Automated Generation of Test Scripts from Temporal Logic Specifications

John Håkansson

Uppsala University 2000

Abstract

This master's thesis describes a project carried out at Volvo Technological Development and Uppsala University, within the ASTEC competence center. The purpose of the project is to reduce time spent on testing, which is dominant in todays development process. The testing process can roughly be divided into generation, execution and evaluation of test cases. In this project we have focused on the automation of test evaluation, mainly to evaluate if the execution of a test case exposes any software defects.

Within the project we have developed a method for generating test oracles to evaluate if a test execution violates requirements. We have implemented this method and tested the implementation using testing equipment at Volvo.

To generate test oracles we use formal requirements in TRIO, and the resulting oracles are generated in FIL, a fault injection language used at Volvo for testing. When testing the implemetation, we were specifically interested in the practical usefulness of the resulting tool.

Contents

1 Introduction **6**
1.1 Project Background 6
1.2 Project Description 6
1.3 Thesis Disposition 7

2 A Simple Example **8**
2.1 A System Description 8
2.2 Formalizing the Requirements 9
2.2.1 The System Requirement: Alw 9
2.2.2 Requirement 1: Lasted 9
2.2.3 Requirement 2: Until 9
2.3 Generating a Test Oracle 10
2.3.1 Using the tool 10
2.3.2 Generation Overview 10
2.3.3 Generating FIL for Requirement 1 11
2.3.4 Generating FIL for Requirement 2 11
2.4 Executing the Test Oracle 11

3 Compiling Temporal Logic to Test Oracles **12**
3.1 Overview 12
3.2 Syntactical Restrictions 13
3.3 Pushing Negations Inwards 14
3.4 Log Variables 14
3.4.1 Recurrence Relations 14
3.4.2 Translation into FIL Statements 15
3.4.3 Time Variables: Evaluating Timed Past Formulas 15
3.5 Future Formulas 16
3.5.1 Recurrence Relations 16
3.5.2 Translation into FIL Statements 17
3.5.3 Nested Future Formulas 17
3.5.4 Timed Future Formulas 18
3.6 Freeze Quantified Formulas 18
3.6.1 Operator Monotonicity and Parameter Logging 18

4 Evaluation of Case Studies **20**
4.1 Introduction 20
4.2 Cruise Control Module 20
4.3 Electronic Throttle Module 20
4.3.1 Description 20
4.3.2 ETM Testing Setup 20
4.3.3 Requirements 21
4.3.4 Test Cases 22

4.3.5 Test Results . . . 22
4.3.6 Conclusions . . . 22

5 Towards Automated Testing **23**
5.1 The Testing Process . . . 23
5.2 Test Generation . . . 23
5.2.1 Action-Driven Testing . . . 24
5.2.2 Data-Driven Testing . . . 24
5.2.3 State-Driven Testing . . . 24
5.2.4 Example Design Rule . . . 24
5.3 Test Execution . . . 24
5.4 Test Evaluation . . . 24
5.5 Conclusions . . . 25

A The TRIO Language Dialect **27**
A.1 Syntax . . . 27
A.1.1 Terms . . . 27
A.1.2 Formulas . . . 27
A.2 Semantics . . . 28
A.2.1 Evaluations . . . 28
A.2.2 Temporal Operators . . . 28
A.2.3 Freeze Quantification . . . 30
A.3 Not really TRIO . . . 31

B The FIL Language **32**
B.1 FIL Program parts . . . 32
B.1.1 Type Definitions . . . 32
B.1.2 Constant Definitions . . . 32
B.1.3 Fault Definitions . . . 32
B.1.4 Readout Definitions . . . 33
B.1.5 Control Definitions . . . 33
B.1.6 Experiment Definitions . . . 33
B.1.7 Campaign Statements . . . 33
B.2 Event Statements . . . 33

C Cruise control case studie **35**
C.1 Description . . . 35
C.2 System Signals . . . 35
C.3 Domains . . . 35
C.4 Constants . . . 37
C.5 Macros . . . 37
C.6 CCM System Requirements . . . 38
C.7 Requirements on ccont . . . 38
C.8 Requirements Disableing cca . . . 38
C.9 Requirements on When To Accept Buttons . . . 40
C.10 Requirements on When cca Should Be Set . . . 41
C.11 Requirements on ccsp . . . 41
C.12 Requirements on ccws . . . 42

D Example Files **45**
D.1 TRIO specification . . . 45
D.2 FIL oracle . . . 45

List of Tables

2.1 The Microwave Oven's System Signals 8

C.1 ECUs involved in CCM operation 36
C.2 Input signals to the CCM 36
C.3 Output signals from the CCM 36
C.4 Requirement support signals 37

List of Figures

1.1 System test setup, overview 6
1.2 Oracle generation, overview 7

4.1 Setup for ETM testing 21

Chapter 1

Introduction

1.1 Project Background

The Master's project that is described in this thesis was carried out within ASTEC (Advanced Software TEChnology), a competence center for industrial and university research collaboration within the area of software technology. Guiding, supervision and ideas for this project have been provided by Bengt Jonsson at the University of Uppsala and Ola Lundkvist at Volvo Technological Development.

The purpose of the project is to reduce time spent on testing. To address this purpose we develop a method, and a tool, to aid automated testing through generation of test oracles from formal specifications. A test oracle generated for a specification is intended to observe the specified system during testing, and report any observed violations of that specification. The problem of generating input data to provoke the system (test case generation) is considered a separate problem, which will not be addressed here.

The tool developed uses TRIO (a timed logic language) to describe specifications, and generates code in FIL (Fault Injection Language). FIL is a language developed at Volvo for purposes of fault injection while testing. The generated FIL code is used to observe system behaviour and analyze it for specification violations.

1.2 Project Description

When testing distributed real-time embedded systems, it is important that the equipment used for testing does not affect the internal timing of the system under test. This is why we consider system test setups comparable to that of Figure 1.1.

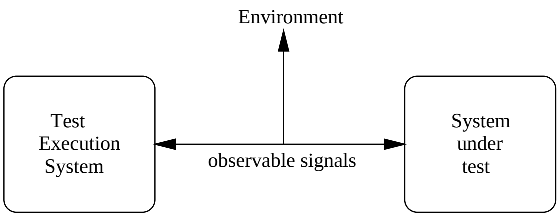

Figure 1.1: System test setup, overview

Using such a setup, the test execution system can observe the environment under roughly the same conditions as the system under test. The test execution system in this setup can be used to execute selected test cases and evaluate the system by observing the system and its environment.

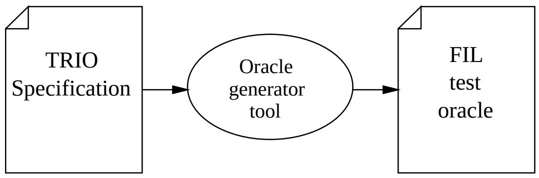

Figure 1.2: Oracle generation, overview

This project focuses on automating evaluation of test executions, by generating test oracles using a tool as in Figure 1.2. A test oracle is something that determines if a system works correctly by observing the system and its environment. It could be anything from an end-user saying 'yes' or 'no' to a generated program, as in this project. The oracles generated by the method described in Chapter 3 observes the environment online, to enable timely evaluation of correctness.

To generate oracles, we need formal requirements. We have chosen to use the TRIO temporal logic language for this since it has been used previously by Volvo for formal requirements specifications (for example in [4]). The TRIO language is described in more detail in Appendix A. The requirements in TRIO are used to generate test oracles in FIL. FIL is a fault injection language used at Volvo in the test execution system, and was initially intended for fault injection. The FIL language is described in Appendix B.

In this project we describe a method to generate test oracles in FIL from TRIO requirements, and evaluate a tool implementing this method. This evaluation was done in a lab at Volvo TU, testing a throttle module using the test execution system to emulate its environment. The purpose of this evaluation was to see if the tool implementing the described method was practically useful.

1.3 Thesis Disposition

Chapter 1 introduces the problems addressed, and describes the project as a whole. In Chapter 2 a simple example is used to describe how test-oracles are generated and how these oracles work.

Chapter 3 describes in more detail the method used to generate FIL code to evaluate TRIO formulas. After this the tool is evaluated in Chapter 4. Further ideas towards the main goal of automated testing are presented in Chapter 5.

In Appendix A the TRIO specification language is described, as is the FIL fault injection language in Appendix B. Appendix C describes the Cruise Control Module (CCM) case study, and finally Appendix D lists the TRIO and FIL files used by the example in Chapter 2.

Chapter 2

A Simple Example

In this chapter we will describe how a test oracle can be generated and used for a simple example system. Initially we describe this system, and a few of its requirements (Section 2.1). We then describe how these requirements can be formalized in TRIO (Section 2.2). After this we describe how an oracle can be generated from TRIO, and how this oracle is executed to evaluate testing.

For reference, the example TRIO requirements and the resulting FIL oracle can be found in Appendix D.

2.1 A System Description

The sample system that we use is a microwave oven. The specification we give here is used to derive a couple of requirements. These requirements are used to describe how we go about generating a test-oracle.

The microwave oven has three states: *off*, *standby* and *on*. It also has a start button and a timer. When the start button is pressed, the microwave oven will enter the *on* state until the timer reaches zero. The microwave oven may not be on for more than an hour at a time. We will also assume that a user can set the timer and that the timer decreases when the oven is on. The oven specification is described by Table 2.1 and the following two requirements.

