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] Department of Computer Systems, Uppsala University, Sweden. e-mail: `bengt@docs.uu.se`
[3] Volvo Technical Development AB, Gothenburgh, Sweden. e-mail: `ol@vtd.volvo.se`

Abstract. The paper concerns testing as a technique to check that a software component satisfies a specification of safety properties. In testing there must be some procedure, often called a *test oracle*, which checks that the actual observed component behavior conforms to the specified safety properties. Often the human tester acts a test oracle, meaning that the testing procedure could become more efficient if a software module would act as test oracle. A module which checks safety properties as the behavior is observed is called an *on-line* test oracle. In this paper, we present a technique to generate automatically on-line test oracles from specifications of safety properties given in a temporal logic. The logic can express quantitative timing properties, and has a quantification construct to express properties of data values. The technique has been implemented and used in case studies at Volvo Technical Development corp., namely 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.

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: Techniques, such as code inspection and formal verification, are also important techniques which require access to source code. When a system is composed of components that are implemented by different subcontractors, source code is often not available. Testing may then 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 pos-

⋆ This work was supported by the NUTEK within the ASTEC competence center and by Volvo Technical Development AB

sible, 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 select certain combinations of input values, and combine them with corresponding correct output values to form *test cases* or *test sequence*. Besides the problem of finding the correct output values, this approach assumes that the system under test is *deterministic*, i.e., that output values are uniquely determined by input values and their occurrence in time, an assumption that is not valid in general for distributed embedded systems.

An alternative approach that we will consider in this paper is to generate a *test oracle* from the safety requirements. 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 which 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 obvious: requirements can be constructed without technical knowledge about the technicalities of testing, and requirements need only be maintained in one form when they are changed. Such a language should be able to express properties that describes 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 logics are widely used for specifying requirements of reactive systems. 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 [7]. Generation of oracles has also been considered for temporal logics with discrete time (* reference: temporal rover *).

In this paper, we will consider the problem of generating test oracles in a temporal logic extended with constructs for expressing quantitative delays and timing properties. Our work differs from earlier work in the following respects.

- The logic contains past operators, metric time, and a quantification construct to express properties of data values.
- We generate on-line test oracles that report violation of a requirement while the behavior is being observed. In contrast, a tester in a model checker has access to a model of the system, which implicitly contains all its past and potential future behavior.
- A tester in a model checker can in general be non-deterministic, whereas a test oracle must be deterministic.

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 specification safety requirements of embedded system components at Volvo. TRIO is a first-order logic with special constructs for handling metric time. It is very

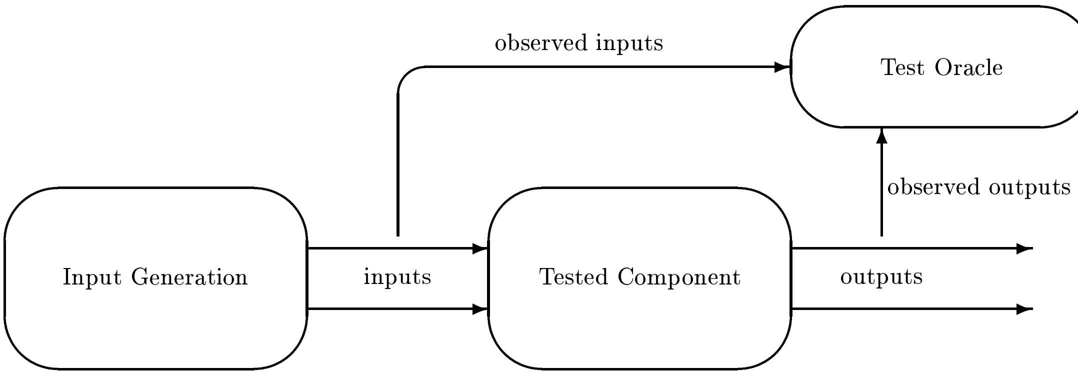

Fig. 1. Role of Test Oracle

expressive, and can easily express undecidable properties. In this presentation, we will instead use a more standard temporal logic (e.g., as in [11]), extended with constructs for metric time and a particular form of quantification, so-called freeze quantification [2], which has a natural operational interpretation as assignment to variables in a program. In the appendices, we will also present the syntax of TRIO formulas which have been used in the case studies.

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. The FIL language has constructs for supplying input values and reading output values of a component, A FIL program executes cyclically with a fixed period. 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. The details and particularities of FIL are not considered in the main paper, but are deferred to an appendix.

We have implemented the translation to FIL programs from safety requirements, and used it on safety requirements for two softare modules: a Cruise controller and a torque controller, showing that the translation can handle specifications of practically occuring software components.

Related Work. Other work, which addresses automated testing and test sequence generation for embedded systems, has been performed in the framework of the language LUSTRE. Ouabdesselam et al [14,15] and Raymond et al [17] 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 [17] also present methods for generating relevant inputs in test sequences. In comparison with this work on LUSTRE, our work uses a more general, declarative temporal logic, which we transform first into an executable form, and then realize in a concrete test execution language.

Kesten et al [8,9] 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 non-deterministic, making it non-suitable for test oracles. Fisher [5] 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. The propositional fragment of our translation is similar to the translation by Fisher. Fisher handles arbitrary propositional formulas with past and future operators. The resulting programs exhibit nondeterminism, which is handled by backtracking. An effort with a similar goal to ours is reported by O'Malley et al. [13], 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 [3] 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. A difference is that the logic considered by the Temporal Rover does not have any quantification construct. 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 [16], 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 [10, Sec. 5], a "history checker" [4] can check whether output conforms to the specification. This history checker is off-line, i.e., it can only check a completed behavior, whereas our test oracles are on-line. A potential problem is that the history must be stored; it may potentially need to record variable values at thousands of time points, since it is not clear which values at which time point will be needed by the history checker.

Our case studies owe much to the specification of a cruise controller in TRIO developed by Nielsen [12]. A full version of the specification case studies presented in this paper appear in the report [6].

Outline. In the next section, we first outline the execution model for components in embedded systems and test oracles. Section 3 briefly present a simple temporal logic with the essential features that we consider. A method for translating a subset of TRIO into test oracles is presented in Section 4. A discussion of how the translation was applied to a case study specification is contained in Section 5. Conclusions and directions for future work are presented in Section 6. Appendix A contains fuller details on the TRIO language. Appendix B contains details about the FIL language. Appendix C presents the requirements that were used in the case study.

2 System Model

We assume that a system, or a component of a system, can be observed through a set of *system variables*. 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). The values of variables change over time. We assume that a test oracle observes the system periodically, at a 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. In what follows, we use *time instant* to mean an instant in time at which the system is observed. We assume that a system is observed over a potentially infinite sequence of time instants.

For a time instant t, let t^+ denote the following time instant, and t^- denote the previous time instant (t^- is not defined 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. More precisely, a temporal formula ϕ expresses a property about time instants in behaviors. Thus, 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.

Appendix A presents the operators of a rich temporal logic, called TRIO. In this section, we consider temporal logics where formulas are built as follows.

Non-Temporal Expressions, i.e., expressions over system variables, such as $v_1 + 4$, $v_1 < v_2$, etc. are temporal logic formulas. We assume given a set of operators for forming expressions, and that (non-temporal) expressions can be translated directly into the target language for test oracles. 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 non-temporal expressions are as follows.

- A *boolean-valued* expression is also called a *state formula*. Examples of state formulas are $v_1 < v_2$, $iseven(v_3)$, etc.
- A *distance term* is an *integer-valued* expression. Distance terms are used in some formulas to denote distance in time. A negative value denotes a distance backwards in time.

Temporal formulas are formed from state formulas and distance terms by applying *temporal operators*. Among the most common operators are the following. We 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 false in the initial time instant.
- $\phi\, \mathcal{U}\, \psi$ states that ψ will hold at some present or future time instant, and that ϕ will hold at all time instants from the current time instant up to the next time instant at which ψ holds.
- $\phi\ \mathcal{W}\ \psi$ is similar to $\phi\ \mathcal{U}\ \psi$, except that it also allows ϕ to hold at all future time instants without ψ ever being true.
- $\phi\ \mathcal{S}\ \psi$ is the past-time analogue of $\phi\, \mathcal{U}\, \psi$, and states 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.
- $\Box\ \phi$ states that ϕ will hold at all present and future time instants.
- $\Diamond\ \phi$ states that ϕ will hold at some present or future time instant.
- $\boxminus\ \phi$ and $\diamondminus\ \phi$ are the past time analogues of $\Box\ \phi$ and $\Diamond\ \phi$.
- $\Box_\tau \phi$ states that ϕ is true from the current time instant up to the next time instant at which at more than τ units of absolute time have elapsed.
- $\boxminus_\tau \phi$ is the past time analogue of $\Box_\tau \phi$.
- $\Diamond_\tau \phi$ states that ϕ will be true at some present or future time instant where at most τ units of absolute time have elapsed.
- $\diamondminus_\tau \phi$ is the past analogue of $\Diamond_\tau \phi$.

