Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Generating On-Line Test Oracles from Temporal Logic Specifications *

John Håkansson[1], Bengt Jonsson[2], Ola Lundqvist[3]

[1] IAR Systems AB, Uppsala, Sweden e-mail: john@iar.se
[2] Uppsala University, Department of Computer Systems, Sweden. e-mail: bengt@docs.uu.se
[3] Volvo Technical Development Corporation, Gothenburgh, Sweden. e-mail: ola.lundkvist@volvo.com

Abstract. This paper is concerned with the problem of checking, by means of testing, that a software component satisfies a specification of temporal safety properties. Checking that an actual observed behavior conforms to the specification is performed by a *test oracle*, which can be either a human tester or a software module. We present a technique for automatically generating test oracles from specifications of temporal safety properties in a metric temporal logic. The logic can express quantitative timing properties, and can also express properties of data values by means of a quantification construct. The generated oracle works *on-line* in the sense that checking is performed simultaneously with observation. The technique has been implemented and used in case studies at Volvo Technical Development Corp.: a cruise control module and a throttle module.

1 Introduction

Embedded computer systems are increasingly employed in safety-critical applications, such as cars, airplanes, etc. Their functionality must therefore be thoroughly validated before being deployed in their actual environment. Of particular concern is to ascertain that they meet safety requirements. For instance, in a car, an activated cruise control must be switched off when the driver presses the braking pedal. All techniques and tools that can increase confidence in the functioning of an embedded system are applicable: special-purpose languages for specification and design, compilers and code generators, and testing and verification techniques for validating the specification and the code.

* This work was supported by VINNOVA within the ASTEC competence center and by Volvo Technical Development Corporation

Testing is one of the most important techniques to check that a component satisfies safety requirements. It has the advantages that it is easy to conduct and that source code need not be available: When a system is composed of components that are implemented by different subcontractors, source code is often not available, and testing may be the only means to check conformance to safety requirements.

To obtain a high degree of confidence, the testing procedures should subject the system to a wide range of input signals, and check that the behavior of the system conforms to its safety requirements. It is tedious to perform a large number of tests manually; test execution should therefore be automated in order to cover a large range of possible input values. Two major problems in both manual and automated testing are the following.

1. To select appropriate inputs to the system, in order to investigate its behavior under a variety of operating and failure conditions. Ideally, the selection should reveal as many potential deficiencies of the system as possible, using the available resources (time, number of tests, etc).
2. To check that the output of the system conforms to its requirements, in particular safety requirements. This problem is often referred to as the *oracle problem*.

These two problems can be considered separately from each other. In this paper, we address the oracle problem.

A common approach to the oracle problem is to prepare a number of *test cases* or *test sequences*, each of which is a predefined sequence of input and corresponding output values. This approach works well only if the system under test is *deterministic*, i.e., if output values are uniquely determined by input values and their ordering in time, an assumption that is not valid in general for distributed embedded systems.

An alternative approach that we consider in this paper is to generate a *test oracle* from the safety require-

ments. A test oracle is a separate module, which observes both the input and the output of a component, and is able to determine whether the observed sequence of inputs and outputs conforms to the (safety) requirements for the system. This approach does not assume unique output values, and solves the oracle problem in one construction. A potential problem is that it may not be trivial to construct a test oracle for an arbitrary requirement.

The structure of our approach is illustrated in Figure 1, which shows a module that generates input values to the component under test. These inputs and the generated outputs are observed by the test oracle, which reports violations of the component's requirements.

In practice, test oracles are usually constructed manually in some programming or scripting language from a specification or from an informal understanding of the requirements. In this paper, we instead advocate to generate oracles from requirements written in an abstract specification language, which can be used both for documentation, validation, formal verification, and test oracle generation. The advantages of this are that formulation of requirements is separated from the technicalities of testing, and that requirements need only be maintained in one form when they are changed. Such a specification language should be able to express properties that describe the systems' behavior over time, such as:

> whenever the input signal *val* has a value which exceeds 5, the output signal *alert* will be *true* within 0.3 seconds,

or such as:

> when a positive edge of the input signal *acc* occurs, the output value *ref-speed* must start to increase until it reaches the value *max-speed*. Between the start and the end point, the increase should correspond to an acceleration of 0.5 km/h/s.

Temporal logic [16] is a family of specification formalisms that have been used mainly for formal verification. Several model checkers (e.g., SPIN [12] and SMV [18]) use temporal logic to specify requirements to be checked. Some temporal patterns are easily represented in temporal logic. There are also many requirements which result in clumsy formulas, and which could be more easily formulated in a procedural formalism for defining test oracles more directly, or in some sequence diagram like formalism. An advantage of a logical formalism is the availability of operators, by which new requirements can be composed from existing parts. As a support for temporal logic specification, Dwyer et al. [5] have proposed a pattern system containing the most widely used types of temporal formulas. They found that a vast majority of specifications used in model checkers are covered by these patterns.

The generation of a test oracle from a temporal logic formula is similar to the problem of generating an automaton which accepts the set of behaviors that are defined by a temporal logic formula. This problem is well-understood for propositional temporal logics, and is implemented in model-checkers such as SPIN [12]. Recently, the techniques for implementing small Büchi automata efficiently from temporal logic formulas have been significantly improved [3,6,25,9]. A difference between model checking and testing is that an oracle for model checking need not be deterministic, since a model checker has access to a complete system model and can consider all possible outcomes of nondeterministic choices in a tester (e.g., by backtracking). Of course, a deterministic oracle is likely to make verification faster by reducing backtracking, but backtracking cannot be eliminated if the system model is nondeterministic. In contrast, an on-line test oracle should preferrably be deterministic, since it is difficult to backtrack during testing.

In the context of testing, generation of oracles has been considered for temporal logics with discrete time [4, 11,20,21,24] Our work differs from that work, mainly by considering a richer logic which contains past operators, metric time, and can handle data values by means of a quantification construct.

Since there are many different temporal logics with metric time, our method for constructing test oracles should be easily adaptable to new logic constructs. We have therefore tried to make it syntax-directed, and to focus on general principles which can be adapted to other similar specification constructs.

As a specification language for requirements, we have in the case studies at Volvo used TRIO, which is a metric temporal logic that has previously been used for specifying safety requirements of embedded system components at Volvo [19]. TRIO is a first-order logic with special constructs for handling metric time. It is very expressive, and can easily express undecidable properties. In this presentation, we will instead use a more restrictive temporal logic, which can only express safety properties, and which can express properties of metric time and of data using so-called freeze quantification [2]. Intuitively, this construct has a natural operational interpretation as assigning the value of an expression to a variable in a temporal formula.

We have implemented the translation to test oracles, and used it on safety requirements for two softare modules provided by Volvo Technical Development: a Cruise controller and a torque controller. As the target language for constructing test oracles, we have used FIL, a language used by Volvo for fault-injection in testing computer components in cars. A FIL program executes cyclically. In each cycle the program reads output values from the component, performs calculations, and then writes input values of the components and updates internal variables of the FIL program.

Related Work. Testing automation for embedded systems has been addressed in the framework of the language LUSTRE. Ouabdesselam et al [21,22] and Ray-

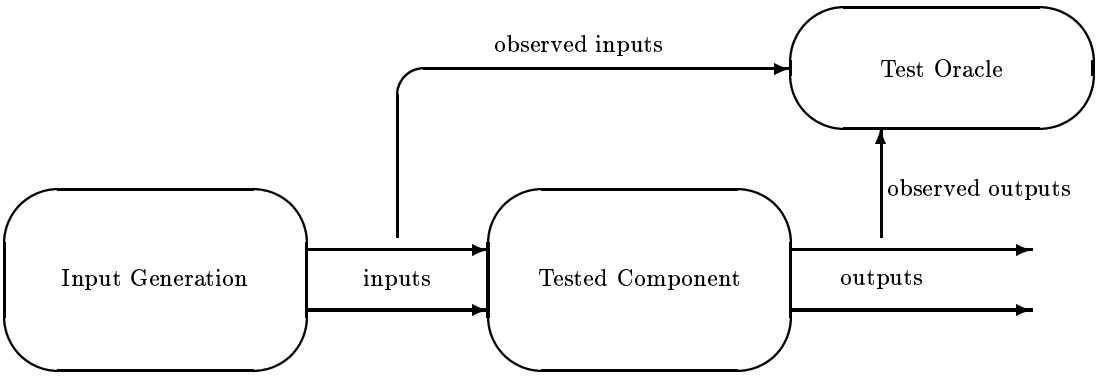

Fig. 1. Role of Test Oracle

mond et al [24] expresses requirements as synchronous observers in LUSTRE. These can be seen as test oracles that are expressed in a high-level executable language. Raymond et al [24] also present methods for generating relevant inputs in test sequences. Our specification language is more expressive than LUSTRE.

Kesten et al [13,14] present algorithms for transforming propositional temporal logic with past operators into automata for use in model-checking. In this work, the resulting automaton form can be highly nondeterministic, making it non-suitable for test oracles. Fisher [8] presents a transformation of temporal logic into an executable form, where nested temporal formulas are flattened by introducing new variables, and where a formula is decomposed to reveal how the past affects the future. Fisher handles arbitrary propositional formulas with past and future operators. The resulting programs exhibit nondeterminism, which is handled by backtracking.

