TRACE CHECKING WITH REAL-TIME SPECIFICATIONS

Rocco Deutschmann [1], Matthias Fruth [1,2], Horst Reichel [2], Hans-Christian Reuss [3]
[1] *Dresden University of Technology, Institute for Combustion Engines and Automotive Engineering*
Address: D-01062 Dresden, Germany.
Phone: (+49-351) 463-36587, Fax: (+49-351) 463-32866, e-mail: deutschmann@ivk.tu-dresden.de
[2] *Dresden University of Technology, Institute for Theoretical Computer Science*
Address: D-01062 Dresden, Germany.
Phone: (+49-351) 463-38548, Fax: (+49-351) 463-38348, e-mail: {fruth | reichel}@tcs.inf.tu-dresden.de
[3] *University of Stuttgart, Institute for Combustion Engines and Automotive Engineering*
Address: Pfaffenwaldring 12, D-70569 Stuttgart, Germany.
Phone: (+49-711) 685-8501, Fax: (+49-711) 685-5710, e-mail: reuss@ivk.uni-stuttgart.de

Abstract: Obtaining full models for the validation and verification of embedded systems is often difficult. The presented approach overcomes this problem by checking finite traces, which does not require any system model. Traces are generated by test car runs or hardware-in-the-loop simulation. We propose a canonical extension of linear-time temporal logic (LTL) for real-time specifications. Our algorithm translates real-time LTL formulae into corresponding Büchi automata that check finite traces. The algorithm has been implemented as part of an industrial validation and verification framework for automotive electronics and successfully applied to real-world systems.
Keywords: Runtime verification, trace checking, real-time specifications, temporal logic, Büchi automata

1. INTRODUCTION

Due to the increasing complexity of software and hardware systems, *formal methods* (Clarke *et al.*, 2001) receive increasing attention in automotive electronics. Contrary to the traditional verification approaches testing and simulation, formal methods, like theorem proving and model checking, cover all possible behaviors of a system; that is, they find all violations of a system specification. However, this is only practically feasible for systems of limited size. For example, model checking suffers the infamous state explosion problem: in the worst case, the size of a system model is exponential in its number of boolean variables. Despite the availability of powerful state-space reduction techniques, this problem occurs with all kinds of formal methods and is now considered the main bottleneck of the formal verification approach.

Semi-formal methods are designed to combine the advantages of both formal and non-formal methods. Since they do not explore the full state space of a system, they are much faster and able to handle much larger systems than formal methods. Moreover, they are applied to systems directly instead of system models and they are fully automatic. Highly expressive formal specification languages increase the probability for finding corner-case bugs. The use of temporal logics for property specification allows easy integration with formal methods.

A prevalent semi-formal technique is *trace checking*: first, temporal-logic specifications are represented as rewriting systems, as by Havelund & Roşu (2001), alternating automata, as by Finkbeiner & Sipma (2004), or Büchi and finite automata, as by Giannakopoulou & Havelund (2001). Then, appropriate algorithms from rewriting or automata theory are applied in order to check the truth of given finite execution traces.

For commercial embedded systems, full models are often not available or particularly expensive to construct, since third-party subsystems are usually kept secret. Our approach is particularly suitable for embedded systems, as it does not even require a partial system model; verification can start just from a property specification.

In the past, there have been numerous approaches to combine automata theory and model checking. Validity checking, also known as automata-theoretic model checking, has early received significant attention (Wolper *et al.*, 1983, Vardi & Wolper, 1986, 1994), while truth checking, that is, determining if a trace satisfies some temporal property, has only recently become popular (Vardi, 1997).

In this paper, we present an automata-theoretic method for trace checking with real-time specifications. For the specification of properties, we develop a canonical extension of linear-time temporal logic (LTL). We develop an algorithm that, for a given real-time specification, on-the-fly constructs an equivalent Büchi automaton and verifies the system's execution trace by traversing the automaton. The algorithm can be executed online, this means that the system can be monitored while it is running, and without storing the full trace.

The next section describes LTL and our real-time extension to it, defines a semantics for finite traces, and introduces Büchi automata. Section 3 explains our trace checking algorithm, and discusses possible efficiency improvements. Finally, we discuss computational limits of the presented approach, and give an outlook to future work.