Formally, the semantics of these operators can be defined as in Figure 1. Let us introduce some terminology. A formula is called a

- *past formula* if it has no future-time temporal operators,
- *future formula* if it has no past-time temporal operators, and
- *principal formula* if its main connective is a temporal operator.

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 τ is a non-temporal expression and x is a variable, then the formula $x := \tau.\phi(x)$ states that $\phi(x)$ will be true with the current value of τ substituted for x. The semantics for freeze quantification is formally defined as

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

Intuitively, in a formula $x := \tau.\phi$, the current value of τ is "frozen" (recorded) in the variable x before evaluating the formula ϕ. This can be used to express time bounds. For instance $Lasts(\phi, 5)$ can be expressed as

$$x := now\ .\ \phi\ \ \mathcal{U}\ \ (now \geq x + 5)$$

Freeze quantification is intuitively related to assignment. For instance, we could consider to check whether the above formula holds at a given time instant by assigning the current time *now* to a local variable x, and then checking

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

Table 1. Semantics of Temporal Operators and of Quantification

that the formula ϕ holds until $now \geq x + 5$ becomes true.

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

$$
\begin{array}{l}
x := now\ .\ y := v\ . \\
\quad true\ \mathcal{U}\ (now = x + 3 \wedge\ v = y + 5)
\end{array}
$$

states that in 3 time units, the variable v will have increased by 5. We use $x.\phi$ as shorthand for $x := now.\phi$.

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 expressed in a simple guarded-command language. In Appendix B, we will consider how they can be represented in a concrete language used in test equipment. The input to the translation is a temporal logic formula Φ which expresses a requirement on a system or 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 gives a report as soon as 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 local information in local variables. 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. The oracle must not affect the variables of the tested component. If the requirement Φ has been violated by the observed behavior at this time instant, the oracle reports this, e.g., via some additional variable. We assume that the oracle is sufficiently fast to be able to execute its local computations in connection with one time instant before the occurrence of the next time instant.

In this section, we will describe the behavior of an oracle by a set of statements of the form

when *guard* **do** *statements*

where *guard* is a boolean expression over the variables of the tested component and the oracle, and where *statements* is a set of assignments that update the local variables of the oracle. At each observation instant, the oracle evaluates ("in parallell") the *guards* of all **when** clauses, and thereafter executes the *statement* part of those **when** clauses, whose *guard* evaluated to *true*. More precisely, all occurrences of system variables refer to their values at the current time instant, all occurrences of local oracle variables on the left-hand side of assignments refer to their values at the current time instant, whereas all other occurrences of local oracle variables refer to their values at the preceding time instant. It is required that no two assignments in *statements* update the same variable. In Appendix B, we will consider how to transform this representation to a concrete language for test equipment at Volvo.

As a simple example, the requirement

$$\Box \begin{bmatrix} \quad speed < minSpeed \\ \vee\ maxSpeed < speed \end{bmatrix} \implies \neg enabled$$

which constrains the relationship between the system variables *speed* and *enabled* and the constants *minSpeed* and *maxSpeed*, can be translated to the test oracle

$$\begin{array}{ll} \textbf{when} & (\quad speed < minSpeed \\ & \quad \vee\ maxSpeed < speed) \\ & \wedge\ enabled \\ \textbf{do} & violation := true \end{array}$$

The translation is straight-forward for simple invariants like the one above. However, for requirements that constrain the behavior of the system over time in a more complex way, violations can not be reported only on the basis of current values of system variables. Consider for instance the past-time formula

$$(speed < minSpeed)\ \ \mathcal{S}\ \ on$$

expressing that the signal *on* was true at some time in the past, and that $speed < minSpeed$ has been true since then. In order to check whether this formula holds at some time instant, the oracle must use local variables to maintain information about the past behavior of the component under test. Analogously, a future-time formula such as

$$(speed < minSpeed)\ \ \mathcal{U}\ \ off$$

expressing that $speed < minSpeed$ should be true until the signal *off* becomes true, imposes requirements on the continuation of system behavior. The oracle must "remember" how such a future-time requirement, stated at the initial time point, constrains the behavior at later time points, again using its local variables.

The general principles for constructing such test oracles are well-understood, at least for the case of propositional temporal logics (* references *). Here, we will present a syntax-directed approach to the generation of test-oracles for a slightly different logic.

4.1 Principles of the Translation

Assume that the oracle shall determine whether a tested system component satisfies the formula Φ, representing a constraint which is evaluated at the first time instant of the behavior. We assume that in Φ (explicit or implicit) negations occur only in front of system variables.

$$\begin{array}{ll}
\neg\neg\phi & \equiv \phi \\
\neg(\phi \wedge \psi) & \equiv \neg\phi \vee \neg\psi \\
\neg(\phi \vee \psi) & \equiv \neg\phi \wedge \neg\psi \\
\neg x.\phi & \equiv x.\neg\phi \\
\neg x := \tau.\phi & \equiv x := \tau.\neg\phi \\
\neg \circ\ \phi & \equiv\ \ominus\ \neg\phi \\
\neg\ \ominus\ \phi & \equiv\ \ominus\ \neg\phi \\
\neg\phi\ \mathcal{U}\ \psi & \equiv \neg\psi\ \mathcal{W}\ (\neg\phi \wedge \neg\psi) \\
\neg\phi\ \mathcal{W}\ \psi & \equiv \neg\psi\ \mathcal{U}\ (\neg\phi \wedge \neg\psi) \\
\neg\phi\ \mathcal{S}\ \psi & \equiv \neg\psi\ \mathcal{S}\ (\neg\phi \wedge \neg\psi) \\
\neg\phi\ \mathcal{S}\ \psi & \equiv \neg\psi\ \mathcal{S}\ (\neg\phi \wedge \neg\psi) \\
\neg\Box\ \phi & \equiv \Diamond\ \neg\phi \\
\neg\Diamond\ \phi & \equiv \Box\ \neg\phi \\
\neg\ \boxminus\ \phi & \equiv\ \Diamond\!\!\!\!-\ \neg\phi \\
\neg\ \Diamond\!\!\!\!-\ \phi & \equiv\ \boxminus\ \neg\phi \\
\neg\Box_\tau\phi & \equiv \Diamond_\tau\neg\phi \\
\neg\Diamond_\tau\phi & \equiv \Box_\tau\neg\phi \\
\neg\ \boxminus_\tau\phi & \equiv\ \Diamond\!\!\!\!-_\tau\neg\phi \\
\neg\ \Diamond\!\!\!\!-_\tau\phi & \equiv\ \boxminus_\tau\neg\phi
\end{array}$$

Table 2. Rules for pushing negations inwards.

This assumption can be satisfied by pushing negations inwards, using the equivalences in Table 2. During test execution, the oracle observes a sequence $\sigma(0)\ \sigma(1)\ \sigma(2) \ldots$ of states of the tested component. After having observed, say, two states, $\sigma(0)$ and $\sigma(1)$, the oracle has received some information about the behavior. The oracle should then compute a formula which represents the constraint on the remaining behavior $\sigma(2)\ \sigma(3)\ \sigma(4) \ldots$, given that Φ holds for the entire behavior and that $\sigma(0)$ and $\sigma(1)$ are as observed. For instance, if Φ is the formula

$$\Box\ (a \implies \circ\ b)$$

saying that b must be true whenever a was true in the previous time instant, and if a is false in state $\sigma(0)$ and true in state $\sigma(1)$, then a suitable requirement at time instant 2 would be

$$b\ \wedge\ \Box\ (a \implies \circ\ b)\ \ .$$

The preceding discussion suggests that during test execution, the test oracle should maintain a requirement which states what the behavior must satisfy from the next time instant to be observed and onwards, given that the states at past time instants are as observed. We can think of this as a relativization of Φ with respect to the sequence of observed states.