Generation of test oracles is considered by O'Malley et al. [20], who present a technique for automatically generating test oracles from formulas in GIL (Graphical Interval Logic), an interval-based temporal logic. The translation resembles the generation of deterministic finite automata from regular expressions. GIL cannot express quantitative timing properties. The Temporal Rover [4] is a tool which generates executable code in, e.g., C, C++, or Java, from temporal logic assertions that are included as comments in the source file. The Temporal Rover has a similar function as our test oracle generator. The logic considered by the Temporal Rover does not have a quantification construct. Havelund and Rosu [11] describe how to synthesize oracles for different fragments of linear past-time logic, and how the oracle code can be made to run efficiently. They do not consider future operators, metric time, or quantification. Generation of test oracles have also been considered in the context of simulation, where they are usually called *simulation checkers*. An example is FoCs [1], developed at IBM Haifa Research Laboratory, which generates simulation checkers from propositional temporal logic specifications.

Approaches to generating oracles for procedures include Peters and Parnas [23], who derive oracles for input-output specifications from a formal notation based on boolean expressions and bounded quantification.

Mandrioli et al. present a technique for test case generation from TRIO specifications. Instead of generating an oracle, they generate execution sequences which should be observed when the system executes. A major problem with this approach is that each execution sequence (test case) must predict the outputs and timing of all outputs of the system. This can only be done for completely deterministic systems, which is rather restricitve since a specification usually allows some freedom in the timing of output events within certain constraints. In the nondeterministic case [15, Sec. 5], a "history checker" [7] can check whether output conforms to the specification. This history checker is off-line, i.e., it can only check a completed behavior.

Our case studies owe much to the specification of a cruise controller in TRIO developed by Nielsen [19]. A full version of the specification case studies presented in this paper have appeared in the report [10].

Outline. In the next section, we present the execution model for components in embedded systems and test oracles. Section 3 presents our temporal logic. The method for generating test oracles from this logic is presented in Section 4. A discussion of how the translation was applied to a case study specification is contained in Section 6. Conclusions and directions for future work are presented in Section 7.

2 System Model

We assume that a system, or a component of a system, can be observed through a set of *system variables*, whose values change over time. We will not formally distinguish

between input and output variables, since they are observed in the same way by the test oracle (such a distinction would however be important when selecting input values). We assume that a test oracle observes the system periodically, at a potentially infinite sequence of *time instants*, and that there is a constant delay between any two consecutive observation instants. The delay could, e.g., be taken as the duration of a basic control loop of the component during which inputs are read, internal computations are carried out, and outputs are generated.

Time instants are totally ordered by $<$. For a time instant t, let t^+ denote the succeeding time instant, and t^- denote the preceding time instant. We adopt the convention that t^- is t when t is the initial time instant. A *state* is a mapping from system variables to values. We use s to range over states, and $s[v]$ to denote the value of variable v in state s. A *behavior* is a mapping from the infinite sequence of consecutive time instants to states. We use σ to range over behaviors, and $\sigma(t)$ to denote the state at instant t in the behavior σ. Thus $\sigma(t)[v]$ is the value of variable v at time instant t of the behavior σ.

In the following, let $\overline{v} = v_1, \ldots, v_m$ be the set of system variables. We assume a distinguished variable *now*, which always contains the value of the current time, i.e., it can be thought of as an idealized system clock. Note that the advance of *now* need not be synchronized with the successive time instants. For instance, if the distance between successive time instants is 3 microseconds, then *now* advances by 3 microseconds between successive time instants.

3 Metric Temporal Logic

A temporal logic is a language for expressing properties of behaviors. For each time instant t and behavior σ, the value of formula ϕ at time instant t in the behavior σ, denoted $\sigma(t)[\phi]$, can be either true or false. As an example, a formula can say that "in exactly 3 time instants, the variable x will be larger than in the current time instant". Such a formula may be true or false, depending on the time instant at which it is evaluated.

Let us now define the temporal logic used in this paper. We assume a given vocabulary for forming *expressions* over system variables, consisting of constants, function and predicate symbols, such as $<$, $+$, mod, etc. We assume that each symbol in this vocabulary has a direct correspondence in the language for programming test oracles. Thus, an expression over system variables can be directly translated into a corresponding expression in the target programming language for test oracles. Examples of expressions are over system variables, such as v_1+4, $v_1 < v_2$, etc. The evaluation of system variables in states is extended to the evaluation of expressions in the natural way. For instance, $\sigma(t)[v_1 + 4]$ is defined as $\sigma(t)[v_1] + 4$.

Two special cases of expressions are as follows.

- A *boolean-valued* expression is also called a *state formula*. Examples of state formulas are $v_1 < v_2$ and $iseven(v_3)$.
- A *distance term* is an *integer-valued* expression, whose value is always nonnegative. In formulas, distance terms are used to denote distance in time.

Temporal formulas are formed from state formulas and distance terms by applying *temporal operators*. Let ϕ and ψ range over temporal formulas, and τ range over distance terms.

- Any *state formula* is also a temporal formula, expressing that the state formula holds at the current time instant.
- $\circ\ \phi$ states that ϕ will hold in the next time instant.
- $\ominus\ \phi$ states that ϕ was true in the previous time instant. By convention, $\ominus\ \phi$ is equivalent to ϕ in the initial time instant.
- $\phi\ \mathcal{W}\ \psi$ states that ϕ will hold at all time instants from the current time instant up to the next time instant at which ψ holds, or that ϕ holds at all present and future time instants.
- $\phi\ \mathcal{S}\ \psi$ states that ψ held at some past time instant, and that ϕ held at all time instants between the last time instant at which ψ was true and the current time instant.
- $\phi\ \mathcal{B}\ \psi$ is the past-time analogue of $\phi\ \mathcal{W}\ \psi$, and states that either $\phi\ \mathcal{S}\ \psi$, or that ϕ was true at all previous time instants.
- $\diamondsuit_\tau \phi$ states that ϕ has been true for at least τ units of absolute time, where τ is evaluated in the current time instant.
- $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$ are temporal formulas if ϕ_1 and ϕ_2 are temporal formulas.

From the temporal operators defined above, additional operators can, in the usual way, be defined as follows.

- $\Box\ \phi$ states that ϕ will hold at all present and future time instants. It can be defined as $\Box\ \phi \equiv \phi\ \mathcal{W}\ \mathit{false}$
- $\boxminus\ \phi$ is the past time analogue of $\Box\ \phi$, and can be defined as $\Box\ \phi \equiv \phi\ \mathcal{B}\ \mathit{false}$
- $\diamondsuit\ \phi$ states that ϕ has been true sometimes in the past, and can be defined by $\diamondsuit\ \phi \equiv \mathit{true}\ \mathcal{S}\ \phi$.
- $\diamondsuit_\tau \phi$ states that ϕ has been true for at least one time instant during the last τ units of absolute time, where τ is evaluated in the current time instant, and can be defined by $\diamondsuit_\tau \phi \equiv \neg \diamondsuit_\tau \neg \phi$.

Quantification In order to express properties of data values and metric time, we use a special form of quantification, so-called "freeze quantification", which is a generalization of the quantification presented by Alur and Henzinger in the logic TPTL [2]. If *exp* is an expression (without temporal operators) and x is a variable, then the formula $x := exp.\phi$ states that ϕ is true with the

$$
\begin{array}{lcl}
\sigma(t)[\circ\ \phi] & \equiv & \sigma(t^+)[\phi] \\
\sigma(t)[\ominus\ \phi] & \equiv & \sigma(t^-)[\phi] \\
\sigma(t)[\phi\ \mathcal{W}\ \psi] & \equiv & [\exists t' \geq t\ .\ \sigma(t')[\psi] \ \wedge\ \forall t'' : t \leq t'' < t'\ .\ \sigma(t'')[\phi]] \ \vee\ [\forall t' \geq t\ .\ \sigma(t')[\phi]] \\
\sigma(t)[\phi\ \mathcal{S}\ \psi] & \equiv & \exists t' \leq t\ .\ \sigma(t')[\psi] \ \wedge\ \forall t'' : t' < t'' \leq t\ .\ \sigma(t'')[\phi] \\
\sigma(t)[\phi\ \mathcal{B}\ \psi] & \equiv & [\exists t' \leq t\ .\ \sigma(t')[\psi] \ \wedge\ \forall t'' : t' < t'' \leq t\ .\ \sigma(t'')[\phi]] \ \vee\ [\forall t' \leq t\ .\ \sigma(t')[\phi]] \\
\sigma(t)[\Box\ \phi] & \equiv & \forall t' \geq t\ .\ \sigma(t')[\phi] \\
\sigma(t)[\boxminus\ \phi] & \equiv & \forall t' \leq t\ .\ \sigma(t')[\phi] \\
\sigma(t)[\Diamond\ \phi] & \equiv & \exists t' \leq t\ .\ \sigma(t')[\phi] \\
\sigma(t)[\Box_\tau \phi] & \equiv & \forall t' \geq t\ :\ (\sigma(t')[now] \leq \sigma(t)[now] + \sigma(t)[\tau])\ .\ \sigma(t')[\phi] \\
\sigma(t)[\Diamond_\tau \phi] & \equiv & \exists t' \geq t\ :\ (\sigma(t')[now] \leq \sigma(t)[now] + \sigma(t)[\tau])\ .\ \sigma(t')[\phi] \\
\sigma(t)[\boxminus_\tau \phi] & \equiv & \forall t' \leq t\ :\ (\sigma(t)[now] - \sigma(t)[\tau]) \leq \sigma(t')[now]\ .\ \sigma(t')[\phi] \\
\sigma(t)[\Diamond_\tau \phi] & \equiv & \exists t' \leq t\ :\ (\sigma(t)[now] - \sigma(t)[\tau]) \leq \sigma(t')[now]\ .\ \sigma(t')[\phi] \\
\sigma(t)[x := exp.\phi] & \equiv & \sigma(t)[\phi[\sigma(t)[exp]/x]]
\end{array}
$$