2. PRELIMINARIES

This section introduces the fundamental formalisms of our approach. First, we recall linear-time temporal logic (LTL), define its syntax, infinite-trace and finite-trace semantics. Then, Büchi automata and a translation from LTL to Büchi automata are introduced. Finally, we define a syntactical extension to LTL for the description of real-time properties.

2.1 Linear-time temporal logic

In runtime verification, reactive systems are observed by monitoring their execution traces. Linear-time temporal logic (LTL), invented by Pnueli (1977), provides effective means for expressing system properties: state formulae describe values of boolean variables in a specific state, and path formulae describe safety and liveness properties that apply to each path outgoing of a specific state.

The set of well-formed LTL formulae is inductively defined as follows, starting from a given set P of atomic propositions:

1. Each atomic proposition $p \in P$ is a formula.
2. If φ and ψ are formulae, then $\neg\varphi$, $\varphi \vee \psi$, $X\varphi$, and $\varphi U \psi$ are formulae.

As usual, we have the abbreviations $\varphi \wedge \psi$ for $\neg(\neg\varphi \vee \neg\psi)$, $true$ for $\varphi \vee \neg\varphi$, $false$ for $\neg true$, $\varphi \rightarrow \psi$ for $\neg\varphi \vee \psi$, $F\varphi$ for $true\, U \varphi$, and $G\varphi$ for $\neg F \neg\varphi$. The strong until operator, U, has a dual, R, called release, such that $\varphi R \psi$ stands for $\neg(\neg\varphi U \neg\psi)$.

An *infinite trace* is an infinite sequence of states $\xi = x_0, x_1, \ldots$ such that $x_i \subseteq P$ for all $i \geq 0$. An LTL formula φ is true in a point $i \geq 0$ of an infinite trace ξ, denoted $\xi^i \vDash \varphi$, if the following holds:

$$\xi^i \vDash p \text{ iff } p \in P \text{ and } p \in x_i \quad (1)$$

$$\xi^i \vDash \neg\varphi \text{ iff } \xi^i \nvDash \varphi \quad (2)$$

$$\xi^i \vDash \varphi \vee \psi \text{ iff } \xi^i \vDash \varphi \text{ or } \xi^i \vDash \psi \quad (3)$$

$$\xi^i \vDash X\varphi \text{ iff } \xi^{i+1} \vDash \varphi \quad (4)$$

$$\xi^i \vDash \varphi U \psi \text{ iff } \xi^k \vDash \psi \text{ for some } k \geq i \text{ and } \xi^j \vDash \varphi \text{ for all } i \leq j < k \quad (5)$$

An infinite trace ξ satisfies an LTL formula φ, denoted $\xi \vDash \varphi$, if $\xi^0 \vDash \varphi$.

2.2 Finite-trace semantics

Although LTL was originally designed for infinite traces, there are several possible ways of interpreting LTL formulae over finite traces. For the following discussion, we assume that each finite trace just reflects a limited part of the infinite behavior of a reactive system, and has therefore to be viewed as the prefix of some infinite trace.

Choosing an adequate finite-trace semantics is often a problem, particularly when next-state or eventuality formulae must be evaluated at the end of a trace. In some cases, the truth value of a formula cannot be decided by naively applying the traditional, infinite-trace semantics to the given finite sequence of states. For instance, even if p is true in all states of a finite trace, it cannot be determined whether the safety formula $G\,p$ holds for the whole, infinite trace; also, it cannot be concluded whether the liveness formula $F \neg p$ holds for an infinite extension of this trace, since p may be true in some future state.

Eisner *et al.* (2003) give a good overview of typical problems with and possible semantics for dealing with finite traces. They distinguish three types of finite-trace semantics: weak, neutral, and strong semantics. In the weak view, a formula is true if and only if it is true in some infinite extension of the finite trace. In the strong view, a formula is true if and only if it evaluates to true within the given, finite trace. The neutral view is equivalent to the traditional, infinite-trace semantics.

However, depending on application contexts, different semantics may be preferable. Most approaches to checking finite traces, as of Havelund & Roşu (2001), Giannakopoulou & Havelund (2001), and Finkbeiner & Sipma (2004), use neutral semantics. Giannakopoulou & Havelund (2001) argue that the X operator in LTL is counterintuitive, since users might misattribute some concept of time to it; accordingly, they use LTL-X, a variant of LTL without the X operator, thus avoiding ambiguous interpretations of X formulae. Roşu & Havelund (to appear) propose a simple stationary semantics which extends each finite trace to an infinite one by repeating its last state.