$\phi \,\mathcal{U}\, \psi$	$\equiv$	$\psi \vee (\phi \wedge \circ (\phi \,\mathcal{U}\, \psi))$
$\phi \,\mathcal{W}\, \psi$	$\equiv$	$\psi \vee (\phi \wedge \circ (\phi \,\mathcal{W}\, \psi))$
$\Box \phi$	$\equiv$	$\phi \wedge \circ \Box \phi$
$\Diamond \phi$	$\equiv$	$\phi \vee \circ \Diamond \phi$

Table 3. Rules for decomposing future operators.

Let $\delta(\Phi\ ,\ \sigma(0)\ \ldots\ \sigma(t-1))$ denote the relativization of Φ with respect to the t first states $\sigma(0)\ \ldots\ \sigma(t-1)$. The relativization will be computed incrementally, i.e., the relativization $\delta(\Phi\ ,\ \sigma(0)\ \ldots\ \sigma(t))$ at time instant $t+1$ is computed from $\delta(\Phi\ ,\ \sigma(0)\ \ldots\ \sigma(t-1))$ and the state $\sigma(t)$. This can be done by decomposing $\delta(\Phi\ ,\ \sigma(0)\ \ldots\ \sigma(t-1))$ into a boolean combination of two types of formulas:

- formulas about the past and present behavior at time instant t, and
- formulas about the behavior at time instant $t+1$ and onwards, which occur inside a $\circ$ operator.

Thereafter $\delta(\Phi\ ,\ \sigma(0)\ \ldots\ \sigma(t))$ can be computed by instantiating formulas in the first class by actually observed values of system variables, followed by simplification.

We decompose $\delta(\Phi\ ,\ \sigma(0)\ \ldots\ \sigma(t-1))$ by a sequence of transformation steps. First, all subformulas with a future temporal operator different from $\circ$ as the outermost operator, which do not occur within the scope of another temporal operator, are transformed using the rules in Table 3. Let us, as an example, motivate the rule for $\mathcal{U}$. A formula of form $\phi \,\mathcal{U}\, \psi$ is equivalent to saying that either ψ is true now, or that ϕ is true now and $\phi \,\mathcal{U}\, \psi$ is true in the next time instant.

A subformula of form $x := \tau.\phi$ is transformed to the equivalent formula $\phi[\sigma(t-1)[\tau]/x]$ in which the value of τ at time instant $t-1$ has been assigned to x.

As stated, formulas which do not have a future operator as the outermost temporal operator will be evaluated in the current state to eithr *true* or *false*. To do this, we prescribe the following.

- Future operators must not occur inside past operators.
- For each subformula ϕ with a past operator as the outermost operator, the oracle has a local *log variable*, denoted $log(\phi)$, which is continuously updated so that the value of $log(\phi)$ at time instant t is the value $(\sigma, t) \models \phi$ of ϕ at time instant t.

For past formulas of form $\boxminus_\tau \phi$ or $\diamondminus_\tau \phi$, we employ timers in a way described below. It is possible that some log variables or timers will never be needed (this could be revealed by analyzing the structure of Φ): then there is no need to maintain the variable $log(\phi)$.

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 is the value of formula or variable ϕ at time instant t of the behavior. This can be done by assigning $log(\phi)$ the value of the expression $\chi(\phi)$, which is defined recursively below. The expression $\chi(\phi)$ intuitively represents the value of ϕ in the current time instant; its definition uses the auxiliary expression $\chi^-(\phi)$ which intuitively represents the value of ϕ in the previous time instant.

- $\chi(\phi) = \phi$ if ϕ has no temporal operators,
- $\chi(\phi_1 \wedge \phi_2) = \chi(\phi_1) \wedge \chi(\phi_2)$,
- $\chi(\phi_1 \vee \phi_2) = \chi(\phi_1) \vee \chi(\phi_2)$.
- $\chi(\ominus\, \phi) = \chi^-(\phi)$
- $\chi(\phi \,\mathcal{S}\, \psi) = \chi(\psi) \vee \big(\chi(\phi) \wedge log(\phi \,\mathcal{S}\, \psi)\big)$
- $\chi(\boxminus\, \phi) = \chi(\phi) \wedge log(\boxminus\, \phi)$
- $\chi(\diamondminus\, \phi) = \chi(\phi) \vee log(\boxminus\, \phi)$
- $\chi^-(\phi) = log(\phi)$ if ϕ has no temporal operators, or has a past operator as outermost operator.
- $\chi^-(\phi_1 \wedge \phi_2) = \chi^-(\phi_1) \wedge \chi^-(\phi_2)$,
- $\chi^-(\phi_1 \vee \phi_2) = \chi^-(\phi_1) \vee \chi^-(\phi_2)$.

The assignment can be guarded by some boolean expression which is false in some situations where $log(\phi)$ need not be changed.

To be able to evaluate past formulas of form $\boxminus_\tau \phi$ or $\diamondminus_\tau \phi$, we do as follows.

- For each subformula of form $\boxminus_\tau \phi$ the oracle has a local timer variable, denoted $timer(\phi)$, which measures for how long ϕ has been continuously true.
- For each subformula of form $\diamondminus_\tau \phi$, the oracle has a local timer variable $timer(\neg\phi)$.
- Timers are started and stopped by the statements

  when $\chi(\phi) \wedge \neg\chi^-(\phi)$ **do start** $timer(\phi)$
  when $\neg\chi(\phi) \wedge \chi^-(\phi)$ **do stop** $timer(\phi)$

- The definition of $\chi(\cdot)$ is exended as follows.

  $$\chi(\boxminus_\tau \phi) \equiv \chi(\phi) \wedge timer(\phi) > \tau - \varepsilon$$
  $$\chi(\diamondminus_\tau \phi) \equiv \chi(\phi) \vee timer(\neg\phi) \leq \tau - \varepsilon$$

  where ε is the time between two successive time instants. The first expression states that $\boxminus_\tau \phi$ is true currently and at all time points at most τ time units back. By the execution model, this holds if the timer was started at a time point more than $\tau - \varepsilon$ time units back. The second expression reflects the duality $\diamondminus_\tau \phi \equiv \neg \boxminus_\tau \neg \phi$.

– The future-time formulas $\Box_\tau \phi$ and $\Diamond_\tau \phi$ can be handled by defining them in terms of other operators, as follows.

$$\Box_\tau \phi \equiv x \ . \ \phi \ \ \mathcal{U} \ \ \mathit{now} > x + \tau$$
$$\Diamond_\tau \phi \equiv x \ . \ \mathit{now} < x + \tau \ \ \mathcal{U} \ \ \phi \wedge \mathit{now} \leq x + \tau$$

The first equivalence says that ϕ must remain true until $\mathit{now} > x + \tau$, i.e., until τ time units have passed. The second equivalence says that $\mathit{now} < x + \tau$ must remain true until $\phi \wedge \mathit{now} \leq x + \tau$ becomes true, i.e., that now must not be increased by more then τ before ϕ becomes true.

An alternative to the above treatment of the operators $\Box$ and $\Diamond$ can be given if it is possible to create and start timers. Assume that there is a construct for creating and starting a timer, which returns the name of that timer. Then we can alternatively define

$$\Box_\tau \phi \equiv x := \textbf{create-timer}() \ . \ \phi \ \ \mathcal{U} \ x > \tau$$
$$\Diamond_\tau \phi \equiv x := \textbf{create-timer}() \ . \ x < \tau \ \ \mathcal{U} \ \ \phi$$

4.2 Simplifications

A potential problem with maintaining the formula $\delta(\Phi \ , \ \sigma(0) \ \ldots \ \sigma(t-1))$ in the oracle is that the size of this formula may grow as more and more states of the behavior are observed. However, if we study the rules for decomposing subformulas with temporal operators, as given in Table 3, we see that they do not generate subformulas of new forms. Thus, in a formula without quantification, only a bounded set of temporal subformulas may become part of $\delta(\Phi \ , \ \sigma(0) \ \ldots \ \sigma(t-1))$. Any nontemporal subformula is evaluated in the current state, and thus reduces to *true* or *false*.