Table 1. Semantics of Temporal Operators and of Quantification

current value of exp substituted for x. Note that x is a logical variable, used only in the formula. We assume that each logical variable x is quantified at most once in a formula, and that logical variables are distinguished from system variables (in order to avoid name clashes in the usual way). The semantics for freeze quantification is formally defined as

$$\sigma(t)[x := exp.\phi] \ \equiv\ \sigma(t)[\phi[\sigma(t)[exp]/x]]$$

Intuitively, in a formula $x := exp.\phi$, the current value of exp is "frozen" (recorded) in the variable x before evaluating the formula ϕ. By referencing now in exp, this can be used to express time bounds. For instance, the formula

$$x := now\ .\ (\phi \wedge now \leq x + 5)\ \mathcal{W}\ \ (now > x + 5)$$

states that the formula ϕ must be true for 5 time units from now on. We will use $x\ .\ \phi$ as a shorthand for $x := now\ .\ \phi$. The property can also be expressed as

$$x\ .\ \Box\ (\phi \vee (now > x + 5))$$

The idea of freeze quantification to freeze the current time was presented by Alur and Henzinger [2]. We can extend the idea to freezing the value of any expression, not just the value of now. For instance, the formula

$$x\ .\ y := v\ .\ \ now < x + 3\ \ \mathcal{W}\ (v = y + 5\ \wedge\ now \leq x + 3)$$

states that the variable v will have increased by 5 within 3 time units,

We can define bounded variants of the "henceforth" and "sometimes" operators in terms of freeze quantification, as follows.

- $\Box_\tau \phi$, defined as $d := \tau.\ x.\ \Box\ (\phi\ \vee\ (now > x + d))$, states that ϕ is true from the current time instant onwards for at least τ units of absolute time. If τ is a constant, the definition can be simplified to $x.\ \Box\ (\phi \vee (now > x + \tau))$.
- $\Diamond_\tau \phi$, defined as

  $$d := \tau.\ x\ .\ (now < x + d)\ \mathcal{W}\ (\phi \wedge now \leq x + d)\ \ ,$$

  states that ϕ will be true at some present or future time instant within the next τ units of absolute time.

A formula is called a *past formula* if it has no future-time temporal operators, and a *future formula* if it has no past-time temporal operators.

Semantics over Infinite and Finite Behaviors Formally, the semantics of the operators presented above can be defined as in Table 1, which defines when an infinite behavior satisfies a temporal logic formula at a particular time instant. We say that a behavior σ satisfies a temporal formula ϕ is $\sigma(t)[\phi]$ is true, where t is the initial time instant.

The semantics in Table 1 is a standard way (as in, e.g., [16]) to elegantly define the meaning of temporal operators. However, it is not directly usable for generating test oracles, since an oracle will never "see" an entire infinite behavior, only a finite sequence of time instants, which can be regarded as a finite prefix of an infinite behavior. We shall therefore define when the oracle should report violations of a specifications on the basis of a finite observation.

Let a *behavior prefix* be a mapping from a finite prefix of the sequence of time instants to states. We say that a behavior σ *extends* a behavior prefix ρ if σ and ρ are equal on the domain of ρ.

Definition 1. A behavior prefix ρ *violates* a temporal formula ϕ for time instant t if there is no behavior σ, which extends ρ, for which $\sigma(t)[\phi]$ is true. A behavior prefix ρ *violates* a temporal formula ϕ if ρ violates ϕ for the initial time instant.

Intuitively, a behavior prefix violates a formula if it cannot be extended to a behavior that satisfies the formula. In such a situation, it seems reasonable to say that the formula is violated.

The logic defined in this paper is designed in order to express so-called *safety properties*; these are special in the the sense that if a behavior σ does not satisfy a safety property ϕ, then σ extends some behavior prefix that violates ϕ. Formally, a formula ϕ defines a *safety property* if whenever $\sigma(t)[\phi]$ is false for behavior σ and time instant t, then σ extends a behavior prefix ρ that violates ϕ for t. Intuitively this means that if a behavior does not satisfy a safety property ϕ, then ϕ will be violated within a finite number of time instants. This can be paraphrased by saying that a safety property is of the form "something bad will not happen"; here "bad" could be taken to denote "violation".

We intend to establish that all formulas expressible in our logic are safety properties. If we exclude quantification, this is a well-known result. To include quantification in the result, we must be a little more careful, since subformulas can have free variables that are bound in some outermore freeze quantification. We say that a formula ϕ with free variables is a safety property if each instance of ϕ is a safety property.

Theorem 1. *All formulas expressible in the logic of this section define safety properties.*

Proof. By structural induction over the formula. Let us consider some of the most interesting cases:

- Consider the formula $\Box\ \phi$, and assume that ϕ is a safety property. Assume that $\sigma(t)[\phi]$ is false. Then there must be some time instant $t' \geq t$ such that $\sigma(t')[\phi]$ is not true. Since ϕ is a safety property, ϕ is violated for t' by some behavior prefix ρ, which by definition then also violates $\Box\ \phi$ for t.
- Consider the formula $x := exp\ .\ \phi$, and assume that ϕ is a safety property. Assume that $\sigma(t)[x := exp\ .\ \phi]$ is false, i.e., $\sigma(t)[\phi[\sigma(t)[exp]/x]]$ is false. Since by assumption $\phi[\sigma(t)[exp]/x]$ is a safety property, it must be violated for t by some behavior prefix ρ, which by definition also violates $x := exp\ .\ \phi$ for t.

Other cases are treated in a similar manner.

We note that Theorem 1 crucially depends on the fact that the logic does not have negation, except in expressions. For instance, the formula $\neg\Box\ v \neq 8$ is not a safety property. It is satisfied by a behavior σ if σ has no state where $v = 8$. However, it is not possible to find any prefix of such a σ, which cannot be extended by some state where $v = 8$. For convenience, we can allow negation of past formulas, since the negation of a past formula can be rewritten into an equivalent past formula without negation outside a temporal operator using the equivalences

$$\begin{aligned} \Diamond_\tau \phi &\equiv \neg \Diamond_\tau \neg \phi \\ \phi\ \mathcal{B}\ \psi &\equiv \neg[\neg\psi\ \mathcal{S}\ (\neg\phi \wedge \neg\psi)] \\ \phi\ \mathcal{S}\ \psi &\equiv \neg[\neg\psi\ \mathcal{B}\ (\neg\phi \wedge \neg\psi)] \end{aligned}$$

We can then also allow implications $\phi_1 \implies \phi_2$ where ϕ_1 is a past formula.

4 Translating Temporal Logic Formulas to Test Oracles

In this section, we present a technique for translating temporal logic requirements into executable test oracles. The test oracles will be described on a general level, in terms of their state variables and how they are maintained. In Section 5, we will report from an application, where oracles are represented in a concrete language used in test equipment. The input to the translation is a temporal logic formula Φ which expresses a requirement of a system component. The translation shall generate a *test oracle*, i.e., an executable program module which observes the input and output variables of the system component, and reports when the observed behavior violates Φ. Violations could be reported, e.g., via a variable written by the test oracle and read by some entity which monitors the test execution.

As described in Section 2, we assume that the oracle observes the system periodically at the sequence of consecutive time instants. The oracle maintains information in local variables, but must not affect the variables of the tested component. At each time instant, it reads values of the (input and output) system variables of the tested component, and then uses these values, together with the values of its internal variables, to compute new values of its internal variables. We assume that the oracle is sufficiently fast to be able to execute its local computations generated at one time instant before the occurrence of the next time instant.

We assume that the requirement Φ to be checked satisfies the following properties.

- future or quantification operators must not occur inside past operators.
- a bound variable (in a freeze quantification) does not appear inside a past operator.

Our presentation of the oracle generation has three parts. We first describe how the oracle handles past subformulas. These can be handled separately, since they do not contain future operators or quantification. In Subsection 4.2 we describe how formulas without quantification are handled. Finally, in Subsection 4.3, we present how to extend the translation to consider formulas with quantification.

4.1 Handling of Past Subformulas

A general strategy of the oracle that checks Φ is to continuously maintain the values of all past subformulas of Φ. This is done by maintaining the following variables:

- For each past subformula ϕ of Φ which either has a past operator different from $\ominus$ as main connective, or such that $\ominus\ \phi$ is a subformula of Φ, the oracle has a local *log variable*, here denoted $log(\phi)$. Its value is continuously updated to be equal to $\sigma(t)[\phi]$ at any time instant t.

– For each subformula of form $\Diamond_\tau \phi$ the oracle has a local timer variable, denoted $timer(\phi)$, which intuitively measures for how long ϕ has been continuously true.

Let us now describe how the oracle updates timers and log variables.