A *finite trace* is a finite sequence of states $\xi = x_0, x_1, \ldots, x_{n-1}$ such that $x_i \subseteq P$ for all $0 \leq i < n$. An LTL formula φ is true in a point $0 \leq i < n$ of a finite trace ξ, denoted $\xi^i \vDash \varphi$, if the infinite-trace semantics together with the following modifications holds:

$$\xi^i \vDash X\varphi \text{ iff } i < n-1 \text{ and } \xi^{i+1} \vDash \varphi \quad (6)$$

$$\xi^i \vDash \varphi U \psi \text{ iff } \xi^k \vDash \psi \text{ for some } i \leq k < n \text{ and } \xi^j \vDash \varphi \text{ for all } i \leq j < k \quad (7)$$

2.3 Büchi automata

In our approach, temporal specifications are represented by Büchi automata, namely labelled generalised Büchi automata (LGBA), that is, Büchi automata with multiple sets of accepting states. For each LTL formula φ, one can construct a corresponding Büchi automaton $\mathcal{A}_\varphi$ that accepts pre-

cisely the traces satisfying φ (Wolper *et al.*, 1983, Vardi & Wolper, 1986, 1994).

A generalised Büchi automaton (GBA) is a quadruple $\mathcal{A} = (\mathcal{Q}, \mathcal{I}, \delta, \mathcal{F})$ where $\mathcal{Q}$ is a finite nonempty set of *states*, $\mathcal{I} \subseteq \mathcal{Q}$ is the set of *initial states*, $\delta : \mathcal{Q} \to 2^{\mathcal{Q}}$ is the *transition function*, and $\mathcal{F} \subseteq 2^{2^{\mathcal{Q}}}$ is the set of sets of *accepting states*; note that $\mathcal{F}$ may be empty. An *execution* of $\mathcal{A}$ is an infinite sequence of states $\sigma = s_0, s_1, \ldots$ such that $s_0 \in \mathcal{I}$ and $s_{i+1} \in \delta(s_i)$ for all $i \geq 0$. An *accepting execution* of $\mathcal{A}$ is an execution such that, for each accepting set $F_i \in \mathcal{F}$, at least one state $q_i \in F_i$ occurs infinitely often in σ. A labelled gerenalised Büchi automaton (LGBA) is a triple $(\mathcal{A}, \mathcal{D}, \mathcal{L})$ where $\mathcal{A}$ is a GBA, $\mathcal{D}$ is the finite *domain*, and $\mathcal{L} : \mathcal{Q} \to 2^{\mathcal{D}}$ is the *labelling function*. An LGBA accepts a trace $\xi = x_0, x_1, \ldots$ over $\mathcal{D}$ if there exists an accepting execution $\sigma = s_0, s_1, \ldots$ such that $x_i \in \mathcal{L}(s_i)$ for each $i \geq 0$.

2.4 Translating LTL to LGBA

One of the first non-worst-case approaches to the translation of LTL formulae into Büchi automata was developed by Gerth *et al.* (1995). This algorithm, based on a tableau construction for LGBA, was later improved by Daniele *et al.* (1999). It can be used on-the-fly, that is, only those parts of the automaton currently needed for the traversal of the trace are constructed and stored in memory. Several optimisations of these techniques have been proposed, and some of them are discussed in the next section.

We now give a brief overview of this algorithm; for more details, consult Gerth *et al.* (1995) and Daniele *et al.* (1999).

Using a tableau procedure, the algorithm constructs a graph whose nodes and edges define states and transitions of the automaton. Each node consists of five fields:

1. *name* is a unique identifier.
2. *incoming* is the set of nodes that lead to this node.
3. *new* is the set of formulae that must hold in this node and have not yet been processed.
4. *old* is the set of formulae that must hold in this node and have already been processed.
5. *next* is the set of formulae that must hold in all immediate successors of this node.

Nodes are expanded by using a set of generic rules, which are applied depending on the type of unprocessed formulae in their *new* fields. In order to process a *new* formula for the current node, the formula is decomposed to new, but simpler, obligations, that are either added to this node, or, by splitting the current node, to exactly two replacing nodes.