Requirement 1

The microwave oven may not be in the *on* state for more than an hour.

Requirement 2

When the microwave oven is in the *standby* state and the start button is pressed, it will become *on* until the timer reaches zero. If the timer is zero, the start button has no effect.

Signal Name	*Description*	*Domain*
state	The current state of the oven	$\{off, standby, on\}$
timer	The time that remains of an on-cycle	$0 \ldots 3600s$
start	The start button	*boolean*

Table 2.1: The Microwave Oven's System Signals

2.2 Formalizing the Requirements

To be able to generate test oracles from a specification we need to make the specification unambiguous. This, as well as simplified parsing, is accomplished by using the formal language TRIO. While we describe how the requirements are formalized, we will also introduce the fundamentals of TRIO. Further details on TRIO can be found in Appendix A.

Essentially, a TRIO specification consists of a sequence of requirements. Each requirement is described as a name and a formula, separated by a colon. The TRIO formulas are ordinary predicate-logic formulas, extended by the introduction of *temporal operators*. A temporal operator describes state changes over time, and time in TRIO is modeled as discrete time-steps.

2.2.1 The System Requirement: Alw

The temporal operator Alw is applied to a sub-formula ϕ, resulting in the temporal formula $Alw(\phi)$. The meaning of this formula is that for the formula $Alw(\phi)$ to hold, the sub-formula ϕ must always hold. Note that always means both in the future and the past; there are specific operators $AlwF$ and $AlwP$ dealing with sub-formulas that must hold in the future and in the past, respectively.

We will use the Alw operator to describe the microwave oven system as two requirements, $r1$ and $r2$, that always must hold. In TRIO this is stated as:

$$sys : Alw(r1 \wedge r2)$$

This should be read as: The requirement sys holds only if the requirements $r1$ and $r2$ both hold at all times.

2.2.2 Requirement 1: Lasted

The temporal formula $Lasted(\phi, d)$ means that the formula ϕ must have been true for d consecutive time-steps in the past. We use the $Lasted$ operator to formalize the first requirement:

$$r1 : \neg Lasted(state = on, ONE\_HOUR + 1)$$

This should be read as: The requirement $r1$ holds at a particular time-step only if the microwave oven has *not* been on for ONE_HOUR time-steps. The constant ONE_HOUR represents the number of time-steps in one hour.

The extra time-step is introduced because the temporal domain of TRIO is discrete, and the temporal operators of TRIO does not include the end-points of intervals.

2.2.3 Requirement 2: Until

The temporal formula $Until(\phi, \psi)$ means that in the future the formula ϕ must hold until the formula ψ holds, and that the formula ψ must hold at some point in the future. Using the $Until$ operator we formalize the second requirement:

$$r2 : (timer > 0 \wedge state = standby \wedge start) \rightarrow Until(state = on, timer = 0)$$

This means that if the timer is non-zero when the oven is in *standby* and *start* is pressed, the microwave oven will be *on* until the timer reaches zero. The TRIO requirements sys, $r1$ and $r2$ together with the signal definitions fully describe the informal requirements of Section 2.1. This TRIO specification is unambiguous as well as easily parsed.

2.3 Generating a Test Oracle

2.3.1 Using the tool

To generate an oracle *example.fil* from a TRIO specification *example.trio*, we type the command:

oracle example.trio example.fil sys

This will generate an incomplete FIL oracle from the requirement *sys* of the TRIO specification. The oracle will observe the system under test, and report when the observations contradict the requirements.

To complete the oracle we must insert *FIL readouts* for the system signals. A FIL readout is a definition of how values can be read into the fault injection equipment, for example from an output port of the observed system. These FIL readouts are written manually in a separate file, and then inserted into the FIL file. The format of FIL readouts is described in Appendix B.

2.3.2 Generation Overview

The first step of the generation is to divide the requirement *sys* into what we call *static requirements*. Static requirements are formulas that *always* must hold, in our example this would be the two requirements $r1$ and $r2$.

For each static requirement the oracle generator then creates an *event-statement*. The event-statement generated by a static requirement has the form:

when *event_expression* **and** new_step = TRUE **do**
    update static_req_error **with** Failed

The *event-expression* should become true when the static requirement does not hold. The variable *new_step* is used to implement TRIO time-steps in FIL; it will be set once for every time-step. To generate event-expressions for temporal operators we use global FIL variables.

For past operators we simply use a *log*-variable to remember previous observations. For example, a variable *time_lasted*1 is used to evaluate the past formula $Lasted(p, d)$. The variable is updated by the following FIL event-statement:

when p = FALSE **and** new_step = TRUE **do**
    time_lasted1 := now

The variable *time_lasted*1 remembers the latest time-step when p was false. This is then used to determine for how long p has been true, by substituting ($now \geq time\_lasted1 + d - 1$) for $Lasted(p, d)$.

For future operators we use a *req*-variable to indicate that an operator must hold for the previous time-step. This is what we call a *dynamic requirement*, since it is only a requirement for the time-steps when the req-variable is set. The req-variable for the requirement $p \rightarrow Until(q, r)$ is updated by these event-statements:

when req_until1 = FALSE **and** p = TRUE **and** new_step = TRUE **do**
    req_until1 := TRUE

when req_until1 = TRUE **and** r = TRUE **and** new_step = TRUE **do**
    req_until1 := FALSE

when req_until1 = TRUE **and** q = FALSE **and** r = FALSE
        and new_step = TRUE **do**
    update req_until1_error **with** Failed
    req_until1 := FALSE

The first statement states that when p holds the dynamic requirement should be activated, and therefore *req_until*1 is set. The second deactivates the requirement if it is fulfilled. The third event-statement reports an error and deactivates the requirement if it is violated.

2.3.3 Generating FIL for Requirement 1

To check requirement 1, we introduce a log-variable ($time\_r1Lasted1$). This variable remember for how long the microwave oven has been on, and is used in the FIL event-statement generated for requirement 1:

```
# Check requirement r1
when now ≥ (time_r1Lasted1 + ONE_HOUR - 1) and new_step = TRUE do
      update r1_error with Failed
```

This event-statement ensures that when the oven has been on for an hour, a readout called *r1_error* is updated with the value *Failed*. For this to work, the variable $time\_r1Lasted1$ must be updated:

```
# Store time_r1Lasted1
when state ≠ on and new_step = TRUE do
      time_lasted1 := now
```

2.3.4 Generating FIL for Requirement 2

No FIL event statement for checking requirement 2 directly is generated, since requirement 2 cannot be violated for any time-step at that time-step. The oracle generator will actually generate an event-statement with a FALSE event-expression, which is then discarded.

Instead FIL code is generated to check the dynamic requirement $Until(state = on, timer = 0)$. We begin with an event-statement to activate the dynamic requirement:

```
# Activate req_r2Until1
when req_r2Until1 = FALSE and timer > 0 and state = standby
         and start = TRUE and new_step = TRUE do
      req_r2Until1 := TRUE
```

This simply states that when ($timer > 0 \wedge state = standby \wedge start$) holds the dynamic requirement must also hold, and therefore it is activated. When the dynamic requirement is active we also need to check if it is met, using these FIL event-statements:

```
# Requirement req_r2Until1 is met
when req_r2Until1 = TRUE and timer = 0 and new_step = TRUE do
      req_r2Until1 := FALSE

# Requirement req_r2Until1 is violated
when req_r2Until1 = TRUE and state ≠ on and timer ≠ 0
         and new_step = TRUE do
      update r2_error with Failed
      req_r2Until1 := FALSE
```

2.4 Executing the Test Oracle

When we are ready to start testing, we load the FIL program into the FI-equipment and start executing the FIL oracle. When the oracle is running we execute some test-cases. This can be done either manually or to some extent automatically.

When a requirement is contradicted by observations, this can be recorded by logging or by alerting the user directly. When we use logging, the log will contain time-stamps and corresponding values for each item logged. If the log is empty then all requirements were met for the entire test execution.

Chapter 3

Compiling Temporal Logic to Test Oracles

3.1 Overview

In this section, we present our method for translating TRIO requirements into executable test oracles in the FIL language. The main idea behind the translation is that each TRIO requirement ϕ generates a set of FIL statements that monitor the system variables of the observed system, and reports when the requirement ϕ is violated. As a simple example, the following requirement:

$$Always[(speed < minSpeed \vee maxSpeed < speed) \quad \Longrightarrow \quad \neg enabled]$$

gives rise to the FIL-expression

when (speed < minSpeed **or** maxSpeed < speed) **and** enabled **do**
    update req_error **with** Failed

This scheme works fine for simple invariants like the one above. However, when the requirement contains temporal operators, it is not enough to consider only one time point when determining whether a requirement is satisfied. For instance, to check that a past-time formula of form $Since((speed < minSpeed), on)$ is satisfied, we need to remember certain aspects of the past history of the system. Conversely, a future-time formula such as $Until((speed < minSpeed), off)$ imposes requirements on the continuation of system behavior. We must therefore introduce auxiliary variables which carry information between different time points, e.g., to log the truth of past-time formulas, or to project future-time requirements to later time points. In the remainder of this section, we will work out the concrete details of this idea.

To explain these details we will use a few notations for describing time. As mentioned in Section 2.2, the TRIO time-model is discrete. To describe a point in time three time-steps after a time t, we write $t + 3$. This is of course applicable to all integers, so we can describe time-points before and after t. We will also use t^+ as shorthand for $t + 1$, and t^- as shorthand for $t - 1$.