Timers. Each timer of form $timer(\phi)$ is maintained by starting and resetting it. It is *inactive* whenever ϕ is false and *active* whenever ϕ is true.

– In the initial time instant, the timer $timer(\phi)$ is inactive if ϕ is false, and active with value 0 if ϕ is true.
– The timer $timer(\phi)$ is started whenever ϕ is true in the current time instant and false in the preceding.
– The timer $timer(\phi)$ is reset and stopped when ϕ is false in the present time instant, and true in the preceding, thus becoming inactive.

Proposition 1. *Under the above rules, we can infer that whenever ϕ is true at time instant t, then the timer $timer(\phi)$ is active and satisfies*

$$\sigma(t')[now] + \sigma(t)[timer(\phi)] = \sigma(t)[now] \quad ,$$

where t' is the earliest time instant such that ϕ is true at all time instants from t' up to and including t.

Log Variables To maintain correct values of log variables, the oracle should at each time instant update the value of each variable $log(\phi)$ so that its value at time instant t becomes $\sigma(t)[\phi]$. This is done by assigning $log(\phi)$ the value of the expression $\chi(\phi)$, which is defined recursively below. Intuitively, $\chi(\phi)$ expresses the current value of ϕ in terms of current values of system variables and previous values of subformulas of ϕ.

$$\begin{array}{ll}
\chi(\phi) & = \phi \quad \text{if } \phi \text{ has no temporal operators} \\
\chi(\phi_1 \wedge \phi_2) & = \chi(\phi_1) \wedge \chi(\phi_2), \\
\chi(\phi_1 \vee \phi_2) & = \chi(\phi_1) \vee \chi(\phi_2). \\
\chi(\ominus\, \phi) & = log^-(\phi) \\
\chi(\phi\ \mathcal{S}\ \psi) & = \chi(\psi) \ \vee\ \big(\chi(\phi) \wedge log^-(\phi\ \mathcal{S}\ \psi)\big) \\
\chi(\phi\ \mathcal{B}\ \psi) & = \chi(\psi) \ \vee\ \big(\chi(\phi) \wedge log^-(\phi\ \mathcal{B}\ \psi)\big) \\
\chi(\boxminus\, \phi) & = \chi(\phi) \wedge log^-(\boxminus\, \phi) \\
\chi(\Diamond\, \phi) & = \chi(\phi) \vee log^-(\Diamond\, \phi) \\
\chi(\Diamond_\tau \phi) & = \chi(\phi) \wedge timer(\phi) \geq \lfloor \tau \rfloor \\
\chi(\Diamond_\tau \phi) & = \chi(\phi) \vee timer(\neg\phi) < \lfloor \tau \rfloor
\end{array}$$

Here $log^-(\phi)$ denotes the value of $log(\phi)$ in the previous time instant, and $\lfloor \tau \rfloor$ denotes the largest multiple of the time between successive time instants, which is less than or equal to τ.

Using the above definitions, the value of $log(\phi)$ at time instant t can be obtained from system variables and timers at time instant t and from log variables at time instant t^-. In the oracle code, care should be taken so that at each time instant, all uses of a log variable (e.g., in right-hand sides of assignments) are performed before the log variable is updated. The assignments can be guarded by a boolean expression which is false when $log(\phi)$ need not be changed.

The initial values for each log variable $log(\phi)$ is defined by the expression $\mathcal{I}(\phi)$, which is defined as follows.

– $\mathcal{I}(\phi) = \phi$ if ϕ has no temporal operators,
– $\mathcal{I}(\phi_1 \wedge \phi_2) = \mathcal{I}(\phi_1) \wedge \mathcal{I}(\phi_2)$,
– $\mathcal{I}(\phi_1 \vee \phi_2) = \mathcal{I}(\phi_1) \vee \mathcal{I}(\phi_2)$.
– $\mathcal{I}(\ominus\, \phi) = \mathcal{I}(\phi)$
– $\mathcal{I}(\phi\ \mathcal{S}\ \psi) = \mathcal{I}(\psi)$
– $\mathcal{I}(\phi\ \mathcal{B}\ \psi) = \mathcal{I}(\phi \vee \psi)$
– $\mathcal{I}(\boxminus\, \phi) = \mathcal{I}(\phi)$
– $\mathcal{I}(\Diamond\, \phi) = \mathcal{I}(\phi)$
– $\mathcal{I}(\Diamond_\tau \phi) = \mathcal{I}(\phi)$
– $\mathcal{I}(\Diamond_\tau \phi) = \mathcal{I}(\phi)$

Proposition 2. *With the previous rules for maintaining log variables and timers, we have after the updates at time instant t*

$$\sigma(t)[log(\phi)] = \sigma(t)[\phi]$$

for each log variable $log(\phi)$.

Proof. By induction over successive time instants, checking that the definitions for each operator in terms of $\chi(\phi)$ is correct.

Example 1. As illustration, we will consider two requirements taken from the cruise control module case study. The first requirement states that

$$(ccont \wedge \neg\ominus\, \neg ccont) \implies \ominus\, (\neg ccont\ \mathcal{S}\ cca)$$

should hold at all time instants, i.e., that the signal *ccont* can become true only if *cca* has been true at least once since *ccont* was true the last time. The requirement is checked by introducing the log variables $log(\neg ccont)$ and $log(\neg ccont\ \mathcal{S}\ cca)$, which are updated using the rules:

$$\begin{array}{lcl}
log(\neg ccont) & = & \neg ccont \\
log(\neg ccont\ \mathcal{S}\ cca) & = & cca \ \vee\ (\neg ccont \wedge log^-(\neg ccont\ \mathcal{S}\ cca)
\end{array}$$

A violation is reported if the expression

$$(ccont \wedge \neg log^-(\neg ccont)) \implies log^-(\neg ccont\ \mathcal{S}\ cca)$$

evaluates to false.

Example 2. Consider the requirement that at all time instants

$$\Diamond_{1000}[\mathrm{abs}(vs - vsa)/vs > 0.05] \implies \neg cca \quad ,$$

stating that if two independent measurements, *vs* and *vsa*, of the vehicle speed for at least 1000 time units differ more than 5% (with reference to *vs*), then the signal *cca* must be false. A false value of *cca* will disable the automatic cruise control, motivated by the fact that the deviation between *vs* and *vsa* indicates a potential problem.

The generated oracle has a timer variable, named $timer(\text{abs}(vs - vsa)/vs > 0.05)$ which is started whenever $\text{abs}(vs - vsa)/vs > 0.05$ becomes true. It will report a violation whenever the boolean expression

$$\neg(timer(\text{abs}(vs - vsa)/vs > 0.05) \geq \lfloor 1000 \rfloor) \ \vee \ \neg cca$$

becomes false.

4.2 Formulas without Quantification

In this subsection, we assume that there is no freeze quantification in Φ. By maintining the values of past-time subformulas in log variables, as described in Subsection 4.1, we need only be concerned with the problem of handing future-time operators, $\circ$, $\mathcal{W}$, and $\Box$. The strategy used by the oracle to detect violations of Φ will be to continuously maintain a "derivative" of Φ, which represents the requirement on the remaining behavior, given what has been observed so far. At initialization, this requirement is equivalent to Φ. At each time instant t, the oracle updates the current derivative of Φ to represent the requirement on the computation from time instant t^+ and onwards, given that the sequence of states from the initial time instant up to t has been observed. This update is based on system and log variables at time instant t and on the previous value of the derivative. When the derivative becomes unsatisfiable (equivalent to false), the oracle reports a violation of Φ.

Our mechanism for maintaining derivatives of Φ builds on the correspondence between temporal logic and automata [26,17], using the fact that any formula ϕ in propositional temporal logic can be represented by an automaton over infinite words, whose runs correspond to the set of behaviors that satisfy ϕ. Each state of such an automaton can roughly be considered as a derivative of ϕ, which represents the requirement on the remaining behavior, given what has been observed so far. The standard construction of an automaton (e.g., in [26]) in general yields a nondeterministic automaton. Since the test oracle ought to be deterministic, we can either determinize this automaton, or (as we will do in this paper) keep the automaton nondeterministic and maintain the derivative as a *set* of possible current automaton states, thus performing a subset-construction "on-the-fly".

Let us now describe the oracle construction more precisely. Let Φ be the requirement to be checked. Define a core subformula of Φ as a formula ϕ which either has $\mathcal{W}$ or $\Box$ as its main operator, or for which $\circ\,\phi$ is a subformula of Φ. We transform Φ into a (possibly nondeterministic) automaton, in which each state (henceforth called *node*, to avoid confusion with states of the tested system) corresponds to a conjunction $\bigwedge_j \phi_j$ of core subformulas ϕ_j. For each node, the oracle has a boolean variable, here denoted $at(\bigwedge_j \phi_j)$. At each time instant t, these variables are updated so that the requirement on the computation from time instant t^+ onwards is equal to the disjunction of all conjunctions $\bigwedge_j \phi_j$ for which $at(\bigwedge_j \phi_j)$ is true.

The rules for updating the variables of form $at(\bigwedge_j \phi_j)$ are obtained from the construction for transforming a temporal logic formula into an automaton, as follows.