Starting with a single node, containing precisely the LTL formula to be translated in *new* and "init" in *incoming*, the algorithm applies tableau rules until all *new* formulae in all nodes have been processed. It maintains a set of completely expanded nodes, which essentially defines the state graph. Once the expansion of a node is completed, it is added to this set, and merged with equivalent nodes if there are any; nodes are equivalent if they have equal *next* and *old* fields, and they are merged by merging their *incoming* fields. For our notation of this algorithm, we require all formulae to be in negation normal form, that is, negation symbols occur only directly in front of propositions. The algorithm uses the following set of tableau rules:

1. If *new* contains an atomic formula ($p \in P$, $true$, or $false$), then discard it if it contradicts with some formula in *old*, otherwise add it to *old*.
2. If *new* contains a formula $X\varphi$, then add $X\varphi$ to *old* and φ to *next*.
3. If *new* contains a formula $\varphi \wedge \psi$, then add each of φ, ψ to *new* unless it is already in *old*.
4. If *new* contains a formula μ of the form $\varphi_1 U \varphi_2$, $\varphi_1 R \varphi_2$, or $\varphi_1 \vee \varphi_2$, split the current node into two replacing nodes that are copies of it. Under consideration of the tableau rules in *Figure 1*, update the *new* and *next* fields of these nodes: add those formulae from $new1(\mu), new2(\mu), next1(\mu)$ that are not already in *old* to the respective *new* and *next* fields of the new nodes.

μ	$new1(\mu)$	$next1(\mu)$	$new2(\mu)$
$\varphi_1 U \varphi_2$	$\{\varphi_1\}$	$\{\varphi_1 U \varphi_2\}$	$\{\varphi_2\}$
$\varphi_1 R \varphi_2$	$\{\varphi_2\}$	$\{\varphi_1 R \varphi_2\}$	$\{\varphi_1, \varphi_2\}$
$\varphi_1 \vee \varphi_2$	$\{\varphi_1\}$	$\{\}$	$\{\varphi_2\}$

Fig. 1. Tableau rules for node splitting.

After termination of the algorithm, a GBA can be obtained from the constructed state graph as follows: the set $\mathcal{Q}$ of states is the set of graph nodes. The set $\mathcal{I}$ of initial states is the set of those states in $\mathcal{Q}$ that have "init" in their *incoming* field. For each two states s and t, a transition $t \in \delta(s)$ from s to t exists if and only if $s \in incoming(t)$. To ensure that each formula $\varphi U \psi$ is eventually satisfied, we require that for each subformula of the form $\varphi U \psi$, $\mathcal{F}$ must include a set containing all nodes q such that either $\varphi U \psi \in new(q)$ or $\psi \in new(q)$.

The GBA can be extended to an LGBA by setting the finite domain $\mathcal{D}$ to $2^{\mathcal{P}}$ and the labelling of a state s to $\mathcal{L}(s) = \{X : X \supseteq Pos(s) \wedge X \cap neg(s) = \emptyset\}$ where $pos(s) = old(s) \cap P$ and $neg(s) = \{\mu : \neg\mu \in old(s) \wedge \mu \in P\}$. Thereby, each state s is labelled with all sets in 2^P that are compatible with $old(s)$.

2.5 Real-time LTL

In the past, there have been various developments in the field of real-time logics (Koymans, 1990, Emerson *et al.*, 1992, Alur & Henzinger, 1993). Our approach aims on extending LTL for real-time specifications by annotating discrete-time boundaries to temporal operators.

For instance, $G\,(p \rightarrow F_{5,10}\,q)$ expresses the property that for each state satisfying p, some future state in the time range from 5 to 10 satisfies q. For real-time systems, the annotated time values map directly to system times; for instance, "whenever p holds, q must follow within 5 to 10 milliseconds".

Our approach is similar to that of real-time CTL (RTCTL), by Emerson *et al.* (1992), since they also annotate next-state and strong until operators with nonnegative integer timepoints, but for the branching-time logic CTL. It is similar to metric temporal logic (MTL), presented by Koymans (1990) and Alur & Henzinger (1993), though their logic includes past-time operators. However, we think that LTL is better suited for on-the-fly verification than CTL, and verifying past-time formulae is not possible online, since it requires trace storing. Therefore, we expect our approach to be computationally more efficient.