Following e.g., Kesten et al. [3], we adopt the idea that formulas, whose truth-value may be relevant for the requirement to be checked, are represented by state variables, which record the current value of the sub-formula. The current value of state formulas can be directly inferred from the system variables. The value of a temporal formula may depend on the history or future of a computation. In contrast to the work in [3], where the past and future are essentially treated symmetrically, we must treat past and future time by different methods, since we are generating an on-line oracle for the requirement to be checked.

Let a temporal formula be called a

- *state formula* if it has no temporal operators,
- *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.

Assume that we intend to check whether a computation satisfies a formula ϕ. We introduce the following auxiliary variables:

- For each principal past formula ψ which may be relevant for the satisfaction of ϕ, we introduce a variable $log(\psi)$. The statements in the FIL program will be constructed so that $log(\psi)$ at time t is assigned the value of ψ at time t. This means that $log(\psi)$ can be used in right-hand sides of assignments at time t^+ and represent the value of ψ at time t. The characterization of which variables and formulas are "relevant" can be given exactly, but an approximation is to include all principal past formulas that are sub-formulas of ϕ.
- We introduce a formula, called $\mathcal{G}$, which represents the constraints on the present and future time instants of the computation that are derived from ϕ. The idea is that $\mathcal{G}$ changes along with the computation to reflect what happens. There are Two special cases.
  - If $\mathcal{G}$ becomes *false*, then it becomes impossible to satisfy $\mathcal{G}$, implying that the computation has violated the requirement.
  - If $\mathcal{G}$ assumes the value *true*, then the requirement has been satisfied and no further checking is necessary

  For practical purposes, we will divide $\mathcal{G}$ into two conjuncts: one conjunct which changes over time and which is updated at each instant in time, and one conjunct which is static over time and never updated. Typically, the second conjunct will include invariant requirements of form $AlwF(\phi)$ for some formula ϕ.
- For certain sub-formulas, typically those that occur as arguments ϕ in formula of form $Lasted(\phi, d)$, we log times. The time variable corresponding to formula ϕ is called $time(\phi)$. The intention is that at time point t the variable $time(\phi)$ equals the latest time $t' < t$ such that $\neg\phi$ holds at time t'.

In the following sections, we will treat each of these forms of variables and their associated FIL statements. For each ϕ, we will develop equations, which express how the values of $log(\phi)$ and $\mathcal{G}$ should be updated in basic system steps. These equations will then result in event-statements updating auxiliary variables and reporting requirement violations.

3.2 Syntactical Restrictions

The translation scheme outlined here assumes the following restrictions on the syntax of requirements to make the presentation and translation simpler.

- We assume that the formula is given with negation occuring only within state formulas or in front of principal formulas. Formulas which do not satisfy this requirement can be transformed to do so using the transformation outlined in Section 3.3.
- In a disjunction, at most one disjunct may contain a future operator.
- Freeze quantification, which is introduced in Section 3.6, may be applied only to future formulas.
- Future operators may not occur inside the scope of past operators.
- Future operators may only be nested if the resulting recursive formula (as described in Section 3.5.3) does not contain future operators in more than one disjunct. For example $Until(\phi, \psi)$ must not contain a future operator in ψ, since the recursive definition of $[Until(\phi, \psi)](t)$ is $[\psi](t) \vee ([\phi](t) \wedge [Until(\phi, \psi)](t{+}1))$.

3.3 Pushing Negations Inwards

A prerequisite for our translation scheme is to push negations inwards in a formula until they reach state formulas or principal formulas. The reason for doing this is that it is an extra burden to have the recurrence rules cope with negations.

In any formula, we can push negations inwards using the following transformation rules.

$$\begin{array}{lcl}
\neg\neg\phi & \Longrightarrow & \phi \\
\neg(\phi \wedge \psi) & \Longrightarrow & \neg\phi \vee \neg\psi \\
\neg(\phi \vee \psi) & \Longrightarrow & \neg\phi \wedge \neg\psi \\
\neg x.\phi & \Longrightarrow & x.\neg\phi \\
\neg x := \tau.\phi & \Longrightarrow & x := \tau.\neg\phi
\end{array}$$

Negations of principal future formulas are handled by transforming the negation into an equivalent principal future formula.

3.4 Log Variables

In this section, we will consider how to maintain the value of each variable of form $log(\phi)$, where ϕ is a principal past formula. Recall that the intention is that $log(\phi)$ at time t shall be assigned the value of ϕ at time t.

Let us extend the idea of log variables to arbitrary past formulas as follows. For *any* past-time formula ϕ, define the expression $\chi(\phi)$ as follows.

- $\chi(\phi) = \phi$ if ϕ is a state formula,
- $\chi(\phi) = log(\phi)$ if ϕ is a formula which has a temporal operator as the outermost operator,
- $\chi(\phi_1 \wedge \phi_2) = \chi(\phi_1) \wedge \chi(\phi_2)$,
- $\chi(\phi_1 \vee \phi_2) = \chi(\phi_1) \vee \chi(\phi_2)$.
- $\chi(\neg\phi) = \chi(\phi')$, where ϕ' is the formula which is equivalent to ϕ, obtained by pushing negations inwards.

For state formulas ϕ that contain only system variables, we can use the expression ϕ also in the right-hand side expressions of the recurrences below, since the system variables are continuously updated.

3.4.1 Recurrence Relations

The basis for constructing assignment statements to variables of form $log(\phi)$ are *recurrence relations*, which show how the value of a past formula at time t depends on values of past formulas at time t^-. Such recurrence relations are fundamental to any approach for verifying or testing requirements formulated in temporal logic (e.g., [6, 2]). Past formulas satisfy the following recurrence relations

$$\begin{array}{lcl}
[Previous(\phi)](t) & = & [\phi](t^-) \\
[Since(\phi,\psi)](t) & = & [\psi](t^-) \;\vee\; \big([\phi](t^-) \;\wedge\; [Since(\phi,\psi)](t^-)\big) \\
[Since_i(\phi,\psi)](t) & = & [\psi](t) \;\vee\; \big([\phi](t) \;\wedge\; [Since(\phi,\psi)](t)\big) \\
[AlwP(\phi)](t) & = & [\phi](t^-) \wedge [AlwP(\phi)](t^-) \\
[AlwP_i(\phi)](t) & = & [\phi](t) \wedge [AlwP(\phi)](t) \\
[SomP(\phi)](t) & = & [\phi](t^-) \vee [SomP(\phi)](t^-) \\
[SomP_i(\phi)](t) & = & [\phi](t) \vee [SomP(\phi)](t)
\end{array}$$

¿From these recurrence formulas in the preceding section, we derive the following recurrence relations between variables of form $log(\phi)$ where ϕ is a past time formula.

$$\begin{array}{lcl}
[log(Previous(\phi))](t) & = & [\chi(\phi)](t^-) \\
[log(Since(\phi,\psi))](t) & = & [\chi(\psi)](t^-) \;\vee\; \big([\chi(\phi)](t^-) \;\wedge\; [\chi(Since(\phi,\psi))](t^-)\big) \\
[log(Since_i(\phi,\psi))](t) & = & [\psi](t) \;\vee\; \big([\chi(\phi)](t) \;\wedge\; [\chi(Since(\phi,\psi))](t)\big) \\
[log(AlwP(\phi))](t) & = & [\chi(\phi)](t^-) \wedge [\chi(AlwP(\phi))](t^-) \\
[log(AlwP_i(\phi))](t) & = & [\phi](t) \wedge [\chi(AlwP(\phi))](t) \\
[log(SomP(\phi))](t) & = & [\chi(\phi)](t^-) \vee [\chi(SomP(\phi))](t^-) \\
[log(SomP_i(\phi))](t) & = & [\phi](t) \vee [\chi(SomP(\phi))](t)
\end{array}$$

3.4.2 Translation into FIL Statements

The previous recurrence relation can now serve as a basis for generating FIL statements that maintain proper values of variables of form $log(\phi)$. One must be careful to consider some aspects of the execution mechanism of FIL.

- Event-statements will only be triggered when the value of the condition changes from *false* to *true*.
- The effect of an assignment at time t will not be visible until the time t^+.
- The auxiliary variables are initialized to default values, for boolean variables the default value is *false*.

The statements for principal past formulas $Previous(\phi)$, $Since(\phi,\psi)$ and $SomP(\phi)$ are:

$$\begin{array}{lll}
\textbf{when} & \chi(\phi) & \textbf{do}\;\; log(Previous(\phi)) := true \\
\textbf{when} & \chi(\neg\phi) & \textbf{do}\;\; log(Previous(\phi)) := false \\
\textbf{when} & \chi(\psi) & \textbf{do}\;\; log(Since(\phi,\psi)) := true \\
\textbf{when} & \chi(\neg\phi \wedge \neg\psi) & \textbf{do}\;\; log(Since(\phi,\psi)) := false \\
\textbf{when} & \chi(\phi) & \textbf{do}\;\; log(SomP(\phi)) := true
\end{array}$$

For $AlwP(\phi)$ the following equality is used, since log-variables are initialized as *false*. The other equalities are used due to the assignment semantics of FIL, which make it impossible to read from a log-variable at time t the value that was logged at time t.

$$\begin{array}{l}
AlwP(\phi) \equiv \neg SomP(\neg\phi) \\
Since_i(\phi,\psi) \equiv \psi \vee (\phi \wedge Since(\phi,\psi)) \\
SomP_i(\phi) \equiv \phi \vee SomP(\phi) \\
AlwP_i(\phi) \equiv \phi \wedge \neg SomP(\neg\phi)
\end{array}$$

3.4.3 Time Variables: Evaluating Timed Past Formulas

Next, let us consider timed past formulas of the form $Lasted(\phi, d)$. Such formulas cannot be evaluated using log-variables since we need to quantify how long the sub-formula ϕ held. To enable this, we introduce a variable $time(\phi)$. The intention is that $time(\phi)$ at time point t will be the latest time $t' < t$ such that $\neg\phi$ held at time t'.