For each conjunction $\bigwedge_j \phi_j$, we find an equivalent formula of form

$$\bigwedge_j \phi_j \quad \equiv \quad \bigvee_i (p_i \ \wedge \ \circ \bigwedge_k \phi_{ik})$$

where each p_i is a past formula, and each $\bigwedge_k \phi_{ik}$ is a conjunction of core subformulas of Φ. The rewriting is carried out by using the following equivalences to decompose each core subformula into two parts: one which states properties about the past and present behavior, and one which describes the behavior from the next time instant and onwards.

$$\phi \ \mathcal{W} \ \psi \equiv \psi \ \vee \ (\phi \ \wedge \ \circ \ (\phi \ \mathcal{W} \ \psi))$$
$$\Box \ \phi \quad \equiv \phi \ \wedge \ \circ \Box \ \phi$$

To motivate the rule for $\mathcal{W}$, we note that a formula of form $\phi \ \mathcal{W} \ \psi$ is equivalent to saying that either ψ is true now, or that ϕ is true now and $\phi \ \mathcal{W} \ \psi$ is true in the next time instant. After using the above rules, the formula $\bigwedge_j \phi_j$ should have been rewritten into a boolean combinations of formulas, each of which is either a past formula or has $\circ$ as its main connective. The result is rearranged into a disjunction of conjunctions, and finally the $\circ$ operators are moved outside conjunctions, if necessary, to arrive at the above form.

Using the equivalence

$$\bigwedge_j \phi_j \quad \equiv \quad \bigvee_i (p_i \ \wedge \ \circ \bigwedge_k \phi_{ik})$$

the rules for maintaining the control variables of the oracle can be formulated as follows.

- The initial values of the variables $at(\bigwedge_j \phi_j)$ are obtained by transforming Φ into a disjunction of conjunctions. This represents "pre-initial" requirements, which should be updated by using the values of system and log variables at the first time instant.
- At each non-initial time instant t, the value of each variable of form $at(\bigwedge_k \phi_{ik})$ should be true if there is some conjunction $\bigwedge_j \phi_j$, decomposed as

  $$\bigwedge_j \phi_j \quad \equiv \quad \bigvee_i (p_i \ \wedge \ \circ \bigwedge_k \phi_{ik})$$

  such that $at(\bigwedge_j \phi_j)$ is true at time instant t^- and p_i is true at time instant t. If there is no such conjunction, then $at(\bigwedge_k \phi_{ik})$ should be false at time instant t. If t is the initial time instant, the rule is the same, except that $at(\bigwedge_j \phi_j)$ should be true in the intialized state of the oracle.

– The oracle will report a violation whenever all boolean variables of form $at(\bigwedge_j \phi_j)$ are false.

The relationship between the control variables and the requirement on the remaining computation is captured in the following characterization.

Proposition 3. *Assume the previous rules for maintaining log variables, timers, and control variables. Then a behavior σ satisfies requirement Φ if and only if at time instant t it satisfies*

$$\bigvee_i (\circ \bigwedge_k \phi_{ik})$$

where the disjunction is taken over all i such that the variable $at(\bigwedge_k \phi_{ik})$ is true at time instant t.

Proof. By induction over successive time instants. Using the rules for decomposition, the derivative at time instant t is decomposed, and disjunctions whose past-part is not satisfied by the system and log variables in the current time instant are discarded.

Since the number of boolean variables of form $at(\bigwedge_j \phi_j)$ is bounded by the number of possible combinations of subformulas of Φ, there may in the worst case be an exponential number of such variables. In practice, this is not a big obstacle, since requirements are usually rather small, and since many of the variables $at(\bigwedge_j \phi_j)$ can often be discarded, using a reachability analysis on the rules for updating, since they can never become true in any execution.

Example 3. The requirement

$$\Phi \equiv \Box\ (ccanc \implies \circ\ \neg cca)$$

states that whenever the button *cccanc* is pressed (i.e., the corresponding signal becomes true), the signal *cca* must be set to false, meaning that it will be false in the next time instant.

We see that there are three possible boolean control variables: $at(\Phi)$, $at(\neg cca \wedge \Phi)$, and $at(\neg cca)$. From the decompositions

$$\begin{aligned} \Phi \quad &\equiv \quad (\neg ccanc \wedge \circ\ \Phi) \\ &\quad\ \vee\ (\circ\ (\neg cca \wedge \Phi)) \\ \neg cca \wedge \Phi \equiv \quad &(\neg cca \wedge \neg ccanc \wedge \circ\ \Phi) \\ &\vee\ (\neg cca \wedge \circ\ (\neg cca \wedge \Phi)) \end{aligned}$$

we see that the variable $at(\neg cca)$ is unreachable since there is no next-part of form $\circ\ \neg cca$ in the above decompositions. The rules for updating the other variables are as follows.

– $at(\Phi)$ is true if either it was true in the preceding instant and *ccanc* is false, or $at(\neg cca \wedge \Phi)$, was true in the preceding instant and $\neg cca \wedge \neg ccanc$ is true.
– $at(\neg cca \wedge \Phi)$ is true if either it was true in the preceding instant and *cca* is false, or $at(\Phi)$ was true in the preceding instant.

A violation is reported if $at(\neg cca \wedge \Phi)$ was true in the preceding instant, and *cca* is true in the present.

4.3 Formulas with Quantification

Let us now consider how to extend the construction in the previous subsection to formulas with quantification. A future-time core subformula ϕ of Φ may then have free variables, which are bound in some scope that includes ϕ. We can think of these free variables as parameters of ϕ. Whenever a control variable of form $at(\bigwedge_j \phi_j)$ is true, the oracle should therefore also maintain actual values of the parameters of each ϕ_j in the conjunction $\bigwedge_j \phi_j$. Thus, for each conjunction $\bigwedge_j \phi_j$ the oracle maintains a local variable for each parameter in some conjunct of $\bigwedge_j \phi_j$. When $at(\bigwedge_j \phi_j)$ is true, the values of these variables represent the "actual" values of the parameters of $\bigwedge_j \phi_j$. When $at(\bigwedge_j \phi_j)$ is false, the values of these variables is irrelevant.

Unfortunately, this extension is in general not sufficient to represent the requirement on the remaining behavior. Two complications are the following.

– The requirement on the remaining computation may need to use conjunctions in which the same core subformula will appear several times, with different actual parameter values. As an example, let the original requirement be

$$\Phi \equiv \Box\ x := v\ .\ \Box\ u \neq x\ \ ,$$

stating that the value of the system variable u is never equal to a present or past value of v. Assuming that the successive values of v are $v_0, v_1, \ldots$ the requirement on the remaining behavior in the second time instant may become

$$\Phi\ \wedge\ \Box\ u \neq v_0\ \wedge\ \Box\ u \neq v_1\quad .$$

To represent this requirement, we need two instances of the subformula $\Box\ u \neq x$. Continuing this example, we see that there is in general no bound on the number of instances of a particular subformula that may be needed in a conjunction.
– A similar problem arises when the requirement on the remaining computation needs to use a disjunction of two instances of the same conjunction of core subformulas. As an example, consider the requirement

$$\Phi \equiv u \neq 0\ \mathcal{W}\ (x := v\ .\ u \neq 0\ \mathcal{W}\ u = x)\ \ ,$$

stating that u must be different from 0 until it becomes equal to some past value of v. The requirement on the remaining behavior in the second time instant may become

$$\Phi\ \vee\ u = v_0\ \vee\ u = v_1\ \ .$$

In order to represent such a requirement in the current scheme, we would need $at(u = x)$ to be true with both the value v_0 and the value v_1 for the parameter x.

If problems of these types arise, we will handle them by weakening the (representation of the) requirement on the remaining behavior. It is then possible that some violations are not reported, but the oracle will not report false violations. This is done as follows.

- In cases where two different instances of a conjunct would be needed in a conjunction, one of the instances is dropped from the conjunction
- In cases where two different instances of a conjunction would be needed (corresponding to two disjuncts in the represented requirement), we replace by *true* those conjuncts that are instantiated differently. In cases where an entire conjunction is replaced by *true* the oracle will allow any future behavior, and thus has no further function.

In some cases, we can avoid these drastic measures by a semantic analysis of the requirement. In Subsection 4.4, we present a simple subsumption mechanism, based on monotonicity, to handle several commonly occuring cases.

We now present the modified oracle construction in more detail. As already stated, for each conjunction $\bigwedge_j \phi_j$ of core subformulas of the original requirement Φ, the oracle has a variable $at(\bigwedge_j \phi_j)$ and variables that correspond to the free variables (parameters) of the conjuncts of $\bigwedge_j \phi_j$. The intention is that at each time instant t, the requirement on the computation from time instant t^+ and onwards is equal to

$$\bigvee_i (\circ \bigwedge_k \phi_{ik}[\overline{d}_i/\overline{x}_i]) \quad ,$$

where the disjunction is taken over all i such that where the disjunction is taken over all i such that $at(\bigwedge_k \phi_{ik})$ is true at time instant t, and where $\overline{d}_i$ are the current values of the parameters $\overline{x}_i$ of $\bigwedge_k \phi_{ik}$.