The conclusion of the previous paragraph is that if Φ does not contain any quantification, and if the oracle is able to simplify formulas in a reasonable way, then only a bounded set of formulas can occur in $\delta(\Phi \ , \ \sigma(0) \ \ldots \ \sigma(t-1))$ for any t.

Let us next consider the problem of quantified formulas. The problem when relativizing quantified formulas appears in the rule for quantification, which states that $x := \tau . \phi$ is transformed to the equivalent formula $\phi[(\sigma, t) \models \tau / x]$, thereby generating a new instantiation of a subformula. There is a priori no limit on the number of subformulas that can be generated. Let us illustrate the problem by a simple example. Consider the requirement

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

which states that for each time instant, there will be a future time instant at which the value of u is equal to the current value of v. Let v_i be the value of v at time instant i, and similarly for u. Then the relativized requirement at time instant 1 is equal to Φ if $u_0 = v_0$ and equal to $\Phi \ \wedge \ \Diamond \ u = v_0$ otherwise. Continuing in this way, we see that it is possible that the relativized requirement at time instant t is equal to

$$\Phi \ \wedge \ \begin{bmatrix} \Diamond \ u = v_0 \\ \wedge \cdots \\ \wedge \Diamond \ u = v_{t-1} \end{bmatrix}$$

which potentially grows without bound as t increases.

Let us now consider a slightly different case, in which the relativized constraints need not grow. Let u and v be integer variables in the tested component. Let

$$\Phi \equiv \Box \ x := v \ . \ \Diamond \ u \geq x$$

which states that for each time instant, there will be a future time instant at which the value of u is greater or equal to the current value of v. In a similar manner as before, it is possible that the relativized requirement at time instant t is equal to

$$\Phi \ \wedge \ \begin{bmatrix} \Diamond \ u \geq v_0 \\ \wedge \cdots \\ \wedge \Diamond \ u \geq v_{t-1} \end{bmatrix}$$

However, this formula can be simplified to

$$\Phi \ \wedge \ \Diamond \ u \geq v_{max}$$

where v_{max} is the maximum amont $v_0, \ldots, v_{t-1}$.

Let us try to generalize the preceding observation by proposing a simple pragmatic simplification mechanism, based on a notion of *monotonicity*.

Let $\leq$ be the standard ordering on real numbers and integers, and the ordering $\mathit{false} \leq \mathit{true}$ 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 on the fly. If a formula contains two ordered disjuncts, then the smaller one can be discarded. Similarly, if a formula contains two ordered conjuncts, then the larger one can be discarded. Let us consider some examples.

- The formula $\Box_d \phi$ is defined as

$$x \,.\, \phi\,\mathcal{U}\;\; now \geq x + d \;\;.$$

  The expressing $now \geq x + d$ is decreasing in x. Thus, if a conjunct of form

$$\phi\,\mathcal{U}\;(now \geq v + d)$$

  for some value v of x is generated, then all existing conjuncts of form

$$\phi\,\mathcal{U}\;(now \geq v' + d)$$

  with $v' < v$ can be discarded. Intuitively, this corresponds to the intuition that if we are requiring that ϕ will hold for d time units on from some time point v, then we are implicitly requiring that ϕ will hold for d time units on from any earlier time point v' (assuming that this requirement has not been violated).
- The more complicated formula

$$\Box \left[\begin{array}{l} \left[\begin{array}{l} \quad \ominus\; \neg resume \\ \wedge\; resume \\ \wedge\; speed < ref \end{array} \right] \\ \Longrightarrow \\ v := speed.\; x. \\ \qquad \Diamond \left[\begin{array}{l} speed = ref \\ \wedge \\ \dfrac{speed - v}{now - x} = 0.5km/h/s \end{array} \right] \end{array} \right]$$

  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*. It is not trivial to simplify a formula which contains many instantiations of the quantified subformulas. In a practical situation, it seems plausible that the antecedent of the implication will be true at a limited set of time instants, implying that the set of instantiated formulas will be bounded.

4.3 Correctness

The preceding subsections have presented a set of techniques for generating an abstract test oracle for checking violations of requirements. In this section, we will consider these generation rules to see in what respect the oracle actually reports violations as soon as they appear. We first state a theorem, which can be thought of as guaranteeing soundness of the generation, i.e., that an oracle gives no false violation reports.

Theorem 1 (Soundness). *Whenever the oracle generated to check a formula Φ simplifies $\delta(\Phi\;,\;\sigma(0)\;\ldots\;\sigma(t-1))$ to false, then there is no behavior σ' such that $\sigma'(i) = \sigma(i)$ for $i = 0, \ldots, t-1$ with $\sigma'(0)[\Phi]$ is true.*

Proof. (Sketch) The theorem relies on the facts that

- log variables are correctly maintained, i.e., that
$$\sigma(t)[log(\phi)] \equiv \sigma(t)[\phi]$$
  for any $t \geq 0$, and that
- the requirement Φ is correctly relativized, i.e., that
$$\sigma(t)[\delta(\Phi\;,\;\sigma(0)\;\ldots\;\sigma(t-1))] \equiv \sigma(0)[\Phi]$$
  for any $t \geq 0$.

Both these facts can be proven by induction on t by checking that the given transformation rules are indeed correct.

It is more complicated to give a theorem about completeness of the oracle, saying roughly that the oracle will always report violations as soon as they occur. We would like the oracle to report a violation whenever the currently observed sequence of states cannot be extended to an infinite behavior which satisfies the original requirement. This problem is essentially the problem of checking satisfiability, i.e., checking whether there is a behavior which satisfies the current relativization of the original requirement, which in general is a hard problem: even for propositional temporal logic with no past operators, checking for satisfiability is a PSPACE-hard problem (* reference *). In fact, there are unsatisfiable formulas, for which an oracle generated as in this paper will never report a violation. A simple example is the formula

$$\Box\, p \;\wedge\; \Diamond\, \neg p \;\;,$$

where p is any state formula. This formula is unsatisfiable, but when observing a behavior in which all states satisfy p, the oracle will compute relativizations which are all equal to the above formula. The problem is that there is no mechanism to detect that the the conjunct $\Diamond \neg p$, expressing that p becomes false at some point in time, is ever fulfilled. A more complete check for satisfiability can be done by a tableau procedure for linear-time temporal logic (* reference *), which however has exponential worst-case time complexity, and it does not appear reasonable to use it in an on-line oracle.

In view of the above discussion, we will here state a completeness theorem only for a limited subset of the logic considered in the paper. Define *Simple Safety Logic* as the subset of the temporal logic considered here, where formulas satisfy the following constraints:

- explicit or implicit negations are not allowed outside temporal operators,
- the only allowed temporal operators are $\circ$, $\ominus$, $\mathcal{W}$, $\mathcal{S}$, $\Box$, $\Diamond$, $\boxminus$, $\diamondminus$, and the metric variants of $\boxminus$ and $\diamondminus$.
- Only positive integers are allowed as distance terms τ in formulas of form $\boxminus_\tau \phi$ and $\diamondminus_\tau \phi$.
- No quantification is allowed.

For simple safety logic, we can state a completeness theorem.

Theorem 2 (Completeness). *Let Φ be a requirement in simple safety logic. If an observed behavior σ violates the requirement Φ at time instant $t-1$ (i.e., there is no behavior σ' with $\sigma'(i) = \sigma(i)$ for $i = 0, \ldots, n-1$ which sastifies Φ), then a violation will be reported at some time instant m where $m \leq n + 2^{|\Phi|}$, where $|\Phi|$ is the size of Φ.*

Proof. (Sketch) Consider a situation where an observed behavior σ violates the requirement Φ at time instant $t-1$. What complicates the proof is that it may take many time instants until the current relativization of Φ is simplified to *false*. For instance, a relativization can specify that the parallel synchronous composition of a collection of state machines will not deadlock; if this relativization is false, it may potentially take an exponential number (in the size of the requirement) of time instants to reach a deadlock. The simple argument in the proof is therefore the following: If σ violates the requirement Φ at time instant $t-1$, then the two following things are true.

- The relativization of Φ will become false at some time instant k with $t-1 \leq k$.
- There are no two time instants k and l with $t \leq k < l$ at which the relativizations are equivalent, 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 relativizations of Φ. Each relativization of Φ is equivalent to a positive boolean combination of subformulas of Φ, and a combination of truth values of the following expressions:

- log variables $log(\phi)$ for each past subformula ϕ of Φ,
- expressions of form $timer(\phi_1) \leq timer(\phi_2) + K$, where the absolute value of the integer K is not more than the largest constant appearing as a distance term.