The set of well-formed real-time LTL (RTLTL) formulae is inductively defined, using the defining rules for LTL and the following one:

3. If φ and ψ are formulae, and a and b integers, then $X_a\varphi$ and $\varphi U_{a,b}\psi$ are formulae.

For RTLTL, we have the additional abbreviations $F_{a,b}\varphi$ for $true U_{a,b}\varphi$, $G_{a,b}\varphi$ for $\neg F_{a,b}\neg\varphi$, and $\varphi R_{a,b}\psi$ for $\neg(\neg\varphi U_{a,b}\neg\psi)$.

The finite-trace semantics of RTLTL is defined with respect to that of LTL, by adding two more rules:

$$\xi^i \vDash X_a\varphi \text{ iff } i < n - a \text{ and } \xi^{i+a} \vDash \varphi \quad (8)$$

$$\xi^i \vDash \varphi U_{a,b}\psi \text{ iff } \xi^k \vDash \psi \text{ for some } k \text{ such that } i+a \leq k \leq i+b, k < n, \text{ and } \xi^j \vDash \varphi \text{ for all } i \leq j < k \quad (9)$$

Note that RTLTL and LTL are still semantically equal, since the following equivalences hold:

$$X_{a+1}\varphi \equiv X X_a\varphi \quad (10)$$

$$\varphi U_{a+1,b+1}\psi \equiv \varphi \wedge X(\varphi U_{a,b}\psi) \quad (11)$$

$$\varphi U_{0,b+1}\psi \equiv \psi \vee (\varphi \wedge X(\varphi U_{0,b}\psi) \quad (12)$$

$$\varphi U_{0,0}\psi \equiv \psi \quad (13)$$

Hence, one can easily translate RTLTL formulae into standard LTL formulae by subsequent application of the above equivalences as rules.

3. TRACE CHECKING

In our application context, traces are recorded in the original system or in a test bench: in other words, they are either generated from a test car run or from measuring data that is obtained by simulation in a hardware-in-the-loop framework.

System properties are specified in RTLTL, and then either canonically rewritten into LTL, or directly used to construct an LGBA, employing an extended version of the presented tableau algorithm. These formulae are then translated into Büchi automata, which are either computed once and optionally stored for later use, or computed and traversed on-the-fly.

Finally, finite traces, which are either prerecorded or read online during a test run, are checked against arbitrary specifications that are given by their corresponding Büchi automata. The trace-checking task itself is easy and performed by classic search algorithms. If a verification run fails, our tool immediately discovers the source of the violation, and indicates the respective time point in the trace.

In this section, we present different operation modes of our algorithms, featuring canonical RTLTL-to-LTL translation, symbolic treatment of real-time formulae for on-the-fly LGBA construction, and online monitoring. We finally discuss efficiency improvements which have been integrated into our software or which we intend to adopt.

3.1 A basic algorithm

The basic algorithm proceeds in three phases: first, the given real-time specification, an RTLTL formula φ, is translated into an equivalent LTL formula φ'. Then, an LGBA $A_{\varphi'}$ that accepts precisely the traces satisfying φ' is computed. Finally, arbitrary finite traces can be verified by traversing the automaton: classic search algorithms are applied in order to find an accepting execution of the automaton; for the interpretation of the given traces, we use a neutral semantics. It turns out that for finite traces, the LGBA acceptance condition reduces to finite automata acceptance (Giannakopoulou & Havelund, 2001). Consequently, it suffices to treat all constructed Büchi

automata as finite automata.

For finding accepting executions, both forward and backward depth-first search are favourable strategies as long as the trace has been pre-computed and stored, while backward depth-first search is the only advisable method for online trace traversal. A comparison of basic search algorithms for checking finite traces with automata can be found in Finkbeiner & Sipma (2004).

Correctness and termination of the tableau procedure for LTL-to-LGBA translation has been proven in the original papers (Gerth *et al.*, 1995, Daniele *et al.*, 1999). For a more comprehensive discussion on the correctness of LTL-to-Büchi translation algorithms in general, the interested reader is referred to Tauriainen & Heljanko (2002). Correctness and termination proofs of our algorithm, including the below described extensions to this basic algorithm, can be found in Fruth (to appear).