This intention is implemented using the following event-statement:

$$\begin{array}{l}
\textbf{when}\;\; \chi(\neg\phi)\;\textbf{do} \\
\qquad time(\phi) := \text{now}
\end{array}$$

Using this time-variable, the *Lasted*-operators can be defined at time t, assuming that $d > 0$, by:

$$\begin{aligned}
\mathit{Lasted}(\phi, d) &\equiv (t \geq \mathit{time}(\phi) + d - 1) \\
\mathit{Lasted_ie}(\phi, d) &\equiv (t \geq \mathit{time}(\phi) + d) \\
\mathit{Lasted_ei}(\phi, d) &\equiv \phi \wedge (t \geq \mathit{time}(\phi) + d - 1) \\
\mathit{Lasted_ii}(\phi, d) &\equiv \phi \wedge (t \geq \mathit{time}(\phi) + d)
\end{aligned}$$

As an example, if $\neg\phi$ at time (t–1) then at time t we have $\mathit{time}(\phi) = (t\text{–}1)$ and thus $\mathit{Lasted}(\phi, d) \equiv (t \geq (t-1) + d - 1) \equiv (t \geq t + d - 2) \equiv 2 \geq d$. This means that $\mathit{Lasted}(\phi, d)$ holds only for $d \leq 2$, which is consistent with the definition of $\mathit{Lasted}(\phi, d)$ in that it excludes both ends (t and $t-1$) of the interval.

3.5 Future Formulas

For Future-time formulas, we cannot define their value through log variables, but must instead represent them as requirements on the future behavior of the system. Thus, we will maintain a formula $\mathcal{G}$, which is represents the requirements and is transformed along with a computation. At each time-step the formula $\mathcal{G}$ will be updated into a formula $\mathcal{G}^+$, representing the requirements for the next time-step. This update procedure depends on the values of defined variables, and system variables. We note that the defined variables represent the state of the system in the previous time instant.

Our goal is then to transform $\mathcal{G}$ into $\mathcal{G}^+$ based on the information given by the value of defined variables at time t and system variables at time t^+. If we let s be the valuation of defined variables, then we can represent this transformation by a function Δ, such that $\mathcal{G}^+ = \Delta(\mathcal{G}, s)$. For conjunctions and disjunctions we have

$$\begin{aligned}
\Delta(\phi_1 \wedge \phi_2, s) &= \Delta(\phi_1, s) \wedge \Delta(\phi_2, s) \\
\Delta(\phi_1 \vee \phi_2, s) &= \Delta(\phi_1, s) \vee \Delta(\phi_2, s)
\end{aligned}$$

In practice, $\mathcal{G}$ will be a conjunction of a number of conjuncts, and we will represent each conjunct and its transformations separately.

3.5.1 Recurrence Relations

In order to define statements that update requirements that contain temporal operators, we use the following definitions and recurrence relations.

$$\begin{aligned}
[\mathit{Next}(\phi)](t) &= [\phi](t^+) \\
[\mathit{Until}(\phi, \psi)](t) &= [\psi](t^+) \;\vee\; \big([\phi](t^+) \;\wedge\; [\mathit{Until}(\phi, \psi)](t^+)\big) \\
[\mathit{Until_i}(\phi, \psi)](t) &= [\psi](t) \vee ([\phi](t) \wedge [\mathit{Until}(\phi, \psi)](t) \\
[\mathit{AlwF}(\phi)](t) &= [\phi](t^+) \wedge [\mathit{AlwF}(\phi)](t^+) \\
[\mathit{AlwF_i}(\phi)](t) &= [\phi](t) \wedge [\mathit{AlwF}(\phi)](t) \\
[\mathit{SomF}(\phi)](t) &= [\phi](t^+) \vee [\mathit{SomF}(\phi)](t^+) \\
[\mathit{SomF_i}(\phi)](t) &= [\phi](t) \vee [\mathit{SomF}(\phi)](t)
\end{aligned}$$

The operators that include the current time-step are defined in terms of their non-inclusive counterparts, so we only define the transformation Δ for the non-inclusive operators.

$$\begin{aligned}
\Delta(\mathit{Next}(\phi), s) &= \textbf{if } [\phi](t^+) \textbf{ then } \mathit{true} \textbf{ else } \mathit{false} \\
\Delta(\mathit{Until}(\phi, \psi), s) &= \textbf{if } [\psi](t^+) \textbf{ then } \mathit{true} \\
&\qquad \textbf{else if } [\neg\phi](t^+) \wedge [\neg\psi](t^+) \textbf{ then } \mathit{false} \\
&\qquad \textbf{else } \mathit{Until}(\phi, \psi) \\
\Delta(\mathit{AlwF}(\phi), s) &= \textbf{if } [\phi](t^+) \textbf{ then } \mathit{AlwF}(\phi) \textbf{ else } \mathit{false} \\
\Delta(\mathit{SomF}(\phi), s) &= \textbf{if } [\phi](t^+) \textbf{ then } \mathit{true} \textbf{ else } \mathit{SomF}(\phi)
\end{aligned}$$

3.5.2 Translation into FIL Statements

In order to translate the above transformation function into FIL statements, we need to define a representation of the formula $\mathcal{G}$ in the FIL language. To do this we take a conjunct $\phi(\phi_F)$ in $\mathcal{G}$ containing a principal future formula ϕ_F. We introduce a FIL variable $req(\phi_F)$ that indicates wether the formula ϕ_F is a member in $\mathcal{G}$ at the next time-step. This boolean variable is set when $\phi(true) \wedge \neg\phi(false)$ holds.

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

This means that $req(\phi_F)$ is set when the subformula ϕ_F is required to hold for the formula $\phi(\phi_F)$ to hold. In addition, we introduce the FIL readout ϕ_F_error which is updated to $Failed$ when ϕ_F is transformed into $false$. We use and maintain $req(\phi_F)$ to update ϕ_F_error.

The FIL statements generated for the $Next$-operator are:

$$\begin{array}{l} \textbf{when } req(Next(\phi)) \textbf{ and } \chi(\phi) \textbf{ do} \\ \quad req(Next(\phi)) := false \end{array}$$

$$\begin{array}{l} \textbf{when } req(Next(\phi)) \textbf{ and } \chi(\neg\phi) \textbf{ do} \\ \quad \textbf{update } Next(\phi)\text{\_error } \textbf{with } \text{Failed} \\ \quad req(Next(\phi)) := false \end{array}$$

The FIL statements generated for the $Until$-operator are:

$$\begin{array}{l} \textbf{when } req(Until(\phi,\psi)) \textbf{ and } \chi(\psi) \textbf{ do} \\ \quad req(Until(\phi,\psi)) := false \end{array}$$

$$\begin{array}{l} \textbf{when } req(Until(\phi,\psi)) \textbf{ and } \chi(\neg\phi \wedge \neg\psi) \textbf{ do} \\ \quad \textbf{update } Until(\phi,\psi)\text{\_error } \textbf{with } \text{Failed} \\ \quad req(Until(\phi,\psi)) := false \end{array}$$

The FIL statement generated for the $SomF$-operator is:

$$\begin{array}{l} \textbf{when } req(SomF(\phi)) \textbf{ and } \chi(\phi) \textbf{ do} \\ \quad req(SomF(\phi)) := false \end{array}$$

The FIL statement generated for the $AlwF$-operator is:

$$\begin{array}{l} \textbf{when } req(AlwF(\phi)) \textbf{ and } \chi(\neg\phi) \textbf{ do} \\ \quad \textbf{update } AlwF(\phi)\text{\_error } \textbf{with } \text{Failed} \\ \quad req(AlwF(\phi)) := false \end{array}$$

For a principal conjunct in $\mathcal{G}$ of the form $Alw(\phi)$, or $AlwF(\phi)$, the variable $req(Alw(\phi))$ will always be set to $true$. For this reason we can do without this variable, and simplify the FIL code to:

$$\begin{array}{l} \textbf{when } \chi(\neg\phi) \textbf{ do} \\ \quad \textbf{update } Alw(\phi)\text{\_error } \textbf{with } \text{Failed} \end{array}$$

3.5.3 Nested Future Formulas

When translating the requirements to FIL-statements, we have not accounted for nested future formulas. To do this, we consider a conjunct $\phi(\phi_F(\phi_G))$ of $\mathcal{G}$ where $\phi_F = \phi_F(\phi_G)$ and ϕ_G both are principal future formulas. Next we generalize the transformation function Δ by using the functions $toTrue$ and $toFalse$ to represent the conditions of the different operators.

$$\begin{array}{rl} \Delta(\phi_F) = & \textbf{if } [toTrue(\phi_F)](t^+) \textbf{ then } true \\ & \textbf{else if } [toFalse(\phi_F)](t^+) \textbf{ then } false \\ & \textbf{else } \phi_F \end{array}$$

This definition can be derived from the recurrance:

$$[\phi_F](t) \;=\; [toTrue(\phi_F)](t^+) \,\vee\, [\neg toFalse(\phi_F)](t^+) \;\wedge\; [\phi_F](t^+)$$

This recurrence implies that the $toTrue(\phi_F)$ subformula may not be a future formula, since it is a disjunct in a disjunction with a known future formula ϕ_F as another disjunct.