We can thus calculate a bound on the number of different "situations" which is doubly exponential in the size of Φ and additionally exponential in the number of past subformulas of Φ.

5 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 [12], and the other was a module for throttle control.

5.1 The Cruise Control Module

A specification in our temporal logic of the Cruise Control Module (CCM) appears in Appendix C. 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. Among the requirements, which were originally stated in a rather precise natural language, there was only one aspect that could not be handled well. In a simplified form, this aspect appears in the formula

> 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.

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

5.2 The Electronic Throttle Module

The Electronic Throttle Module (ETM) controls the throttle, which controls the air flow into the engine. A major function of the ETM module is to control the angle of the trottle position, in response to requests by the Engine Control Module (ECM).

5.2.1 Requirements

The requirement formulas that were 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

$$\begin{array}{lcl} signal & \Longrightarrow & \Diamond\!\!\!\!- \;\; \boxminus_d condition \\ \Diamond\!\!\!\!- \;\; \boxminus_d condition & \Longrightarrow & \Diamond_{small\_delay} \Box \; signal \end{array}$$

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.

5.2.2 Testing Setup

In the conducted test experiments, we did not connect the testing equipment directly to the ETM module, 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.

5.2.3 Results

Since the ETM module 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.

6 Conclusions and Directions for Future Work

In this paper, we have presented a translation which automatically constructs test oracles from specifications in a restricted first-order temporal logic. The translation is related to transformations of temporal logic to an executable form [5], but we restrict the language to avoid non-determinism. The freeze-quantification construct allowed us to introduce a restricted form of first-order temporal logic without causing excessive nonterminism in many practical cases.

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 anything about how useful the approach is for detecting errors under realistic conditions.

As future work, we plan to address the problem of generating suitable inputs to drive automated testing process.

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, 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. 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.
4. M. Felder and A. Morzenti. Validating real-time systems by history-checking trio specifications. *ACM Trans. on Software Engineering and Methodology*, 3(4), Oct. 1994.
5. 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.
6. J. Håkansson. Automated generation of test scripts from temporal logic specifications. Master's thesis, Uppsala University, 2000.
7. G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, SE-23(5):279–295, May 1997.
8. 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*, 1993.
9. Y. Kesten, A. Pnueli, and L. o. Raviv. Algorithmic verification of linear temporal logic specifications. In *Proc. ICALP '98, 25^{th} International Colloquium on Automata, Lnaguages, and Programming*, Lecture Notes in Computer Science. Springer Verlag, 1998.
10. 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.
11. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer Verlag, 1992.
12. J. Nielsen. Real-time specification using the TRIO language. Master's thesis, Royal Institute of Technology, 1998.
13. 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.
14. F. Ouabdesselam and I. Parissis. Testing synchronous critical software. In *Proc. 5th Int. Symp. on Software Reliability Engineering*, pages 239–248, 1994.
15. 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, 1996.
16. 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.
17. P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber. Automatic testing of reactive systems. In *RTSS98*, 1998.

A Overview of TRIO

In this appendix, we give a more complete overview of the language TRIO, for which the translation has been developed in the actual case study. In TRIO, the basic temporal operator is the binary operator *Dist*, which is used to define a set of useful derived temporal operators. Usually, the *Dist* operator is not used explicitly. More common is to use operators that can be derived from the *Dist* operator. In this work, we will work directly on the temporal operators that are defined in Table A together with a definition in the temporal logic of Section 3, and an informal explanation. Observe that TRIO excludes current state $\sigma(t)$ in the definitions of *Until* and *Since*, and also in *Lasted* and *Lasts*. There are alternative versions of these operators, which are distinguished by two additional letters, one for the start and one for the end of the interval. The letter 'e' denotes that the endpoint is excluded and the letter 'i' denotes that the endpoint is included in the interval. As an example, $Lasted(\phi, \tau)$ is associated with open intervals but $Lasted_{ie}(\phi, \tau)$ is associated with intervals closed at the left end point.

B Generation of Test Scripts in FIL

In Section 4, we presented a translation from temporal logic to test oracles represented in an abstract guarded-command language. In this section, we consider how to realize test oracles in a specific target language, more precisely the language FIL, which has been used as the target for our implemented translation. In the next subsection, we give a general overview of the characteristics of the FIL language. The subsection which follows thereafter present some design issues in the implemented translation for the FIL language.

B.1 Overview of the FIL Language

The FIL language is designed to be used 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,

Operators	Definition	Intuitive definition
$\mathit{Next}(\phi)$	$\circ\ \phi$	ϕ becomes true at the next time instant.
$\mathit{Previous}(\phi)$	$\ominus\ \phi$	ϕ was true at the previous time instant.
$\mathit{Becomes}(\phi)$	$\ominus\ \neg\phi \wedge \phi$	ϕ becomes true at this time instant, having been false in previous time instant.
$\mathit{PosEdge}(\phi)$	$\ominus\ \neg\phi \wedge \phi$	ϕ becomes true at this time instant, having been false in previous time instant.
$\mathit{NegEdge}(\phi)$	$\ominus\ \phi \wedge \neg\phi$	ϕ becomes false at this time instant, having been true in previous time instant.
$\mathit{Edge}(\phi)$	$\ominus\ \phi \neq \phi$	ϕ becomes false or true at this time instant.
$\mathit{SetTrue}(\phi)$	$\circ\ \phi$	ϕ becomes true at the next time instant.
$\mathit{SetFalse}(\phi)$	$\circ\ \neg\phi$	ϕ becomes false at the next time instant.
$\mathit{AlwF}(\phi)$	$\circ\ \Box\ \phi$	ϕ is true at all future time instants.
$\mathit{AlwF}(\phi)$	$\Box\ \phi$	ϕ is true at all present and future time instants.
$\mathit{SomF}(\phi)$	$\circ\ \Diamond\ \phi$	ϕ is true at some future time instant.
$\mathit{SomF}(\phi)$	$\Diamond\ \phi$	ϕ is true at some present or future time instant.
$\mathit{Alw}(\phi)$	$\Box\ \phi \wedge \boxminus\ \phi$	ϕ is true at all times.
$\mathit{Som}(\phi)$	$\Diamond\ \phi \vee$ ⟠ ϕ	There is a time (future, past or present) when ϕ holds.
$\mathit{Until}(\phi, \psi)$	$\circ\ \phi\ \mathcal{U}\ \psi$	ψ will hold at some future time instant, andt ϕ will hold at all time instants between the current time instant and the next time instant at which ψ holds.
$\mathit{Until\_i}(\phi, \psi)$	$\phi\ \mathcal{U}\ \psi$	"inclusive" version of $\mathit{Until}(\phi, \psi)$.
$\mathit{UntilW}(\phi, \psi)$	$\circ\ (\phi\ \mathcal{W}\ \psi)$	Weak version of $\mathit{Until}(\phi, \psi)$, which allows ϕ to hold indefinitely.
$\mathit{Since}(\phi, \psi)$	$\ominus\ (\phi\ \mathcal{S}\ \psi)$	past-time analogue of $\mathit{Until}(\phi, \psi)$.
$\mathit{Since\_i}(\phi, \psi)$	$\phi\ \mathcal{S}\ \psi$	past-time analogue of $\mathit{Until\_i}(\phi, \psi)$.
$\mathit{SinceW}(\phi, \psi)$	$\ominus\ (\phi\ \mathcal{S}\ \psi \vee \boxminus\ \phi)$	Weak version of $\mathit{Since}(\phi, \psi)$.
$\mathit{AtLeastOnceSince}(\phi, \psi)$	$\neg\ \ominus\ (\mathit{stttSince}\neg\phi\psi)$	ϕ was true at least once since the last time that ψ was true.
$\mathit{Lasted}(\phi, \tau)$	$\ominus\ \boxminus_{\tau-\varepsilon}\phi$	ϕ was true at all time instants between the last time instant which is at least τ units of absolute time back, and the current time instant.
$\mathit{Lasted}_{ii}(\phi, \tau)$	$\boxminus_{\tau}\phi$	ϕ was true at all time instants between the last time instant which is at least τ units of absolute time back, and the current time instant, including the end points of this interval.
$\mathit{WithinP}(\phi, \tau)$	$\ominus$ ⟠$_{\tau-\varepsilon}\phi$	ϕ held sometime within τ time units in the past.
$\mathit{WithinF}(\phi, \tau)$	$\circ\ \Diamond_{\tau-\varepsilon}\phi$	ϕ will hold sometime within τ time units in the future.