The mechanism for updating the oracle variables is obtained by transforming each conjunction $\bigwedge_j \phi_j$ into an equivalent formula as follows

$$\bigwedge_j \phi_j \quad \equiv \quad \bigvee_i (p_i \ \wedge \ \circ \bigwedge_k \phi_{ik} \ \wedge \ \theta) \quad ,$$

where p_j is a past formula, each $\bigwedge_k \phi_{ik}$ is a conjunction of core subformulas of Φ, and where θ is a "substitution" of form $x_1 = e_1 \wedge \cdots \wedge x_m = e_m$, which states how variables $x_1, \ldots, x_m$ which are free in $\bigvee_i (p_i \ \wedge \ \circ \bigwedge_k \phi_{ik})$ but still bound in $\bigwedge_j \phi_j$, correspond to expressions $e_1, \ldots, e_m$ over free variables of $\bigwedge_j \phi_j$. The transformation is carried out using the same equivalences as before, i.e., the rules

$$\begin{array}{ll} \phi \ \mathcal{W} \ \psi \equiv \psi \ \vee \ (\phi \ \wedge \ \circ \ (\phi \ \mathcal{W} \ \psi)) \\ \Box \ \phi \quad \equiv \phi \ \wedge \ \circ \Box \ \phi \end{array}$$

plus the following rule for quantification

$$x := exp \ . \ \phi \wedge \theta \ \equiv \ \phi \ \wedge \ (\theta \wedge x = exp\theta) \quad ,$$

where $exp\theta$ denotes the result of replacing in exp each left-hand side x_i of θ by a corresponding right- hand side e_i of θ. This results in extending the current substitution θ into $(\theta \wedge x = exp\theta)$.

If in some decomposition

$$\bigwedge_j \phi_j \quad \equiv \quad \bigvee_i (p_i \ \wedge \ \circ \bigwedge_k \phi_{ik} \ \wedge \ \theta) \quad ,$$

some conjunction $\bigwedge_k \phi_{ik}$ contains several occurrences of the same subformula (with different parameters), then all but one of these conjuncts are dropped. As a result, the right-hand side of the equivalence may become weaker than the left-hand side.

The rules for maintaining oracle variables are now formulated as follows.

- The initial values of the variables $at(\bigwedge_j \phi_j)$ are obtained by transforming Φ into a disjunction of conjunctions. Note that these conjunctions have no free parameters.
- At each non-initial time instant t, and for each control variable $at(\bigwedge_k \phi_{ik})$, consider all decompositions of form

  $$\bigwedge_j \phi_j \quad \equiv \quad \bigvee_i (p_i \ \wedge \ \circ \bigwedge_k \phi_{ik} \ \wedge \ \theta) \quad ,$$

  such that $at(\bigwedge_j \phi_j)$ is true at time instant t^- and p_i is true at time instant t, where the values of non-system variables in p_i are obtained using the substitution θ and current values of the parameters of $\bigwedge_j \phi_j$.
  - If $at(\bigwedge_k \phi_{ik})$ occurs only in one such decomposition, then $at(\bigwedge_k \phi_{ik})$ should be true at time instant t. The appropriate values of the parameters of $\bigwedge_k \phi_{ik}$ are obtained from the parameters of $\bigwedge_j \phi_j$ via the substitution θ.
  - If $at(\bigwedge_k \phi_{ik})$ does not occur in such a decomposition, then $at(\bigwedge_k \phi_{ik})$ should be false at time instant t.
  - If $at(\bigwedge_k \phi_{ik})$ occurs in several such decompositions, for which the assignments to parameters would be inconsistent, then the conjuncts that are affected by inconsistent assignments are removed (replaced by *true*). If the resulting conjunction $\bigwedge_k \phi_{i'k}$ does not appear in another such decomposition, then $at(\bigwedge_k \phi_{i'k})$ should become true, otherwise the pruning of inconsistently assigned conjuncts continues.
- The oracle will report a violation whenever all boolean variables of form $at(\bigwedge_j \phi_j)$ are false.

As an illustration, consider the requirement

$$\Phi \equiv \Box \ x := v \ . \ \Box \ u \neq x$$

which states that for each time instant, there is never a present or future time instant at which the value of u is equal to the current value of v. Let the successive values of v be $v_0, v_1, \ldots$. The decomposition of Φ is

$$\Phi \equiv u \neq x \wedge \circ(\Box\, u \neq x \wedge \Phi) \wedge x = v \quad .$$

The decomposition of $\Box\, u \neq x \wedge \Phi$ is (after renaming the bound variable in Φ to y to avoid name clashes)

$$\Phi \equiv \begin{array}{l} u \neq x \wedge u \neq y \\ \wedge\ \circ(\Box\, u \neq x \wedge \Box\, u \neq y \wedge \Phi) \\ \wedge\ y = v \end{array} \quad .$$

To avoid the explosion of the formula, we weaken the conjunction inside the $\circ$ operator, e.g., to become

$$\Phi \implies \begin{array}{l} u \neq x \wedge u \neq y \\ \wedge\ \circ(\Box\, u \neq x \wedge \Phi) \\ \wedge\ y = v \end{array} \quad .$$

An oracle that uses this decomposition, will in effect perform the weaker check that u is never equal to the first or current value of v.

Proposition 4. *Assume the previous rules for maintaining log variables, timers, control variables, and quantification. Then a behavior σ satisfies requirement Φ also satisfies*

$$\bigvee_i (\circ \bigwedge_k \phi_{ik})$$

at any time instant t, where the disjunction is taken over all i such that $at(\bigwedge_k \phi_{ik})$ is true at time instant t.

4.4 Simplification by Monotonicity

In this subsection, we will propose a simple mechanism which in some cases can avoid the weakening of the requirement incurred by removing duplicate conjuncts, as described in Section 4.3, which is based on a simple subsumption mechanism.

Let $\leq$ be the standard ordering on real numbers and integers, and the ordering *false* $\leq$ *true* on booleans. We will call an expression *increasing* or *decreasing* wrp. to an argument, if its value is increasing (decreasing) wrp. to that argument. For instance,

- $x > 7$ is increasing in x, and
- $x \leq 7$ is decreasing in x.

We observe that the future-time temporal operators are monotonic in all of their arguments, i.e., if the argument becomes larger (i.e., "more true"), then the formula becomes larger. The same is true for conjunction and disjunction.

Monotonicity can be used for simplification of conjunctions and disjunctions as follows. If $\phi[x]$ is a formula which is increasing in x, then we can transform conjunctions and disjunctions as follows.

$$\begin{array}{rcl} \phi[x] \wedge \phi[y] & \equiv & \phi[\min(x,y)] \\ \phi[x] \vee \phi[y] & \equiv & \phi[\max(x,y)] \end{array}$$

Let us consider some examples.

Example 4. As a first example, consider the requirement $\Box\, b \implies \Box_5 c$, stating that whenever the boolean system variable b is true, then the variable c should be true for at least 5 time units. The requirement is defined in terms of freeze quantification as

$$\Phi \equiv \Box\ (b \implies x.\ \Box\ (c \vee (now > x + 5))) \quad .$$

A decomposition of Φ yields

$$\begin{array}{l} \neg b \wedge \circ\Phi \\ \vee \left[\begin{array}{l} (c \vee (now > x + 5)) \\ \wedge\ \circ\ (\Box\ (c \vee (now > x + 5)) \wedge \Phi) \\ \wedge\ x = now \end{array}\right] \end{array} ,$$

A further decomposition of $\Box\ (c \vee (now > y + 5)) \wedge \Phi$ (where we have renamed the free variable x to y to avoid name clashes) yields

$$\begin{array}{l} \left[\begin{array}{l} \neg b \wedge (c \vee (now > y + 5)) \\ \wedge\ \circ\ (\Box\ (c \vee (now > y + 5)) \wedge \Phi) \end{array}\right] \\ \vee \left[\begin{array}{l} (c \vee (now > y + 5)) \\ \wedge\ (c \vee (now > x + 5)) \\ \wedge\ \circ \left[\begin{array}{l} \Box\ (c \vee (now > y + 5)) \\ \wedge\ \Box\ (c \vee (now > x + 5)) \\ \wedge\ \Phi \end{array}\right] \\ \wedge\ x = now \end{array}\right] \end{array} .$$

The expression $now > x + 5$ is decreasing in x, hence we can use monotonicity to simplify the formula into

$$\begin{array}{l} \left[\begin{array}{l} \neg b \wedge (c \vee (now > y + 5)) \\ \wedge\ \circ\ (\Box\ (c \vee (now > y + 5)) \wedge \Phi) \end{array}\right] \\ \vee \left[\begin{array}{l} (c \vee (now > \max(x,y) + 5)) \\ \wedge\ \circ\ (\Box\ (c \vee (now > \max(x,y) + 5)) \wedge \Phi) \\ \wedge\ x = now \end{array}\right] \end{array} .$$

As a further optimization, we could use the observation that since y has previously been bound in a freeze quantification, a free occurrence of y will always be bound to a past value of *now*, implying that the expression $\max(x,y)$ is always equal to x.