The next step is to expand the general form of the transformation function to allow nesting in $toFalse(\phi_F)$. This is done by using a recursive form of Δ where if not $toTrue$ holds then $toFalse$ must not hold, and ϕ_F must continue to hold.

$$\begin{aligned}\Delta(\phi_F) = \;& \textbf{if } [toTrue(\phi_F)](t^+) \textbf{ then } true \\ & \textbf{else } \Delta(\neg toFalse(\phi_F), s) \wedge \phi_F\end{aligned}$$

Finally we generate FIL statements; the FIL statements from $\Delta(\neg toFalse(\phi_F), s)$ are generated as described in section 3.5.2. On top of these statements the following are generated to check ϕ_F:

when $\phi(true) \wedge \neg\phi(false)$ **do**
    $req(\phi_F) := true$

when $req(\phi_F)$ **and** $\chi(toTrue(\phi_F))$ **do**
    $req(\phi_F) := false$

3.5.4 Timed Future Formulas

For timed future formulas we use freeze-quantification, described in section 3.6. The following equivalences are used for this:

$$\begin{aligned}
Lasted(\phi, d) &\equiv x.Until(\phi, y.(y \geq x{+}d{-}1)) \\
Lasted_ie(\phi, d) &\equiv \phi \wedge Lasted(\phi, d) \\
Lasted_ei(\phi, d) &\equiv Lasted(\phi, d{+}1) \\
Lasted_ii(\phi, d) &\equiv \phi \wedge Lasted(\phi, d{+}1)
\end{aligned}$$

3.6 Freeze Quantified Formulas

The evaluation $[x{:=}\tau.\phi(x)](t)$ of a freeze quantified formula can be defined by $[\phi(v)](t)$, where $v = [\tau](t)$. The special case of freeze quantification of the current time $x.\phi(x)$ can be defined as $\phi(v)$, where $v = [now](t)$. These definitions implies that non-temporal formulas can be implemented by substituting x for τ.

For a principal future sub-formula $\phi_F(x)$ of $\phi(x)$ we need to store the value $v_{x,t} = [\tau](t)$ when adding $\phi_F(v_{x,t})$ to $\mathcal{G}$. Using a variable to store $v_{x,t}$ we would need a different variable for each time-step t when $\phi_F(v_{x,t})$ is added to $\mathcal{G}$. Since FIL does not provide indexing of variables we would also have to generate different event-statements for each such FIL variable.

3.6.1 Operator Monotonicity and Parameter Logging

The problem with the above approach is that a large number of parameterized formulas may be added to $\mathcal{G}$, with a large number of different parameter values. This problem can be handled in several ways. Let us assume that the formulas of $\mathcal{G}$ are conjuncts.

1. One way is to consider only a suitably large subset of the parameter values. E.g., one could use the parameter value for the first instant in time when the formula is added, for the last instant in time (if there is one), and for some random intermediate instants in time.

2. In some cases, we can eliminate some of the formulas in $\mathcal{G}$ as redundant, in case they are entailed by others. For example, if $\mathcal{G}$ contains both the formula $\phi(a)$ and $\phi(b)$, for some values a and b of the parameter, and if $\phi(a) \implies \phi(b)$ is a valid implication, then $\phi(b)$ is a redundant conjunct, and may be discarded.

To pursue the second alternative, we define a notion of *monotonicity* in temporal operators. Typically, parameters of temporal formulas will be values of real time, or some measured values. Sometimes we may decide entailment based on comparison of measured values.

So, let us consider the ordering $\leq$ on real numbers, and the ordering on booleans where *false* $\leq$ *true*. We will call an arithmetic 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 then observe that all future-time temporal operators (*Until*, *AlwF*, *SomF*) are monotonic in all of their arguments, i.e., if the argument becomes "truer", then the formula becomes "truer". The same is true for conjunction and disjunction.

The notion of monotonicity is used as follows: Whenever a new formula $\phi(b)$ is created, such that $\phi(a)$ with $a \leq b$ is already a conjunct, then

- if $\phi(x)$ is increasing in x, then $\phi(b)$ is discarded, and
- if $\phi(x)$ is decreasing in x, then $\phi(a)$ is discarded.

Let us consider some examples.

1. The formula $\mathit{Lasts}(\phi, d)$ can be defined as x . $\mathit{Until}(\phi, (now \geq x+d-1))$. Here $now \geq x+d-1$ is decreasing in x. Thus, if we generate a requirement of form $\mathit{Until}(\phi, (now \geq x+d-1))$ for some parameter value of x, then we can discard all other requirements of this form for smaller values of x, that were generated earlier and are still present.

2. The more complicated formula

$$Always \left[\begin{array}{ll} & Previous(\neg resume) \ \wedge \ resume \ \wedge \ speed < ref \\ \Longrightarrow & \\ & v := speed.\ x.\ \mathit{SomF}(speed = ref \ \wedge \ \dfrac{speed - v}{now - x} = 0.5km/h/s) \end{array} \right]$$

is neither increasing nor decreasing in x. However, its antecedent is a formula which is not true for consecutive time points, and hence there is hope that not too many parameterized formulas will be generated.

Chapter 4

Evaluation of Case Studies

4.1 Introduction

To evaluate the practical usefulness of the generated test oracles, we developed two case studies from actual specifications used at Volvo. The reasons that we developed two case studies are that the cruise control specification have previously been converted to TRIO ([4]), but the fault-injection equipment used at Volvo is currently tuned to test the throttle.

4.2 Cruise Control Module

The Cruise Control Module (CCM) specification is described in appendix C. This is the TRIO specification presented in [4], modified for the purpose of test oracle generation and using freeze-quantifiers to describe some requirements not described in [4]. Since we had no CCM available to us at the time of testing, these requirements have not been run in a realistic environment. We did however generate an oracle, and when inspecting the oracle code we found no errors or ambiguities.

4.3 Electronic Throttle Module

4.3.1 Description

The Electronic Throttle Module (ETM) is used to control the throttle of a car, which in turn controls the air flow into the engine. The throttle angle is controlled to a position as requested by the Engine Control Module (ECM). The throttle is connected to the ETM's digital and analog I/O, and both the ETM and the ECM are connected to a CAN bus.

4.3.2 ETM Testing Setup

When testing the ETM we did not use an ECM; instead we generated the CAN signals from the ECM to the ETM using a *CANalyzer*, see figure 4.1. The CANalyzer is a software package used to analyze and generate CAN bus traffic. This software was run on a laptop with a PC-card CAN adapter. Some of the signals generated are necessary to keep the ETM alive, the others are generated by pressing keys on the CANalyzer laptop to execute test cases.

The test oracles generated were run on a fault injection equipment, connected to the CAN bus. Both the CANalyzer and the FI equipment were run on an MS Windows platform.

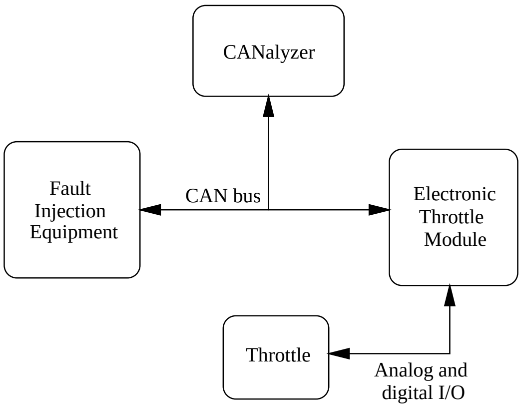

Figure 4.1: Setup for ETM testing

4.3.3 Requirements

Note that the requirements described here are not the exact requirements from an actual Volvo specification. The requirements that were tested all define fault diagnosis signals. There are three signals that have been defined and tested using our tool to generate test oracles.

The signal $SFT10$ should be set if the brake lights are on and the brake pedal is not pressed (this is a consistency check). The signal $SFT13$ indicates that the requested throttle angle is lower that 0% or higher than 100% of the specified range. $SFT16$ is activated if the cruise control is active even though the No Cruise Control reconfiguration is active. The signals used to define the requirements are:

Signal	Description
blsw	Brake Light Switch active.
bpa	Brake Pedal Active (pressed).
rta	Requested Throttle Angle (percent).
cca	Cruise Control Active.
NCCT	No Cruise Control reconfiguration activated.

The requirements in TRIO, based on these signals:

$$sftr10 : SFT10 \leftrightarrow SomP(Lasted(blsw \wedge \neg bpa, CT10))$$

The fault diagnosis signal $SFT10$ should only be set if at some previous time the brake light switch $blsw$ was set and the brake pedal active bpa was not for a time $CT10$.

$$sftr13 : SFT13 \leftrightarrow SomP(Lasted(rta < 0 \vee rta > 100, CT13))$$

The signal $SFT13$ should be set only if at some previous time the requested throttle angle rta was less than 0% or greater than 100% for a time $CT13$.

$$sftr16 : SFT16 \leftrightarrow SomP(Lasted(NCCT \wedge cca, CT16))$$

The signal $SFT16$ should be set when the cruise control have been active cca in the no cruise control reconfiguration $NCCT$ for a time $CT16$.