Table 4. defined temporal operators in TRIO.

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.

As described in Section 2, each time instant in the model underlying TRIO formulas correspond to one control loop. The duration of a basic control loop in FIL can be adjusted (above a minimum) to the system under test by adding a suitable delay at the end of each control loop.

Unstructured *variables* in FIL can be of type reals, integers, and of enumerated types. The input and output of the component under test are linked to variables of the FIL program in an *interface definition* (called *readout definition* in FIL). For instance, the output from a sensor can be linked to a variable in the FIL program.

The sensor can be defined to update the FIL variable in different ways, e.g., at regular intervals, only when it changes, etc. In this way, the inputs and outputs of the component under test can be considered as normal variables from the point of view of the FIL program.

An important part of the FIL program are *fault definitions*, which define ways to change the input of the component under test in order to simulate faults. This part of a FIL program is important for generating relevant inputs, but in this work we do not consider how to define suitable faults.

A typical FIL program defines a set of "experiments". Each experiment consists of injecting one or several faults, as 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. The purpose of our generated test oracle is to automate checking that the output conforms to stated safety requirements.

The computation in each control loop is described in a set of *event statements*. Each event statement is triggered by a boolean condition over input and local variables, and may cause update of output or local variables. In one control cycle, all event statements are considered to be executed simultaneously, as an atomic statement. If two event statements perform conflicting updates to some variable, a run-time error is generated.

In the initial execution cycle, all event statements with true conditions are executed. For the remaining execution cycles, each event statement is triggered only at time instants when its condition has changed from *false* to *true*. This means, e.g., that the update to log variables described in Section 4 can not be programmed as

$$log(\phi) := \chi(\phi)$$

but rather as the two statements

$$\begin{array}{l} \textbf{when } \chi(\phi) \textbf{ do } log(\phi) := true \\ \textbf{when } \chi(\neg\phi) \textbf{ do } log(\phi) := false \end{array}$$

Often, the condition can be simplified further, by restricting to the case where the log variable needs to be changed. nnnnn As an example, consider the log variable $log(\phi\ \mathcal{S}\ \psi)$, where ϕ and ψ are formulas which can be straightforwardly expressed in FIL. As described above, we update it by the statements.

$$\begin{array}{ll} \textbf{when } \chi(\psi) & \textbf{do } log(\phi\ \mathcal{S}\ \psi) := true \\ \textbf{when } \neg\chi(\phi) \wedge \neg\chi(\psi) & \textbf{do } log(\phi\ \mathcal{S}\ \psi) := false \end{array}$$

The variable $log(\phi\ \mathcal{S}\ \psi)$ should be initialized to *true*.

Representation of formulas. In Section 4 it was suggested to introduce log variables to represent the values of certain past subformulas of the original formula. In a similar way, we introduce variables to hold the values of subformulas of the relativizations of Φ. The set of subformulas that need to be represented in this way can be determined by a static analysis of Φ.

Subformulas that can be instantiated online by quantification can be represented by a structure which contains values of those parameters for which the corresponding formulas should hold.

B.2 Example

Additional details of generated FIL programs will be illustrated by an example of translation. We consider the first requirement of the cruise control module, which states that at an edge of the signal $igsw$, the signal $ccont$ must be set to false within `deadline` time units. This requirement can be expressed as the formula:

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

Following the translation in Section 4, the resulting oracle should have a variable $log(igsw)$ to record values of $igsw$. The consequent $\Diamond_{\texttt{deadline}} \neg ccont$ is rewritten as

$$x\ .\ (now < x + \texttt{deadline})\ \mathcal{U}\ \neg ccont \quad .$$

. Derivatives of Φ are of one of the forms

$$\begin{array}{l} \Phi \\ \Phi \wedge (now < d + \texttt{deadline})\ \mathcal{U}\ \neg ccont \end{array}$$

for some value d, which has been instantiated for x. Note that repeated instantiations of the quantified variable x will be discarded, since by monotonicity they are redundant. The first form will transform into the second form whenever $log(igsw)$ is unequal to $igsw$. The second form will transform to the first whenever $\neg ccont \wedge now < d + \texttt{deadline}$ is true, or into an error report when $ccont \wedge now < d + \texttt{deadline}$ is true.

The FIL program for this requirement is shown in Figure 2. In this program, the first 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

```
# 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 (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 not (now ≥ r1_x + deadline)
                        and ccont = FALSE          do
                r1_active := FALSE

campaign
        experiment oracle()
```

Fig. 2. Fil Program for example Requirement.

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 work. 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 **experimentdef**. A log variable `prev_igsw` to record values of $igsw$ is declared. A timer `r1_timer` to measure the duration of *ccont* is declared. The relativizations of Φ are of two types: one is simply Φ, and the other also requires *ccont* to become false within some short remaining time interval. The variable `r1_active` is true iff the relativization is of the second type. The log variable `prev_igsw` is updated in a **when-do** clause. The two last **when-do** clauses capture the cases that the deadline is missed, and the case that *ccont* is set to false within `deadline` time units.

C Cruise control case studie

The requirements specified here are derived from requirements specified in [12]. They have been changed somewhat to increase readbility. The requirements 17 and 18 have been numbered 18 and 17, respectively.

C.1 Description

The functionality of cruise control is provided by the Cruise Control Module (CCM), one of several Electronic Control Units (ECUs) in a car. The driver controls the operation of the CCM through five buttons:

- ON/OFF toggle button
- SETPLUS button
- SETMINUS button
- RESUME button
- CANCEL button

The signal *ccont*, set by the CCM, lights an indication light on the dashboard. All other system signals are either input to the CCM from other ECUs or output from the CCM to other ECUs.

C.2 System Signals

As described in [12], the signals $ccws$ and $resuming$ are introduced to support requirements specification, and need not be part of an implementation.

ECU	*Name*
ABS	Anti Blocking System
TCM	Transmission Control Module
CCM	Cruise Control Module
ECM	Engine Control Module
ETM	Electronic Throttle Module
CEM	Central Electric Module

Table 5. ECUs involved in CCM operation

The system signals are described in tables 5 to 8. The domains of these signals are described in section C.3.

C.3 Domains

- Domain of gear lever positions, gear: { Park, Neutral, Reverse, Drive, Drive_L }
- Domain of signal qualities, quality: { Good, Evaluation, Reduced, Poor }
- Domain of speed measures, km/h: 1..270
- Domain of acceleration measures, km/h/s: -0.6 .. 0.6
- Voltage, volts: 0..15
- Percentage, percent: 1..100

C.4 Constants

- δ - Size of timestep in seconds.
- ϵ - Acceptable deviation of acceleration.
- $minVoltage$ - Minimum battery voltage: 9 Volts.
- *sdd* - Speed deviation delay, 1 second.
- *add* - Acceleration denied delay, 2 seconds.
- $minSpeed$ - Minimum cruise control speed, 35 km/h.
- $maxSpeed$ - Maximum cruise control speed, 200 km/h.
- $maxAcc$ - Maximum cruise control acceleration, 3 m/s^2.
- $maxSmoothAcc$ - Limits temporary acceleration for smoothness.
- $tapValue$ - Speed change per button press, 2 km/h.
- $buttonDelay$ - Delay to accelerate or decelerate, 0.5 second.

