1999: pp. 481-497.

[3] Havelund, K. and Pressburger, T., *Model Checking Java Programs Using Java PathFinder.* International Journal on Software Tools for Technology Transfer (STTT), Vol. **2**(4), April 2000.

[4] Holzmann, G.J., *The Model Checker SPIN.* IEEE Transactions on Software Engineering, Vol. **23**(5), May 1997: pp. 279-295.

[5] Clarke, E.M., Grumberg, O., and Peled, S.A., *Model Checking*: The MIT press, 1999.

[6] Havelund, K. and Rosu, G. "Monitoring Java Programs with Java PathExplorer", in *Proc. of the First Workshop on Runtime Verification (RV'01)*. 23 July 2001, Paris, France, Electronic Notes in Theoretical Computer Science 55(2).

[7] Cohen, S., http://www.compaq.com/java/download/jtrek.

[8] O'Malley, T.O., Richardson, D.J., and Dillon, L.K. "Efficient Specification-Based Test Oracles", in *Proc. of the Second California Software Symposium (CSS'96)*. April 1996.

[9] Havelund, K. and Rosu, G., "Testing Linear Temporal Logic Formulae on Finite Execution Traces", RIACS Technical Report TR 01-08, May 2001.

[10] Clavel, M., *et al.* "The Maude system", in *Proc. of the 10th International Conference on Rewriting Techniques and Applications (RTA'99)*. July 1999, Trento, Italy. Springer-Verlag, Lecture Notes in Computer Science 1631, pp. 240-243.

[11] Drusinsky, D. "The Temporal Rover and the ATG Rover", in *Proc. of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*. August/September 2000, Stanford, CA. Springer, Lecture Notes in Computer Science 1885, pp. 323-330. K. Havelund, J. Penix, and W. Visser, Eds.

[12] Gerth, R., Peled, D., Vardi, M.Y., and Wolper, P. "Simple On-the-fly Automatic Verification of Linear Temporal Logic", in *Proc. of the 15th IFIP/WG6.1 Symposium on Protocol Specification, Testing and Verification (PSTV'95)*. June 1995, Warsaw, Poland, pp. 3-18.

[13] Hopcroft, J.E. and Ullman, J.D., *Introduction to Automata Theory, Languages, and Computation*: Addison-Wesley, 1979.

[14] Hopcroft, J. "An nlogn algorithm for minimizing states in a finite automaton", in *Proc. of the Theory of Machines and*

*Computations*. 1971, New York. Academic Press, pp. 189-196. Z. Kohavi, Ed.

[15] Paige, R. and Tarjan, R.E., *Three Partition Refinement Algorithms.* SIAM Journal of Computing, Vol. **16**(6), 1987: pp. 973-989.

[16] Etessami, K. and Holzmann, G. "Optimizing Buchi automata", in *Proc. of the 11th International Conference on Concurrency Theory (CONCUR'2000)*. August 2000, Pennsylvania, USA, LNCS (Lecture Notes in Computer Science) 1877, pp. 153-167.

[17] Magee, J. and Kramer, J., *Concurrency: State Models & Java Programs*: John Wiley & Sons, 1999.

[18] Cheung, S.C., Giannakopoulou, D., and Kramer, J. "Verification of Liveness Properties using Compositional Reachability Analysis", in *Proc. of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'97)*. September 1997, Zurich, Switzerland. Springer, Lecture Notes in Computer Science 1301, pp. 227-243. M. Jazayeri and H. Schauer, Eds.