There is however a problem with this way of defining the fault diagnosis signals. When a signal from the ECM, say bpa, is sent over the CAN bus, it will be seen at approximately the same time by both the ETM and the fault injection equipment. However, when a fault diagnosis signal is set in the ETM then the fault injection equipment will not know about it immediately. The delay introduced depends on the periodicity of the signal, as well as the propagation time over the CAN network and through the system buffers.

To solve this, we divide each requirement into two. For $sftr10$ the resulting requirements are:

$$sftr10a : SFT10 \rightarrow SomP(Lasted(blsw \wedge \neg bpa, CT10))$$

$$sftr10b : SomP(Lasted(blsw \wedge \neg bpa, CT10)) \rightarrow WithinF(SFT10, Delay)$$

The other requirements are divided the same way. The requirement $sftr10a$ describes what must have happened for the signal $SFT10$ to become set, and $sftr10b$ describes under what conditions $SFT10$ must be set. The $sftr10b$ requirement introduces the delay ($Delay$) from when $SFT10$ should become set until such time that this change must be visible to the FI-equipment.

4.3.4 Test Cases

We have designed one test case for each of the requirements. The test cases where executed on the CANalyzer.

The requirement $sftr10$ is tested by setting the signal $blsw$ and then reseting bpa. The result of this should be that $SFT10$ is set after some time $CT10$.

$sftr13$ is tested by setting rta to -10. This should mean that $SFT13$ becomes set after the time $CT13$.

$sftr16$ is tested by entering the $NCCT$ reconfiguration, for example by triggering $SFT13$, and then setting cca. This should set $SFT16$ after the time $CT16$.

These test cases should not report any violations of the requirements, since the ETM is in production. If they do report violations, signals are logged so that we can double check if the requirements were met and the test oracle was wrong. Of interest is also how many requirements we can check for a time-step setting.

4.3.5 Test Results

The test oracle did occasionally report an error, but at most for a single time-step. This is because the fault diagnosis signals $SFTn$ are sent on the CAN bus every 100 ms, and we have a time-step of 100 ms, so there is no margin for signal propagation. Except for this we found no inconsistencies while testing.

The times we measured turned out to be unreliable, i.e. the same tests gave significantly different results at separate occasions, so unfortunately we have no numerical values of interest to present. The oracle for the ETM requirements (a total of eight) were successfully run with a time-step of 25 ms, and the oracle for the CCM with 25 requirements where run with a time-step of 50 ms.

4.3.6 Conclusions

The test oracles generated can be used to check if requirements are met, but one must be careful about the timing issues. Both the fault injection equipment's capacity and the properties of the observable signals influence the selection of time-step.

To get around this additional complexity, it would be desirable to incorporate these concerns into the tool. It should be possible to do this using simple timing properties of the test-system and the respective signals.

Another interesting improvement would be to report what dynamic requirements are still active when the testing is aborted. The reason for this is that testing of course cannot go on forever.

Chapter 5

Towards Automated Testing

The main goal of this work is to enable automation of the test process. The generation of test oracles is a step in this direction, but further work is needed.

The main goal of this project is to enable automation of the entire testing process. The generation of test oracles is a step in this direction, but further work is needed. The problem of automated testing has been described previously in [5], and much of the discussions here is based on that work.

5.1 The Testing Process

There are three purposes of the testing process:

1. To demonstrate that the software functions as specified
2. To find failures in the software
3. To exercise all parts of the software

To meet these purposes, the tester will go through three steps: generating test cases (section 5.2), executing test cases (section 5.3) and finally evaluating these tests (section 5.4). There are however some prerequisites: we need a specification for all three steps, and we also need the software to be able to execute and evaluate the tests.

5.2 Test Generation

The fact that test cases can be generated based solely on the specification might lead one to do this in parallel with software development. To do this, test case generation must be automated as specifications might change during development and thus render manually generated test cases invalid.

A test case can be described by an identifier, the requirements exercised by the test case, a list of named inputs and sequences of values for these inputs. Test cases are automatically generated by treating the specification as a knowledge base, and then applying test case design rules. If the specification changes, new test cases can be designed, generated and documented by the test generation tool. A test case generator can be compared to a compiler, reading and analyzing input and using the test case design rules to generate output.

The design rules for generating test cases can be categorized into five categories: action driven, data driven, logic driven, event driven and state driven. A selection of these categories is described below. After this we suggest a design rule (section 5.2.4) based on the information gathered while generating the test oracle.

5.2.1 Action-Driven Testing

In action driven testing, for each action at least one normal and at least one abnormal test case must be generated. The normal test case will detect missing actions. To generate abnormal test cases for an action we need to select abnormal input.

5.2.2 Data-Driven Testing

History shows that most software failures occur when applying an input at its lowest or highest valid value. Generating test cases from this is called boundary analysis, which simply means that test cases are generated for these boundary input values. Another method for data-driven testing is called partition analysis, and means that input domains are divided into partitions, and test cases are generated for these partitions. One method of partitioning the input domains is to consider values that are handled equally by the specification as values in the same partition. The defects detected by data-driven design are incorrect data processing.

5.2.3 State-Driven Testing

In state-driven testing every state or context in which actions are allowed is verified, by generating test cases with inputs that exercise state transitions in valid and invalid sequences. State-driven testing is used to detect incorrect state sequencing.

5.2.4 Example Design Rule

One possible design rule for a requirement of the form $Alw(in \rightarrow out)$ would be to set the input signal in, because it must be set for the requirement to be tested. Using rules like this we can generate a set of input values for which the requirement could be violated.

One way to do this is to use the notion of monotonicity over formulas, as described in chapter 3. The rule would be to analyze the monotonicity of the requirement with respect to input values, and select input values that make the requirement falser.

The resulting set of interesting input values could be quite large (infinite in many cases), so other rules need to be applied to select a subset of this set.

5.3 Test Execution

Executing test cases means to run the software under test and apply inputs documented in the test case to that software. When the software processes its input, it will generate output that can be compared to the expected system behavior.

Capture-replay tools are common, they create test scripts by recording input. It would probably be possible to generate scripts directly from the test case generator to execute the test cases.

5.4 Test Evaluation

The tests are ended by evaluating how thoroughly the purposes of testing were satisfied:

1. How completely was the specified behavior demonstrated?

2. How many failures were found in the software?

3. How many statements, branches and paths where exercised?

To answer these questions testers need to evaluate each test case, to discover if the software passes or fails this test case, and evaluate how well the test cases demonstrate requirements. The pass/fail evaluation can be automated, as in this project, using a test oracle to evaluate system response compared to specifications.

Evaluation of how well test cases demonstrate requirements is done using metrics such as requirements coverage, failure coverage and code coverage. The requirements coverage metric can easily be formulated as the ratio of passed requirements over the total number of requirements.

5.5 Conclusions

The method described in this project can be used to generate oracles from specifications, making it possible to automatically verify that executed test cases conform to specifications. There are however some considerations. One is the selection of the time-step used. Too long time-steps means we might miss events, and to short means test execution systems might not cope, and requirements have to be rewritten to account for periods and response times.

Also these issues could be automated. The oracle generator would need to know about signal periods and response times of different parts, such as the test execution system. To know these response-times we would need to use some real-time OS.

To fully automate test evaluation, we also need to evaluate how well the executed test cases cover the specification. However, the next natural step towards automated testing would be the generation of test cases, since we need test cases to evaluate.

As noted in section 5.2, test generation tools read and analyze the specification, using design rules to generate test cases. As the test oracle generator also read and analyze the specification, these two tools could be merged into one, using different rules for test case generation and test oracle generation.

Bibliography

[1] Rajeev Alur and Tomas A. Henzinger. Logics and models of real time: A survey. In *Real Time: Theory in Practice, Lecture Notes in Computer Science 600*. Springer-Verlag, 1992.

[2] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. 15th Work. Protocol Specification, Testing, and Verification*, Warsaw, June 1995. North-Holland.

[3] Y. Kesten, A. Pnueli, and L. o. Raviv. Algorithmic verification of linear temporal logic specifications. In *Proc. ICALP '98*, Lecture Notes in Computer Science. Springer Verlag, 1998.

[4] Johan Nielsen. Real-time specification using the trio language. Master's thesis, Royal Institute of Technology, 1998.

[5] Robert M. Poston. *Automating Specification-Based Testing*. IEEE Computer Society, 1996.

[6] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc.* 1^{st} *IEEE Int. Symp. on Logic in Computer Science*, pages 332–344, June 1986.

Appendix A

The TRIO Language Dialect

The TRIO dialect supported by this tool includes definition of macros and constants. It is not a proper subset of TRIO, as described in section A.3.

A.1 Syntax

A.1.1 Terms

A *term* is a syntactical element denoting a value, and can be constructed from variables x_i, constants c_j, functions f and subterms $\tau_1, \ldots, \tau_n$.

$$
\begin{array}{ll}
\text{term } \tau ::= & x_i \\
& c_j \\
& \tau_1 + \tau_2,\ \tau_1 - \tau_2,\ \tau_1 * \tau_2,\ \tau_1/\tau_2 \\
& f(\tau_1, \ldots, \tau_n)
\end{array}
$$

Simple example terms:

17	A constant value.
foo	A variable.
$(foo + bar)/2$	Calculate average of two variables.
$percent(speed, maxSpeed)$	A function application.