C.5 Macros

Functions

$$acceleration(v_0, t_0) \equiv time. \left(\frac{ccws - v_0}{(time - t_0) \cdot \delta} \right)$$

Signal	*Description*	*Domain*
$ccseti$	The signal is true when the driver presses the SETPLUS button	boolean
$ccsetd$	The signal is true when the driver presses the SETMINUS button	boolean
$cccanc$	The signal is true when the driver presses the CANCEL button	boolean
$cconoff$	The signal is true when the driver presses the ON/OFF button	boolean
ccr	The signal is true when the driver presses the RESUME button	boolean
$igsw$	The signal indicates that the ignition is turned on	boolean
bpa	The signal is true when the driver presses the brake pedal	boolean
$vbatt$	The signal indicates the battery voltage of the car	volts
$atglp$	The signal indicates the position of the gear lever	gear
$atglpqf$	The signal indicates the quality level of the signal $atglp$, determined by the sender	quality
vs	The signal indicates the vehicle speed, sent from the ABS	km/h
$vsqf$	The signal indicates the quality level of the signal vs, determined by the sender	quality
vsa	The signal indicates the vehicle speed, sent from the TCM	km/h
$vsaqf$	The signal indicates the quality level of the signal vsa, determined by the sender	quality
va	The signal indicates the vehicle acceleration, calculated in CCM based on vs	km/h/s
cco	The signal indicates that the torque request from the driver through the accelerator pedal is higher than the torque request from the cruise control function, sent from ECM torque control function	boolean
ccd	The signal indicates that cruise control is not allowed due to some detected error in the car, sent from the ECM safety function	boolean

Table 6. Input signals to the CCM

Signal	*Description*	*Domain*
cca	The signal indicates for the ECM torque control function that the torque request from the cruise control is allowed to control the engine torque	boolean
$ccont$	The signal indicates for the driver through the dashboard that the cruise control is on, but does not have to be active	boolean
$ccsp$	The signal is used internally in the cruise control function to store the reference speed for the cruise control	km/h
tcc	The signal is the torque request from the cruise control function for the ECM torque control function	percent

Table 7. Output signals from the CCM

Signal	*Description*	*Domain*
$ccws$	The signal represents what the vehicle speed should be under normal conditions when the engine torque is controlled by the cruise control	boolean
$resuming$	The signal is true after the button ccr is pressed or the cruise control has been overridden until the vehicle speed corresponds to the stored set speed ($ccsp$)	boolean

Table 8. Requirement support signals

Predicates

$$AccOk(v_0, t_0, acc) \equiv \\ |acceleration(v_0, t_0) - acc| < \epsilon$$

C.6 CCM System Requirements

The CCM requirement is that the requirements 1 through 14 and 18 through 36 will always be met.

$$sys : AlwP(r1 \wedge \ldots \wedge r14 \wedge r18 \wedge \ldots \wedge r36)$$

C.7 Requirements on ccont

Requirements 1 to 4 concern the *ccont* signal, indicating if the cruise control is on or off.

Requirement 1

The signal *ccont* must be set to false at an edge of *igsw*.

$$r1 : Edge(igsw) \rightarrow SetFalse(ccont)$$

Requirement 2

When *ccd* is true, *ccont* must always be false.

$$r2 : ccd \rightarrow \neg ccont$$

Requirement 3

When *ccont* is false, a positive edge on the signal *cconoff* must set *ccont* to true if not forbidden due to Reqirement 1 or 2.

$$r3 : \begin{bmatrix} \quad \neg ccont \\ \wedge \; PosEdge(cconoff) \\ \wedge \; \neg Edge(igsw) \\ \wedge \; \neg ccd \end{bmatrix} \rightarrow SetTrue(ccont)$$

Requirement 4

When *ccont* is true, a positive edge on the signal *cconoff* must set *ccont* to false.

$$r4 : ccont \wedge PosEdge(cconoff) \rightarrow SetFalse(ccont)$$

C.8 Requirements Disableing cca

Requirements 5 to 14 disables *cca* signal, indicating that the CCM must not control the engine torque. Variables *dcca*5 to *dcca*14 have been added to simplify requirements in section C.9.

Requirement 5

The signal *cca* must be set to false at an edge of *igsw*.

$$r5 : Edge(igsw) \rightarrow SetFalse(cca)$$

$$dcca5 : Edge(igsw)$$

Requirement 6

If *ccont* is false, *cca* must be false.

$$r6 : \neg ccont \rightarrow \neg cca$$

$$dcca6 : \neg ccont$$

Requirement 7

If *bpa* is true, *cca* must be false.

$$r7 : bpa \rightarrow \neg cca$$

$$dcca7 : bpa$$

Requirement 8

If *vbatt* is lower than 9 Volt, *cca* must be false.

$$r8 : vbatt < minVoltage \rightarrow \neg cca$$

$$dcca8 : vbatt < minVoltage$$

Requirement 9

If *atglp* is set to Park, Neutral or Reverse, *cca* must be false.

$$r9 : \begin{pmatrix} \quad atglp = Park \\ \vee \; atglp = Neutral \\ \vee \; atglp = Reverse \end{pmatrix} \rightarrow \neg cca$$

$$dcca9 : \begin{pmatrix} \quad atglp = Park \\ \vee \; atglp = Neutral \\ \vee \; atglp = Reverse \end{pmatrix}$$

Requirement 10

If not all of $atglpqf$, $vsqf$ and $vsaqf$ are set to Good, cca must be false.

$$r10 : \neg \begin{pmatrix} \quad atglpqf = Good \\ \wedge\ vsaqf = Good \\ \wedge\ vsqf = Good \end{pmatrix} \rightarrow \neg cca$$

$$dcca10 : \neg \begin{pmatrix} \quad atglpqf = Good \\ \wedge\ vsaqf = Good \\ \wedge\ vsqf = Good \end{pmatrix}$$

Requirement 11

If vs and vsa for more than one second differs more than 5% with reference to vs, cca must be false.

$$r11 : Lasted\_ii(abs(vs - vsa)/vs > 0.05, sdd) \rightarrow \neg cca$$

$$dcca11 : Lasted\_ii(abs(vs - vsa)/vs > 0.05, sdd)$$

Requirement 12

If vs is lower than 35 km/h or higher than 200 km/h, cca must be false.

$$r12 : (vs < minSpeed \vee maxSpeed < vs) \rightarrow \neg cca$$

$$dcca12 : vs < minSpeed \vee maxSpeed < vs$$

Requirement 13

If va is higher than 3 m/s^2 for more than 2 seconds, cca must be false.

$$r13 : Lasted\_ii(va > maxAcc, add) \rightarrow \neg cca$$

$$dcca13 : Lasted\_ii(va > maxAcc, add)$$

Requirement 14

If $cccanc$ is pressed, cca must be set to false.

$$r14 : cccanc \rightarrow SetFalse(cca)$$

$$dcca14 : cccanc$$

C.9 Requirements on When To Accept Buttons

Reuirements 15 to 17 describe when buttons SETPLUS, SETMINUS and RESUME should be accepted. These requirements are numbered differently ($acc15$ to $acc17$) because they do not specify safety properties.

Before we define these requirements we define when to accept buttons, based on the requirements. These definitions are the ones actually used to determine if a button have been accepted. Requirement 17 only affects the RESUME button.

$$accseti : PosEdge(ccseti) \wedge acc15 \wedge acc16$$

$$accsetd : PosEdge(ccsetd) \wedge acc15 \wedge acc16$$

$$accr : PosEdge(ccr) \wedge acc15 \wedge acc16 \wedge acc17$$

We also define two variables indicating how many buttons are currently pressed.

$$nobutton : \neg ccseti \wedge \neg ccsetd \wedge \neg ccr$$

$$\begin{aligned} onebutton : & (ccseti \wedge \neg ccsetd \wedge \neg ccr) \vee \\ & (\neg ccseti \wedge ccsetd \wedge \neg ccr) \vee \\ & (\neg ccseti \wedge \neg ccsetd \wedge ccr) \end{aligned}$$

Requirement 15

To accept a positive edge from any of the action buttons $ccseti$, $ccsetd$ or ccr, cca must not be forbidden due to any other requirements at the time of the accepted positive edge of the button signal.

$$\begin{aligned} acc15 : & dcca5 \vee dcca6 \vee dcca7 \vee dcca8 \vee \\ & dcca9 \vee dcca10 \vee dcca11 \vee dcca12 \vee \\ & dcca13 \vee dcca14 \end{aligned}$$

Requirement 16

To accept a positive edge from any of the action buttons *ccseti*, *ccsetd* or *ccr*, only one action button is allowed to be pressed. If more than one action button is pressed, all action buttons have to be released before a positive edge is accepted.

$$acc16 : Previous(nobutton) \wedge onebutton$$

Requirement 17

To accept a positive edge from *ccr*, *cca* must have been activated at least once since the last time *ccont* became true.

$$acc17 : AtLeastOnceSince(cca, PosEdge(ccont))$$

Buttons Continously Pressed

These two variables define when SETPLUS or SETMINUS have been pressed continuosly for 0.5 seconds, indicating that the car should accelerate or decelerate, respectively.

$$acc : Since(ccseti, accseti) \wedge Lasted\_ii(ccseti, buttonDelay)$$

$$dec : Since(ccsetd, accsetd) \wedge Lasted\_ii(ccsetd, buttonDelay)$$

C.10 Requirements on When cca Should Be Set

Requirements 18 to 20 define when the signal *cca* should be true, indicating that the CCM control the engine torque.