Example 5. An example requirement from the cruise control module is

$$\Phi \equiv \Box \left[\begin{array}{l} (\ominus\, \neg igsw \wedge igsw) \vee (\ominus\, igsw \wedge \neg igsw)] \\ \implies \\ \Diamond_{\mathtt{d}} \neg ccont \end{array}\right] ,$$

stating that at an edge of the signal *igsw*, the signal *ccont* must be set to false within `d` time units. Let us use $\phi[x]$ to denote

$$(now < x + \mathtt{d})\ \mathcal{W}\ (\neg ccont \wedge (now \leq x + \mathtt{d})) \quad .$$

Let us also denote $\ominus\ (\neg igsw \wedge igsw) \vee (\ominus\ igsw \wedge \neg igsw)$ by $\ominus\ igsw \neq igsw$, denote $\neg(\ominus\ igsw \neq igsw)$ by $\ominus\ igsw = igsw$, and denote $\neg ccont \wedge (now \leq x + \mathtt{d})$ by $\psi[x]$. The

requirement can then be expressed in terms of freeze quantification as

$$\Phi \equiv \Box\,[\ominus\, igsw \neq igsw \implies x\,.\,\phi[x]]\ .$$

The occurrences of the past operator $\ominus$ cause the oracle to have a variable $log(igsw)$ to record values of $igsw$. The relevant control variables will be of form $at(\Phi)$ and $at(\Phi \wedge \phi[y])$, where the second conjunct has a parameter y. Decompositions of these conjunctions become

$$\begin{aligned}
\Phi \quad &\equiv \quad \ominus\, igsw = igsw \vee \psi[x] \wedge \circ\, \Phi \wedge x = now \\
&\quad\ \vee\ (now < x + \mathtt{d}) \wedge \circ\, (\Phi \wedge \phi[x]) \wedge x = now \\
\Phi \wedge \phi[y] &\equiv \quad (\ominus\, igsw = igsw \vee \psi[x]) \wedge \psi[y] \\
&\quad\ \wedge \circ\ \Phi \wedge x = now \\
&\quad\ \vee\ \ (now < x + \mathtt{d}) \wedge \psi[y] \\
&\quad\ \wedge \circ\ (\Phi \wedge \phi[x]) \wedge x = now \\
&\quad\ \vee\ \ (\ominus\, igsw = igsw \vee \psi[x]) \wedge (now < y + \mathtt{d}) \\
&\quad\ \wedge \circ\ (\Phi \wedge \phi[y]) \wedge x = now \\
&\quad\ \vee\ \ (now < x + \mathtt{d}) \wedge (now < y + \mathtt{d}) \\
&\quad\ \wedge \circ\ (\Phi \wedge \phi[x] \wedge \phi[y]) \wedge x = now
\end{aligned}$$

By monotonicity, $\phi[x] \wedge \phi[y]$ is equivalent to $\phi[\min(x, y)]$, and even to $\phi[y]$ observing that we always have $y < x$ in the above situation. The same holds for $\psi[x]$. We also note that we can simplify $(now < x + \mathtt{d})$ to *true* if $x = now$. After simplification, we get

$$\begin{aligned}
\Phi \quad &\equiv \quad \ominus\, igsw = igsw \vee \psi[x] \wedge \circ\, \Phi \wedge x = now \\
&\quad\ \vee \circ\ (\Phi \wedge \phi[x]) \wedge x = now \\
\Phi \wedge \phi[y] &\equiv \quad (\ominus\, igsw = igsw \vee \psi[x]) \wedge \psi[y] \\
&\quad\ \wedge \circ\ \Phi \wedge x = now \\
&\quad\ \vee\ \ \psi[y] \\
&\quad\ \wedge \circ\ (\Phi \wedge \phi[x]) \wedge x = now \\
&\quad\ \vee\ \ (now < y + \mathtt{d}) \\
&\quad\ \wedge \circ\ (\Phi \wedge \phi[y]) \wedge x = now
\end{aligned}$$

The rules for updating the control variables become

- $at(\Phi)$ should be true if either $at(\Phi)$ in the preceding instant and $\ominus\, igsw = igsw \vee \psi[now]$ is true in the present, or $at(\Phi \wedge \phi[y])$ was true in the preceding instant, and $(\ominus\, igsw = igsw \vee \psi[now]) \wedge \psi[y]$ is true presently.
- $at(\Phi \wedge \phi[y])$ should be true if either $at(\Phi)$ in the preceding instant, or if $at(\Phi \wedge \phi[y])$ was true in the preceding instant, and either $\psi[y]$ or $now < y + \mathtt{d}$ is true presently; the new value of y is set to now if $\psi[y]$ is true, otherwise y is unchanged.

An example where monotonicity cannot be applied is the formula,

$$\Box \left[\begin{array}{l} \left[\begin{array}{l} \ominus\, \neg resume \\ \wedge\ resume \\ \wedge\ speed < ref \end{array} \right] \\ \implies \\ v := speed.\ x. \\ \quad now < x + 2(ref - v) \\ \quad \mathcal{W} \\ \quad \left[\begin{array}{l} speed = ref \\ \wedge \\ \dfrac{speed - v}{now - x} = 0.5 km/h/s \end{array} \right] \end{array} \right]$$

expressing that when a positive edge of the input signal *resume* occurs and *speed* is lees than *ref*, the output value *speed* must start to increase until it reaches the value *ref*. Between the start and the end point, the increase should correspond to an acceleration of 0.5 km/h/s. This formula is neither increasing nor decreasing in x and v, due to the last subformula which expresses the average acceleration over the time period at which the variable *speed* attains the value *ref*.

4.5 Correctness

In this section, we consider soundness of the generated oracle, meaning that it will not report false violations, and completenss, meaning that violations are reported within some reasonable delay.

Theorem 2 (Soundness). *Whenever the oracle generated to check a formula Φ has assigned all boolean variables of form $at(\bigwedge_j \phi_j)$ to false, then the observed behavior violates Φ.*

Proof. (Sketch) Follows from Proposition 4

It is more complicated to give a theorem about completeness of the oracle, saying that the oracle will always report violations as soon as they occur, or after some delay. In view of the weakening of requirements that include quantification, we will state a completeness result only for unquantified formulas. We will further restrict the treatment by assuming that there is an upper bound K on the values that may be assumed by the distance term τ in operators $\boxdot_\tau$ and $\diamondsuit_\tau$.

To check whether the original requirement Φ has been violated at a time instant t is essentially the problem of checking whether the current derivative of Φ is satisfiable, i.e., whether there is an infinite continuation of the observed behavior that satisfies the current derivative. In the case where all system variables are boolean, this problem is NP-hard. Thus, to check whether Φ has been violated at t we can either run a satisfiability check in exponential time, or wait until all variables of form $at(\bigwedge_j \phi_j)$ become false at some later time instant. That this happens is guaranteed by the following theorem.

Theorem 3 (Completeness). *If an observed behavior σ violates the requirement Φ at time instant t^- then a violation will be reported at most $2^{|\Phi|} * (K/\epsilon) * m$ time instants later, where $|\Phi|$ is the size of Φ, m is the number of $\boxdot_\tau$ or $\diamondsuit_\tau$ operators, and ϵ is the distance in time between consecutive time instants.*

Proof. (Sketch) If σ violates the requirement Φ at time instant t^-, then there cannot be two later time instants k and l with $t \leq k < l$ at which the derivatives are equivalent (meaning that the values of log variables and control variables are equal), otherwise the behavior which indefinitely repeats the loop between time instants k and l

will satisfy Φ. A bound on the number of time instants until σ becomes *false* is therefore obtained as the number of possible inequivalent derivatives of Φ. Each relativization of Φ is equivalent to a positive boolean combination of subformulas of Φ, and combination of truth values of log variables $log(\phi)$ for each past subformula ϕ of Φ, and some value of $\lfloor \tau \rfloor$ for a distance term τ in the formula.

5 Short Description of FIL

In Section 4, we presented a translation from temporal logic to test oracles represented in an abstract guarded-command language. In the case studies, the oracles were realized in the language FIL, which has been designed specifically for testing components of embedded systems, in particular in automotive applications.

A FIL program executes cyclically, as a control loop. Each iteration of the basic control loop

1. reads values of system variables of the component under test (these can be considered as input variables of the tester), and values of local variables of the tester,
2. then performs internal calculations, and
3. finally provides values of input variables of the component under test, and new values of updated local variables.

The duration of each control loop may vary, depending on how fast these operations can be performed. The duration of a basic control loop can be adjusted (above a minimum) to the system under test by adding a suitable delay at the end of each control loop.

A typical FIL program defines a set of "experiments". Each experiment consists of injecting one or several faults, defined by corresponding fault definitions, and then observing the output of the component. Typically, this output is inspected manually, or observed indirectly through the behavior of the car. An automatically generated test oracle makes it possible to automate checking that the output conforms to stated safety requirements.

We will illustrate the style of FIL programs by an example, showing the FIL program resulting from the requirement of Example 5 in Section 4.4.

$$\Phi \equiv \Box \left[\begin{array}{l} \ominus\, (\neg igsw \wedge igsw) \vee (\ominus\, igsw \wedge \neg igsw)] \\ \Longrightarrow \\ \Diamond_{\texttt{deadline}} \neg ccont \end{array} \right],$$

stating that at an edge of the signal *igsw*, the signal *ccont* must be set to false within `deadline` time units.

A FIL representation of the generated oracle is shown in Figure 2. In this program, the first (noncomment) line defines Boolean as an enumerated type. The next two lines define some fault, which will be input to the system on test. The fault can be replaced by an arbitrary definition of faults that will be exercised during testing. The selection and definition of these faults is not in the scope of this paper. The next three lines define the signal *err* as an output of the oracle that will be used to report violations of the requirement. The test oracle is defined in the section that begins with **experiment-def**. The variable `prev_igsw` represents the log variable $log(igsw)$. The variable `r1_x` represents the parameter x of the parameterized formula $\phi[x]$. The variable `r1_time` is used to give diagnostic information in an error report. The variable `r1_active` represents that control is at variable $at(\Phi \wedge \phi[x])$ and not at variable $at(\Phi)$.