A.1.2 Formulas

A *formula* is a syntactical element denoting a truth-value, evaluated to either *true* or *false*. From propositions p, predicates P, terms τ_1, τ_2 and subformulas $\phi_1, \ldots, \phi_n$ we can build formulas as below.

$$
\begin{array}{ll}
\text{formula } \phi ::= & \mathit{true} \\
& \mathit{false} \\
& p \\
& P(\phi_1, \ldots, \phi_n) \\
& \tau_1 < \tau_2,\ \tau_1 \leq \tau_2,\ \tau_1 > \tau_2,\ \tau_1 \geq \tau_2,\ \tau_1 = \tau_2,\ \tau_1 \neq \tau_2 \\
& \neg\phi_1 \\
& \phi_1 \wedge \phi_2, \phi_1 \vee \phi_2, \phi_1 \rightarrow \phi_2 \\
& x_i.\phi_1 \\
& x_i := \tau.\phi_1 \\
& \text{temporal operators, se section A.2.2}
\end{array}
$$

Simple example formulas:

$auto$	A proposition.
$3 \leq foo$	Compares two terms.
$\neg auto$	Logical negation.
$(3 \leq foo) \rightarrow (\neg auto)$	Logical implication.
$Until(3 \leq foo, auto)$	Temporal operator.

Freeze quantification ($x_i.\phi$, $x_i := \tau.\phi$) is described in section A.2.3, temporal operators in section A.2.2.

A.2 Semantics

A.2.1 Evaluations

TRIO is a timed logic, using discrete time-steps for defining it's semantics. To describe an evaluation we will write $[\tau](t)$ for a term τ at time-step t, and $[\phi](t)$ for a formula ϕ at time-step t. This means that the value of $[\phi](t)$ is the evaluation of the formula ϕ at time-step t in the current execution. We will also use t^+ as an abbrevation for the next time-step after t, which is $t{+}1$, and t^- for the previous time-step $t{-}1$.

Formula ϕ	Evaluation $[\phi](t)$
$true$	True.
$auto$	Value of proposition $auto$ ($true$ or $false$).
$3 \leq foo$	True only if 3 is smaller than or equal to the value of foo.
$\neg auto$	True only if $auto$ is false.
$p_1 \wedge p_2$	True only if both p_1 and p_2 are true.
$\phi_1 \vee \phi_2$	$[\neg(\neg\phi_1 \wedge \neg\phi_2)](t)$
$\phi_1 \rightarrow \phi_2$	$[\neg\phi_1 \vee \phi_2](t)$

A.2.2 Temporal Operators

Some of these operators are associated with time intervals. These intervals are normally open in both ends, but there are alternative versions. These versions of the operators are indexed with two 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, $Lasted(f,t)$ is associated with open intervals but $Lasted_{ie}(f,t)$ is associated with intervals closed at the left end point.

Previous

The formula $Previous(\phi)$ holds at a time-step t if the formula ϕ held at the previous time-step t^-.

$$[Previous(\phi)](t) \equiv [\phi](t^-)$$

Next

The formula $Next(\phi)$ holds at a time-step t if the formula ϕ will hold at the next time-step t^+.

$$[Next(\phi)](t) \equiv [\phi](t^+)$$

Until, UntilW

The formula $Until(\phi, \psi)$ holds if the formula ϕ holds until the formula ψ holds. The formula ψ must eventually hold for this to be true, this condition is removed in the $UntilW$ operator.

$$[Until(\phi, \psi)](t) \equiv \begin{array}{l} [\psi](t') \text{ for some } t' > t, \text{ and} \\ [\phi](t'') \text{ for all } t'' \text{ such that } t < t'' < t'. \end{array}$$

$$[UntilW(\phi, \psi)](t) \equiv \quad [Until(\phi, \psi)](t) \text{ or } [AlwF(\phi)](t).$$

Since, SinceW

The formula $Since(\phi, \psi)$ holds if the formula ϕ has held since the formula ψ held. The formula ψ must have held in the past for this to be true, this condition is removed in the $SinceW$ operator.

$$[Since(\phi, \psi)](t) \equiv \begin{array}{l} [\psi](t') \text{ for some } t' < t, \text{ and} \\ [\phi](t'') \text{ for all } t'' \text{ such that } t' < t'' < t. \end{array}$$

$$[SinceW(\phi, \psi)](t) \equiv [Since(\phi, \psi)](t) \text{ or } [AlwP(\phi)](t).$$

Som, SomF, SomP

The formula $Som(\phi)$ holds if the formula ϕ holds at some time-step. The formula $SomF(\phi)$ holds if ϕ holds at some future time-step. The formula $SomP(\phi)$ holds if ϕ held at some time-step in the past.

$$[Som(\phi)](t) \equiv [\phi](t') \text{ for some } t'.$$

$$[SomF(\phi)](t) \equiv [\phi](t') \text{ for some } t' > t.$$

$$[SomP(\phi)](t) \equiv [\phi](t') \text{ for some } t' < t.$$

Alw, AlwF, AlwP

The formula $Alw(\phi)$ holds if the formula ϕ holds at some time-step. The formula $AlwF(\phi)$ holds if ϕ holds at some future time-step. The formula $AlwP(\phi)$ holds if ϕ held at some time-step in the past.

$$[Alw(\phi)](t) \equiv [\phi](t') \text{ for all } t'.$$

$$[AlwF(\phi)](t) \equiv [\phi](t') \text{ for all } t' > t.$$

$$[AlwP(\phi)](t) \equiv [\phi](t') \text{ for all } t' < t.$$

Before

The formula $Before(\phi, \psi)$ holds if ψ holds at some time-step, and ϕ held at some time-step before then.

$$[Before(\phi, \psi)](t) \equiv [\psi](t') \text{ for some } t', \text{ and } [\phi](t'') \text{ for some } t'' < t'.$$

Becomes

The formula $Becomes(\phi)$ holds if the formula ϕ became true at this time-step, after having been false at the previous time-step.

$$[Becomes(\phi)](t) \equiv [\phi](t) \text{ and } [\neg\phi](t^-)$$

Within, WithinF, WithinP

The formula $Within(\phi, d)$ holds if the formula ϕ holds at some time-step within d time-steps from now. $WithinF(\phi, d)$ holds if ϕ holds within d time-steps in the future, and $WithinP(\phi, d)$ holds if ϕ holds within d time-steps in the past.

$$[Within(\phi, d)](t) \equiv [\phi](t') \text{ for some } t' \text{ such that } t - d < t' < t + d.$$

$$[WithinF(\phi, d)](t) \equiv [\phi](t') \text{ for some } t' \text{ such that } t < t' < t + d.$$

$$[WithinP(\phi, d)](t) \equiv [\phi](t') \text{ for some } t' \text{ such that } t - d < t' < t.$$

Lasted

The formula $Lasted(\phi, d)$ holds if the formula ϕ held at all time-steps within d time-steps in the past.

$$[Lasted(\phi, d)](t) \equiv [\phi](t') \text{ for all } t' \text{ such that } t - d < t' < t.$$

Lasts

The formula $Lasts(\phi, d)$ holds if the formula ϕ holds at all time-steps within d time-steps in the future.

$$[Lasts(\phi, d)](t) \equiv [\phi](t') \text{ for all } t' \text{ such that } t < t' < t + d.$$

NextTime

The formula $NextTime(\phi, d)$ holds if the next time-step that ϕ holds is d time-steps in the future.

$$NextTime(\phi, d) \equiv [\phi](t + d) \text{ and } [\neg\phi](t') \text{ for all t' such that } t < t' < t + d.$$

LastTime

The formula $LastTime(\phi, d)$ holds if the last time-step that ϕ held was d time-steps in the past.

$$LastTime(\phi, d) \equiv [\phi](t - d) \text{ and } [\neg\phi](t') \text{ for all t' such that } t - d < t' < t.$$

A.2.3 Freeze Quantification

Freeze quantification is used to compare values observed at different points in time. A freeze quantified formula $x := \tau.\phi$ is a formula where the value of x is frozen to the value of τ at the current time-step before ϕ is evaluated.

The evaluation of a freeze quantification $x := \tau.\phi(x)$ is written $[x := \tau.\phi(x)](t)$, and defined by $[\phi(v)]$ for a value $v = [\tau](t)$. We also use $x.\phi$ as an abbrevation for $x := now.\phi$, where *now* denotes the current time-step.

As an example, $x.Until(\phi, now \geq x{+}10)$ means that ϕ must hold until $now \geq x{+}10$ holds, which will happen after ten time-steps.

$$\begin{aligned}[x.Until(\phi, now \geq x{+}10)](t) \equiv\ & [now \geq [now](t){+}10](t') \text{ for some } t' > t, \text{ and} \\ & [\phi](t'') \text{ for all t" such that } t < t'' < t'. \\ \equiv\ & [t' \geq t{+}10](t') \text{ for some } t' > t, \text{ and} \\ & [\phi](t'') \text{ for all t" such that } t < t'' < t'.\end{aligned}$$

Freeze quantification is also described in [1].