Requirement 18

When a positive edge of *ccseti* or *ccsetd* is received and accepted, *cca* must be set to true.

$$r18 : (accseti \vee accsetd) \rightarrow SetTrue(cca)$$

Requirement 19

When a positive edge of *ccr* is received and accepted, *cca* must be set to true.

$$r19 : accr \rightarrow SetTrue(cca)$$

Requirement 20

Only when the requirements 17 and 19 are fulfilled, is *cca* allowed to be set to true.

$$r20 : PosEdge(cca) \rightarrow (accseti \vee accsetd \vee accr)$$

C.11 Requirements on ccsp

Requirements 21 to 26 concern the *ccsp* signal, storing the cruise control reference speed.

Requirement 21

When a positive edge of *ccseti* or *ccsetd* is received and accepted and *cca* by this edge goes from false to true, *ccsp* must be set to *vs*.

By this edge: accset and PosEdge(cca) simulaneously?

$$r21 : (accseti \vee accsetd) \wedge PosEdge(cca) \rightarrow SetTo(ccsp, vs)$$

Requirement 22

When a positive edge of *ccseti* is received and accepted and when *cca* is already true before the edge, *ccsp* must be set to a value 2 km/h higher than *vs*, but maximum 200 km/h.

$$r22 : accseti \wedge Previous(cca) \rightarrow SetTo(ccsp, min(vs + tapValue, maxSpeed))$$

Requirement 23

When a positive edge of *ccsetd* is received and accepted and when *cca* is already true before the edge, *ccsp* must be set to a value 2 km/h lower than *vs*, but minimum 35 km/h.

$$r23 : accsetd \wedge Previous(cca) \rightarrow SetTo(ccsp, max(vs - tapValue, minSpeed))$$

Requirement 24

When a positive edge of *ccseti* is received and accepted, and *ccseti* continues to be true continuously for more than 0.5 seconds, *ccsp* must be set to the value of vs as long as the button is pressed.

$$r24 : acc \rightarrow SetTo(ccsp, vs)$$

Requirement 25

When a positive edge of *ccsetd* is received and accepted, and *ccsetd* continues to be true continuously for more than 0.5 seconds, *ccsp* must be set to the value of vs as long as the button is pressed.

$$r25 : dec \rightarrow SetTo(ccsp, vs)$$

Requirement 26

The signal *ccsp* must be set to 0 when *ccont* is false.

$$r26 : \neg ccont \rightarrow SetTo(ccsp, 0)$$

C.12 Requirements on ccws

Requirements 27 to 36 concern the signals $ccws$ and $resuming$. We define the variable $smooth$ to verify that acceleration will be smooth.

$$\begin{aligned} smooth : v := ccws\ .\\ (t.Next(abs(acceleration(v, t))\\ < maxSmoothAcc)) \end{aligned}$$

Requirement 27

When a positive edge of *ccseti* or *ccsetd* is received and accepted and this event changes *cca* from false to true, then $ccws$ must be set to vs.

$$\begin{aligned} r27 : (accseti \vee accsetd) \wedge PosEdge(cca)\\ \rightarrow SetTo(ccws, vs) \end{aligned}$$

Requirement 28

When *cco* is true, $ccws$ must be equal to vs.

$$r28 : cco \rightarrow SetTo(ccws, vs)$$

Requirement 29

When a positive edge of *ccr* is received and accepted and vs is lower than *ccsp*, $ccws$ must start to follow a curve towards *ccsp* and $resuming$ must be set to true. Under normal conditions, this curve should increase smoothly from the starting point and also connect smoothly to *ccsp*. Between the start and the end point, the curve should correspond to an acceleration of 0.5 km/h/s.

$$\begin{aligned} r29 : (accr\ \wedge\ vs < ccsp) \rightarrow\\ SetTrue(resuming) \wedge\\ Until(smooth, NegEdge(resuming)) \wedge\\ v := ccws\ .\\ (t.Until(resuming,\\ \neg resuming \wedge AccOk(v, t, maxAcc))) \end{aligned}$$

Requirement 30

When a positive edge of *ccr* is received and accepted and vs is higher than *ccsp*, $ccws$ must start to follow a curve towards *ccsp* and $resuming$ must be set to true. Under normal conditions, this curve should decrease smoothly from the starting point and also connect smoothly to *ccsp*. Between the start and the end point, the curve should correspond to an deceleration of 0.5 km/h/s.

$$\begin{aligned} r30 : (accr\ \wedge\ vs > ccsp) \rightarrow\\ SetTrue(resuming) \wedge\\ Until(smooth, NegEdge(resuming)) \wedge\\ v := ccws\ .\\ (t.Until(resuming,\\ \neg resuming \wedge AccOk(v, t, maxAcc))) \end{aligned}$$

Requirement 31

At a negative edge of *cco* when *cca* is true, $ccws$ must start to follow a curve towards *ccsp* and $resuming$ must be set to true. Under normal conditions, this curve should decrease smoothly from the starting point and also connect smoothly to *ccsp*. Between the start and the end point, the curve should correspond to a deceleration of 0.5 km/h/s.

$$
\begin{aligned}
r31 : &(NegEdge(cco) \wedge cca) \rightarrow \\
&\quad SetTrue(resuming) \wedge \\
&\quad Until(smooth, NegEdge(resuming)) \wedge \\
&\quad v := ccws\ . \\
&\qquad (t.Until(resuming, \\
&\qquad\quad \neg resuming \wedge AccOk(v, t, maxAcc)))
\end{aligned}
$$

Requirement 32

When a positive edge of *ccseti* is received and accepted, *ccws* must start to follow a curve towards *ccsp* and *resuming* must be set to true. Under normal conditions, this curve should increase smoothly from the starting point and also connect smoothly to ccsp. Between the start and the end point, the curve should correspond to an acceleration of 0.5 km/h/s.

$$
\begin{aligned}
r32 : &\ accseti \rightarrow \\
&\quad SetTrue(resuming) \\
&\quad Until(smooth, NegEdge(resuming)) \wedge \\
v := &\ ccws\ . \\
&\qquad (t.Until(resuming, \\
&\qquad\quad \neg resuming \wedge AccOk(v, t, maxAcc)))
\end{aligned}
$$

Requirement 33

When a positive edge of *ccsetd* is received and accepted, *ccws* must start to follow a curve towards *ccsp* and *resuming* must be set to true. Under normal conditions, this curve should decrease smoothly from the starting point and also connect smoothly to ccsp. Between the start and the end point, the curve should correspond to a deceleration of 0.5 km/h/s.

$$
\begin{aligned}
r33 : &\ accsetd \rightarrow \\
&\quad SetTrue(resuming) \\
&\quad Until(smooth, NegEdge(resuming)) \wedge \\
v := &\ ccws\ . \\
&\qquad (t.Until(resuming, \\
&\qquad\quad \neg resuming \wedge AccOk(v, t, maxAcc)))
\end{aligned}
$$

Requirement 34

When a positive edge of *ccseti* is received and accepted, and *ccseti* continues to be true for more than 0.5 seconds, *ccws* must increase corresponding to an acceleration of 0.4 km/h/s.

$$r34 : acc \rightarrow v := ccws.(t.Next(AccOk(v, t, 0.4)))$$

Requirement 35

When a positive edge of *ccsetd* is received and accepted, and *ccsetd* continues to be true for more than 0.5 seconds, *ccws* must increase corresponding to a deceleration of 0.4 km/h/s.

$$r35 : dec \rightarrow v := ccws.(t.Next(AccOk(v, t, -0.4)))$$

Requirement 36

The signal *resuming* is set to false when *vs* is equal to *ccsp* or *cca* is false or *cco* is true or the conditions of requirements 34 and 35 are fulfilled.

$$
r36 : \begin{bmatrix} & vs = ccsp \\ \vee & \neg cca \\ \vee & cco \\ \vee & acc \\ \vee & dec \end{bmatrix} \rightarrow SetFalse(resuming)
$$