The log variable `prev_igsw` is updated in a **when-do** clause. The subsequent **when-do** clause represent making the variable $at(\Phi)$ false. The following **when-do** clause represent the case when the deadline is missed and a violation is detected. The last **when-do** clause represent a transition of control from $at(\Phi \wedge \phi[x])$ to $at(\Phi)$. The cases when control is unchanged is implicitly represented by the absence of a corresponding **when-do** clause.

6 Applications to a Case Study

To investigate the practical usefulness of the generated test oracles, we implemented the translation and applied it to temporal logic requirements of two components that were specified by Volvo. One component is a cruse control module, which had earlier been specified in TRIO by Nielsen at Prover Technology AB [19], and the other was a module for throttle control.

6.1 The Cruise Control Module

We made a specification in our temporal logic of the Cruise Control Module (CCM), containing 36 requirements. These formulas were based on requirements formulated in natural language. The fact that we were able to capture an existing set of requirements in our subset of temporal logic can be taken as evipdence for the usefulness of this subset. One aspect that could not be handled well is examplified in a simplified form in the requirement

> When *some condition*, then signal *ccws* must start to follow a curve towards *ccsp*, which increases smoothly from the starting point and also connects smoothly to *ccsp*.

A requirement of this form is hard to formalize in logic. In principle, one could envisage the construction of a test oracle which continuously monitors the signal *ccws*, and is able to observe a sufficient degree of "smoothness". However, our logic seems not appriate for expressing the functionality of such an oracle.

6.2 The Electronic Throttle Module

The Electronic Throttle Module (ETM) controls the throttle, which controls the air flow into the engine. A major

```
# Define boolean type
enumerationdef Boolean as [ FALSE, TRUE ]
# Insert some fault here:
faultdef some_fault() as
        emfi_ecu_io_b_stuckat(0, 1)

# The 'err' readout is used to log failures.
readoutdef err oftype string as
        userdef ()
        updatemode usercontrolled

# Check requirements:
experimentdef oracle() as
        fault some_fault
        timer now
        variable r1_x oftype time
        variable r1_time oftype time
        variable r1_active oftype Boolean
        variable prev_igsw oftype Boolean

        # Initialization
        when now ≥ 0 s do
                startlogging err
                start now
                r1_active := FALSE
                prev_igsw := FALSE

        # Update prev_igsw
        when prev_igsw ≠ igsw do
                prev_igsw := igsw

        # Check if ccont must be set to false.
        when igsw = TRUE and prev_igsw = FALSE or prev_igsw = TRUE and
                        igsw = FALSE and not r1_active        do
                r1_x := now
                r1_time := now
                r1_active := TRUE

        # Check if req.1 is not ok.
        when r1_active = TRUE and not (now > r1_x + deadline)
                        or (ccont = TRUE) and (now ≥ r1_x + deadline)        do
                update err with ("r1 failed at time " + tostring(r1_time))
                r1_active := FALSE

        # Check if req.1 remains ok.
        when r1_active = TRUE and (now ≤ r1_x + deadline)
                        and ccont = FALSE        do
                r1_active := FALSE

campaign
        experiment oracle()
```

Fig. 2. Fil Program for example Requirement.

function of the ETM is to control the angle of the trottle in response to the position of the accelerator pedal and doing consistency checks with requests from the Engine Control Module (ECM).

The requirements that were formalized and then translated for the ETM were all of a form saying that some signal should be set if some condition on other signals has been true for a specified period of time. Due to the delays between different components in the test setup (see below), we had to be careful with allowing an appropriate delay for signal propagation. The translated requirements were typically on the form

$$\begin{aligned} \mathit{signal} &\implies \Diamond\, \Diamond_d \mathit{condition} \\ \Diamond\, \Diamond_d \mathit{condition} &\implies \Diamond_{\mathit{small\_delay}} \mathit{signal} \end{aligned}$$

expressing that the signal *signal* is raised if the condition *condition* has been true for d time units. We implicitly understand that the $\Box$ operator is applied to all requirements.

6.3 Testing Setup

Due to inavailability of adequate test equipment during the project, we were not able to exercise the translated oracles on a real CCM.

An ETM was available for experiments. In the conducted test experiments, we did not connect the testing equipment directly to the ETM, for practical reasons. Instead, the ETM, the ECM, and the test equipment communicated via a CAN bus. The scheduling on the CAN bus introduces a delay on signalling between the modules, which means that the *small_delay* parameter in the requirements must be adjusted with care.

Since the ETM used for testing was already in production, no violations were reported. When we introduced a too small value for the *small_delay* parameter, violations were reported.

Summarizing, this experiment was too small as a real evaluation of our approach, but it revealed that our translation works as intended. It was not entirely trivial to tune the requirements to the signalling delays in the test setup, so maybe these delays should be discounted automatically, during the translation.

7 Conclusions

In this paper, we have presented a translation which automatically constructs test oracles from specifications in a propositional temporal logic with a quantification construct of limited power. The translation is related to transformations of temporal logic to an executable form. Case studies that were performed show that reasonable sets of safety requirements can be expressed in our restricted subset of temporal logic, and that the translated test programs can execute in actual test settings. We have not performed sufficient experiments to conclude how useful the approach is for detecting errors under realistic conditions.

As future work, the problem of generating suitable inputs to drive automated testing process should be investigated further.

Acknowledgments Martin Leucker, Sven-Olof Nyström, and the anonymous referees provided many helpful comments.

References

1. Y. Abarbanel, I. Beer, L. Gluhovsky, and S. Keidar an Y. Wolfsthal. FoCS: Automatic generation of simulation checkers from formal specifications. In Emerson and Sistla, editors, *Proc. 12^{th} Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 538–542. Springer Verlag, 2000.
2. R. Alur and T. Henzinger. A really temporal logic. In *Proc. 30^{th} Annual Symp. Foundations of Computer Science*, pages 164–169, 1989.
3. M. Daniele, F. Giunchiglia, and M.Y. Vardi. Improved automata generation for linear temporal logic. In *Proc. 11^{th} Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 249–260. Springer Verlag, 1999.
4. D. Drusinsky. The remporal rover and the ATG rover. In K. Havelund, editor, *SPIN Model Checking and Software Verification, Proc. 7th SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330, Stanford, California, 2000. Springer Verlag.
5. M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. 21st Int. Conf. on Software Engineering*, pages 411–420. IEEE Computer Society Press, May 1999.
6. K. Etessami and G. Holzmann. Optimizing Büchi automata. In C. Palamidessi, editor, *Proc. CONCUR 2000, 11^{th} Int. Conf. on Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–167, 2000.
7. M. Felder and A. Morzenti. Validating real-time systems by history-checking trio specifications. *ACM Trans. on Software Engineering and Methodology*, 3(4):308–339, Oct. 1994.
8. M. Fisher. A normal form for temporal logics and its applications in theorem-proving and execution. *Journal of Logic and Computation*, 7(4):429–456, Aug. 1997.
9. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In Berry, Comon, and Finkel, editors, *Proc. 13^{th} Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer Verlag, 2001.
10. J. Håkansson. Automated generation of test scripts from temporal logic specifications. Master's thesis, Uppsala University, 2000.
11. K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In J.-P. Katoen and P. Stevens, editors, *Proc. TACAS '02, 8^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 324–356. Springer Verlag, 2002.

12. G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, SE-23(5):279–295, May 1997.
13. Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In Courcoubetis, editor, *Proc. 5^{th} Int. Conf. on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 97–109, 1993.
14. Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In *Proc. ICALP '98, 25^{th} International Colloquium on Automata, Lnaguages, and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 1–16. Springer Verlag, 1998.
15. D. Mandrioli, S. Morasca, and A. Morzenti. Generating test cases for real-time systems from logic specifications. *ACM Trans. on Computer Systems*, 13(4):365–398, Nov. 1995.
16. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer Verlag, 1992.
17. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer Verlag, 1995.
18. K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
19. J. Nielsen. Real-time specification using the TRIO language. Master's thesis, Royal Institute of Technology, Stockholm, 1998.
20. T.O. O'Malley, D.J. Richardson, and L.K. Dillon. Efficient specification-based test oracles for critical systems. In *Proc. 1996 California Software Symposium*, April 1996.
21. F. Ouabdesselam and I. Parissis. Testing synchronous critical software. In *Proc. 5th Int. Symp. on Software Reliability Engineering*, pages 239–248, Monterey, CA, Nov. 1994.
22. I. Parissis and F. Ouabdesselam. Specification-based testing of synchronous software. In *Proc. 4th ACM SIGSOFT Symp. on Foundations of Software Engineering*, pages 127–134. ACM Press, 1996.
23. D.K. Peters and D.L. Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, March 1998.
24. P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber. Automatic testing of reactive systems. In *Proc. 19^{th} IEEE Real-Time Systems Symposium*, pages 200–209. IEEE Computer Society Press, 1998.
25. F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Proc. 12^{th} Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 247–263. Springer Verlag, 2000.
26. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. LICS '86, 1^{st} IEEE Int. Symp. on Logic in Computer Science*, pages 332–344, June 1986.