A.3 Not really TRIO

In TRIO all operators are defined by first order quantifiers ($\forall$ and $\exists$) and $Dist(\phi, d)$, an operator which holds at time t if ϕ holds at time $t + d$. Our tool can't really handle the genericity of this solution, so we limit the language to some predefined temporal operators and replace the quantifiers with the freeze-quantifiers we have introduced.

Appendix B

The FIL Language

The FIL language was designed for use at Volvo for fault-injection and is executed on their fault-injection apparatus, the FI-system. A FIL program is described by definition, experiments and a campaign. The interesting part for our purpose is the experiment, which consists of a number of event-statements that can react to changes in the FI-system environment. Event statements are described in section B.2. These are all the parts of a FIL program:

- (Optional) Type definitions
- (Optional) Constant definitions
- Fault definitions
- (Optional) Readout definitions
- (Optional) Control definitions
- Experiment definitions
- Campaign statement

B.1 FIL Program parts

B.1.1 Type Definitions

A userdefined type definition in FIL is an enumeration names of elements in a domain, where the names are separated by commas:

enumerationdef id **as** [*names*]

B.1.2 Constant Definitions

A constant in FIL is defined as the value of an expression:

constantdef id **oftype** type **as** *exp*

B.1.3 Fault Definitions

Definitions of faults, not used by oracles but a required part of a FIL program. Defines faults that can be injected using the FI equipment.

B.1.4 Readout Definitions

A readout is an object that samples measurements from the environment, i.e. sensors, actuators or signals on the CAN bus. *update_mode* is one of **onchange** , **usercontrolled** or **sampleintervall** *expression.*

readoutdef id **oftype** type **as**
  readout_type(*opt_parameters*)
  updatemode *update_mode*

B.1.5 Control Definitions

A FIL control is an output object, used to control the FI-system environment.

controldef id **oftype** type **as**
  control_type(*opt_parameters*)

B.1.6 Experiment Definitions

An experiment consists of a number of event-statements that can react to changes in the FI-system environment.

experimentdef id(*opt_arguments*) **as**
  fault_instance
  opt_variable_definitions
  event_statements

B.1.7 Campaign Statements

A campaign defines a set of experiments.

campaign
  experiment id(*opt_parameters*)

or:

campaign
  for id **oftype** type **in** *generate_range*
  generate
    experiment_instances
  end

B.2 Event Statements

The *event_statements* are the executable parts of the experiment definitions. The syntax for an *event_statement* is:

when *event_expression* **do**
  opt_variable_definitions
  statements

An *event_statement* consists of a boolean expression (*event_expression*), local variable definitions (*opt_variable_definitions*) and sequential *statements.* The semantics of the *event_statement* is:

- The *statements* of the *event_statement* are executed when the value of the *event_expression* changes from false to true. This means that the *event_statement* is not a sequential construct, *event_statements* are executed in parallell (or simulated parallell).
- The *event_expression* of each *event_statement* is initialized to false when the experiment is started, with the result that all *statements* in *event_statements* whose *event_expression* is true when the experiment is started are executed immediately (in the first *delta timepoint*).
- All *event_statements* that become true at the same *delta timepoint* are executed within that *delta timepoint*, with the same values of input, global variables, readouts and timers. A result from this is that values assigned to a variable at some *delta timepoint*are not visible until the next *delta timepoint*.
- If the execution of an *event_statement* makes an other *event_statement*'s *event_expression* become true, that other *event_statement* will be executed at the next *delta timepoint*.
- Assignments to global variables are postponed until all *event_statements* of this *delta timepoint* have been executed.
- Only one *event_statement* may assign a value to a global variable at a given *delta timepoint*. Without this restriction it would be impossible to predict which value the variable should have at the next *delta timepoint*. Assigning more than one value at any *delta timepoint* to a global variable generates a runtime error.

Appendix C

Cruise control case studie

The requirements specified here are derived from requirements specified in [4]. They have been changed somewhat to increase readbility. The requirements 17 and 18 have been numbered 18 and 17, respectively. These requirements are not actual requirements for an actual vehicle.

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 [4], the signals *ccws* and *resuming* are introduced to support requirements specification, and need not be part of an implementation.

The system signals are described in tables C.1 to C.4. 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

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 C.1: ECUs involved in CCM operation

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 C.2: 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 C.3: 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 C.4: Requirement support signals

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

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

Predicates

$$SomF(formula) \equiv Until(true, formula)$$

$$AlwF(formula) \equiv \neg SomF(\neg formula)$$

$$Lasted\_ii(formula, time) \equiv x.Since(formula, y.(x - y > time))$$

$$PosEdge(formula) \equiv \neg Previous(formula) \wedge formula$$

$$NegEdge(formula) \equiv Previous(formula) \wedge \neg formula$$

$$Edge(formula) \equiv PosEdge(formula) \vee NegEdge(formula)$$

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

$$AtLeastOnceSince(formula_1, formula_2) \equiv \\ \neg Since(\neg formula_1 \wedge \neg formula_2, formula_2)$$

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.

$$r3 : \neg ccont \wedge PosEdge(cconoff) \wedge \neg Edge(igsw) \wedge \neg ccd \rightarrow SetTrue(ccont)$$

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 $dcca5$ to $dcca14$ 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.

$$r9 : (atglp = Park \vee atglp = Neutral \vee atglp = Reverse) \rightarrow \neg cca$$

$$dcca9 : atglp = Park \vee atglp = Neutral \vee atglp = Reverse$$

Requirement 10

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

$$r10 : \neg(atglpqf = Good \wedge vsaqf = Good \wedge vsqf = Good) \rightarrow \neg cca$$

$$dcca10 : \neg(atglpqf = Good \wedge vsaqf = Good \wedge vsqf = Good)$$

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.

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

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

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.

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

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.

$$smooth : v := ccws.(t.Next(abs(acceleration(v, t)) < maxSmoothAcc))$$

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*.

$$r27 : (accseti \vee accsetd) \wedge PosEdge(cca) \rightarrow SetTo(ccws, vs)$$

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.

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

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.

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

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.

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

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.

$$r36 : (vs = ccsp \lor \neg cca \lor cco \lor acc \lor dec) \rightarrow SetFalse(resuming)$$

Appendix D

Example Files

This appendix contains the actual text-files used in the example of chapter 2.

D.1 TRIO specification

```
/* Microwave Oven - Example
 * System Requirements in TRIO.
 */

class Micro

domains
  oven_state  : { off, standby, on  }

td items

  propositions
    start    /* Input signal: Start button. */

  values
    ONE_HOUR : temporal    /* Constant */
    timer    : temporal
    state    : oven_state

axioms
  r1: ~Lasted(state = on, ONE_HOUR)
  r2: (timer > 0 & state = standby & start) -> Until(state = on, timer = 0)
  sys: Alw(r1 & r2)

end Micro
```

D.2 FIL oracle

This is the generated FIL oracle. To be complete it needs signal definitions for *start*, *timer* and *state*, as well as a constant definition for ONE_HOUR.

```
# Check axioms of 'sys'.
```

```
# Define requirement type.
ENUMERATIONDEF Require AS [ Ok, Failed ]

# Define boolean values.
CONSTANTDEF FALSE OFTYPE integer AS 0
CONSTANTDEF TRUE  OFTYPE integer AS 1


# Define delta-time span.
CONSTANTDEF deltaTimeSpan OFTYPE time AS 100 ms
CONSTANTDEF firstTimeStep OFTYPE time AS 100 ms

# Insert CAN-definitions here.

READOUTDEF r1_error OFTYPE Require AS
userdef()
UPDATEMODE usercontrolled

READOUTDEF r2_error OFTYPE Require AS
userdef()
UPDATEMODE usercontrolled


# Oracle experiment for 'sys'.
EXPERIMENTDEF sys() AS
FAULT    sys_fault
TIMER    total_time
VARIABLE next_step OFTYPE Time
VARIABLE now OFTYPE Integer
VARIABLE new_step OFTYPE Integer
VARIABLE time_r1Lasted1 OFTYPE Integer
VARIABLE req_r2Until1 OFTYPE Integer


WHEN total_time >= 0 s DO
STARTLOGGING r1_error
STARTLOGGING r2_error
START total_time
next_step := firstTimeStep
now := now + 1

WHEN now >= 1 AND total_time >= next_step DO
next_step := next_step + deltaTimeSpan
now := now + 1
new_step := TRUE

WHEN new_step = TRUE DO
new_step := FALSE

WHEN total_time >= 10 s DO
EXIT

# Check r1
WHEN (now >= ((time_r1Lasted1+ONE_HOUR)-1) AND new_step = TRUE) DO
```

```
UPDATE r1_error WITH Failed

# Store time_r1Lasted1
WHEN (state /= on AND new_step = TRUE) DO
time_r1Lasted1 := now


# Check req_r2Until1
WHEN (req_r2Until1 = FALSE AND timer > 0 AND state = standby AND start = TRUE AND new_step = TRUE) D
req_r2Until1 := TRUE

WHEN (req_r2Until1 = TRUE AND timer = 0 AND new_step = TRUE) DO
req_r2Until1 := FALSE

WHEN (req_r2Until1 = TRUE AND state /= on AND timer /= 0 AND new_step = TRUE) DO
req_r2Until1 := FALSE
UPDATE r2_error WITH Failed


CAMPAIGN
EXPERIMENT sys()
```