Roşu & Havelund (to appear) propose a useful taxonomy for runtime verification techniques. Their assessment follows three criteria: first, is the algorithm trace-storing or non-storing (online)? Second, is it synchronous or asynchronous, that is, does it detect errors at the same time as they occur (very hard to achieve) or not? And third, is it predictive or exact? For runtime verification with real-time specification, we are also concerned with the issue whether real-time formulae have to be unfolded (consuming a lot of memory), and we want to know whether the automata are constructed on-the-fly (memory-efficient).

According to these classification measures, all our algorithms are asynchronous and exact, and each of them can be used in a non-storing fashion. However, they differ in the last two criteria.

In the remainder of this section, we will outline functional enhancements of our basic algorithm, regarding the last two points: symbolically representing real-time formulae and on-the-fly operation. For an exhaustive presentation of the omitted technical details, we refer to the second author's thesis report on runtime verification (Fruth, to appear).

3.2 On-the-fly verification

The tableau procedure of Gerth *et al.* (1995), together with the improvements of Daniele *et al.* (1999), can easily be executed on-the-fly. Nodes are then expanded on demand, that is, depending on the propositions that hold in the current state of the trace. Only completely expanded nodes are visited. Already visited nodes are remembered by storing their hash values; when a node experiences future changes, its hash value also changes, and it must be revisited. By storing only those parts of the automaton in memory that are currently needed for the traversal of the trace, considerable efficiency improvements can be achieved.

3.3 Real-time LGBA

In the basic algorithm, no matter whether applied on-the-fly or not, all real-time operators are unfolded for LGBA construction. Clearly, in order to find an accepting execution, the unfolding of some operators is eventually needed. However, finding a way to avoid unnecessary unfoldings would be of significant benefit.

For this purpose, nodes of the tableau graph are allowed to contain real-time operators. The tableau rules are naturally extended by the equivalences 10 to 13 from 2.5; for instance, satisfying $X_5\varphi$ in the current node requires satisfying $X_4\varphi$ in all immediate successors of this node. Since the structure of these rules is completely analog to that of the existing rules for LTL, they are omitted here.

3.4 Further efficiency improvements

Several authors have proposed different methods for achieving reductions in running time and memory consumption (Daniele *et al.*, 1999, Somenzi & Bloem, 2000, Giannakopoulou & Havelund, 2001, Tauriainen, 2003). Daniele *et al.* (1999) and Giannakopoulou & Havelund (2001) describe a revised version of the formerly state-of-the-art tableau method of Gerth *et al.* (1995), eliminating redundancies in the translation phase. Somenzi & Bloem (2000) and Etessami & Holzmann (2000) independently introduce a formula rewriting phase preceding the automata construction. They also apply simulation relations in order to minimise the obtained Büchi automata; unfortunately, this technique cannot be used on-the-fly. Wolper (2001) propose an early formula rewriting step, as well as early detection of nontrivial inconsistencies.

4. CONCLUSION

We have presented a real-time extension to linear-time temporal logic and a method for checking finite traces against real-time specifications formulated in this language. Our approach can be used online, the automata construction can be performed on-the-fly, and RTLTL formulae are unfolded into LTL only if necessary. All algorithms have been implemented in the programming language Python as part of an industrial validation and verification framework for automotive electronics.

The presented verification method is efficient and scalable. The constructed Büchi automata are probably already essentially optimal. Büchi automata are a widely known and well-researched representation for temporal-logic specifications, and our results are competitive to those of others; comparisons with the best available reference algorithms (Daniele *et al.*, 1999, Gastin & Oddoux,

2001) show competitive results for computation time and memory usage.

Further efficiency gains may be possible by improving the initial formula rewriting step, by adding further tableau rules for real-time operators, and by employing automata-theoretic reductions methods.

REFERENCES

Alur, R. and Henzinger, T. A. (1993). Real-Time Logics: Complexity and Expressiveness. *Information and Computation*, 104(1):35–77.

Clarke, E. M., Grumberg, O. and Peled, D. A. (2001). Model Checking. The MIT Press.

Daniele, M., Giunchiglia, F. and Vardi, M. Y. (1999). Improved Automata Generation for Linear Temporal Logic. In Proceedings of the 11th International Conference on Computer Aided Verification, volume 1633 of *Lecture Notes in Computer Science*. Springer.

Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A. and Van Campenhout, D. (2003). Reasoning with Temporal Logic on Truncated Paths. In Proceedings of the 15nd International Workshop on Computer Aided Verification, volume 2725 of *Lecture Notes in Computer Science*. Springer.

Emerson, E. A., Mok, A. K., Sistla, A. P. and Srinivasan, J. (1992). Quantitative Temporal Reasoning. *Real-Time Systems*, 4(4):331–352.

Etessami, K. and Holzmann, G. J. (2000). Optimizing Büchi Automata. In Proceedings of the 11th International Conference on Concurrency Theory, volume 1877 of *Lecture Notes in Computer Science*. Springer.

Finkbeiner, B. and Sipma, H. (2004). Checking Finite Traces using Alternating Automata. *Formal Methods in System Design*, 24(2):101–127.

Fruth, M. (to appear). Runtime Verification of Embedded Real-Time Systems. Diploma thesis, Dresden University of Technology, Dresden, Germany.

Gastin, P. and Oddoux, D. (2001). Fast LTL to Büchi Automata Translation. In Proceedings of the 13th Conference on Computer Aided Verification, volume 2102 of *Lecture Notes in Computer Science*. Springer.

Gerth, R., Peled, D. A., Vardi, M. Y. and Wolper, P. (1995). Simple On-the-fly Automatic Verification of Linear Temporal Logic. In Proceedings of the 15th IFIP WG 6.1 International Symposium on Protocol Specification, Testing and Verification, volume 38 of *IFIP Conference Proceedings*. Chapman and Hall.

Giannakopoulou, D. and Havelund, K. (2001). Runtime Analysis of Linear Temporal Logic Specifications. RIACS Technical Report 01.21, Research Institute for Advanced Computer Science, Moffett Field, California, USA.

Havelund, K. and Roşu, G. (2001). Monitoring Programs using Rewriting. In Proceedings of the 16th IEEE International Conference on Automated Software Engineering. IEEE Computer Society Press.

Koymans, R. (1990). Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems*, 2(4):255–299.

Pnueli, A. (1977). The Temporal Logic of Programs. In Proceedings of the 18th IEEE Symposium on Foundations of Computer Science. IEEE Computer Society Press.

Roşu, G. and Havelund, K. (to appear). Rewriting-based Techniques for Runtime Verification. *Automated Software Engineering*.

Somenzi, F. and Bloem, R. (2000). Efficient Büchi Automata from LTL Formulae. In Proceedings of the 12th International Conference on Computer Aided Verification, volume 1855 of *Lecture Notes in Computer Science*. Springer.

Tauriainen, H. (2003). On Translating Linear Temporal Logic into Alternating and Nondeterministic Automata. Research report A83, Laboratory for Theoretical Computer Science, Helsinki University of Technology, Espoo, Finland.

Tauriainen, H. and Heljanko, K. (2002). Testing LTL Formula Translation into Büchi Automata. *International Journal on Software Tools for Technology Transfer*, 4(1):57–70.

Vardi, M. Y. (1997). Alternating Automata: Unifying Truth and Validity Checking for Temporal Logics. In Proceedings of the 14th International Conference on Automated Deduction, volume 1249 of *Lecture Notes in Computer Science*. Springer.

Vardi, M. Y. and Wolper, P. (1986). An Automata-Theoretic Approach to Automatic Program Verification. In Proceedings of the 1st Symposium on Logic in Computer Science. IEEE Computer Society Press.

Vardi, M. Y. and Wolper, P. (1994). Reasoning

about Infinite Computations. *Information and Computation*, 115(1):1–37.

Wolper, P. (2001). Constructing Automata from Temporal Logic Formulas: A Tutorial. In Lectures on Formal Methods and Performance Analysis: Proceedings of the 1st EEF/Euro Summer School on Trends in Computer Science, Revised Lectures, volume 2090 of *Lecture Notes in Computer Science*. Springer.

Wolper, P., Vardi, M. Y. and Sistla, A. P. (1983). Reasoning about Infinite Computation Paths (Extended Abstract). In Proceedings of the 24th IEEE Symposium on Foundations of Computer Science. IEEE Computer Society Press.