In: G. Biswas et. al. (eds.), DX'07, 18th International Workshop on Principles of Diagnosis. May 29-31, 2007. Nashville (USA)

Obtaining Models for Test Generation from Natural-language-like Functional Specifications

M. W. Esser[1], P. Struss[1,2]

[1]Technische Universität München, Boltzmannstr. 3 D-8578 Garching, Germany
[2]OCC'M Software, Gleissentalsstr. 22, D-82041 Deisenhofen, Germany
{esser, struss}@in.tum.de, struss@occm.de

Abstract

The paper presents first results of a project that aims at a model-based tool for functional testing of control software for passenger vehicles. The objective is that this tool can be used in today's engineering practice and, hence, the approach must not require costly changes in the current test generation process and not assume data and skills that do not exist in reality. Firstly, the input to test generation, i.e. the specification of functional requirements, exists mainly as natural language text, rather than in a formal language. Secondly, its output, i.e. a set of proposed tests, has to be justified and explained in terms that are comprehensible to the test engineers in order to allow them to inspect, revise, and complement it. We focus on design decisions that are motivated by this objective. The proposed solution offers a natural-language-template-based interface for acquiring software requirements. The content of the filled-in templates can be represented in propositional logic and temporal relations and form the model of the intended correct behavior. Models of potential faulty behaviors are generated from this OK model by a number of (types of) transformations. The fault types are defined mainly to match the intuition behind manually generated test cases and, hence, can deliver similar, but more systematic, test suites. This forms the basis for the intuitive justification of the tests and its manual post-processing.

1. Introduction

As long as formal specification of software functions and software generation and verification based on formal specifications are not established, testing of software is an essential step in software development. And even if, some day, the precondition would be achieved, there would be no guarantee that the starting point, the specification, captures the intuitive user expectations appropriately, which again establishes a need for testing. This holds even more if the software has a high impact on safety of people and/or environment such as the control software on vehicles.

Automotive companies spend high efforts on testing software, often delivered by suppliers, on test benches with "hardware in the loop" [Boot et al. 99] describes automated testing techniques for HIL). Figure 1 shows an exemplary test bench. Large parts of the physical behavior of the vehicle are simulated, and the electronic control unit (ECU) is tested in this context. Today, the test suites for this process are generated by hand, based on the information contained in the requirement documents. This results in high development costs and does not provide a reliable assurance of the scope of testing.

In a project with a major German car manufacturer, we aim at addressing these issues by producing a tool that generates sets of tests automatically from the requirement specification. Since the software is to be tested in the context of the (simulated or real) vehicle, a coherent approach to testing software and physical parts in a uniform way is attractive. We are addressing this requirement by extending our model-based test generation algorithm [Struss 94] to software testing. A feasibility study for this approach was presented in [Esser-Struss 07].

The major challenge of the project is to build a tool that supports the existing work process, rather than requiring a major revision of the process, long education of the involved staff, etc. This implies, in particular,

- on the input side: we cannot expect the requirement engineers to learn and use a formal language for formulating requirements. The current requirement documents state mainly **functional** requirements and are mainly natural language texts.

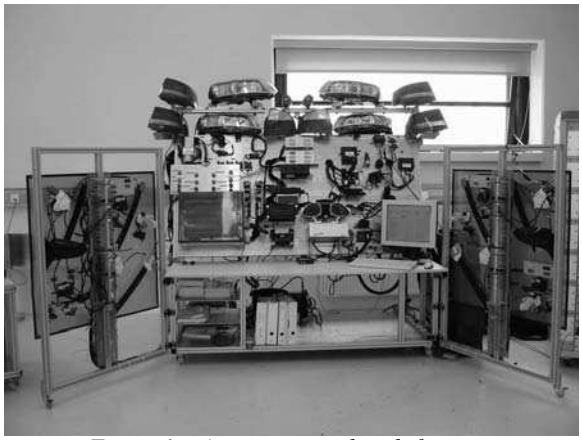


Figure 1 – A common test bench for cars.

- on the output side: the automatically generated tests need to be inspected, revised, and extended by the test engineers and, therefore, presented, justified, and explained in intuitive terms.

In this paper, we focus on the attempt to address these requirements. The proposed solution offers natural-language-template-based interface for acquiring software requirements (section 3). The content of the filled-in templates is automatically transformed into a representation in propositional logic and temporal relations and forms the model of the intended correct behavior (section 4). Models of potential faulty behaviors are generated from this OK model by a number of (types of) transformations (section 5). The fault types are defined mainly to match the intuition behind manually generated test cases and, hence, can deliver similar, but more systematic, test suites. This forms the basis for the intuitive justification of tests and their postprocessing (section 6).

In the following section, we give an overview of the approach and its elements.

2. The Approach

The perspective on testing is that confirmation of one behavior mode (OK) requires discriminating it from all possible faults. In the model-based test generation algorithm presented in [Struss 94], models (of physical systems) are represented as (finite) relations. Useful test inputs (the vertical axis in figure 2) are computed as the complement of those that may trigger the same observable response under both behavior modes (the horizontal axis). Such a test *guarantees* to refute at least one of the two beheviors.

The task of test generating for conformance testing is finding a set of test cases such that for each pair (m_{ok}, $m_{fail,i}$), where m_{ok} is the model of the correct system and $m_{fail,i}$ is one out of a given set $\{m_{fail,1}, \ldots, m_{fail,n}\}$ of models of system faults, a definitely discriminating test exists in the set.

Applying this approach to test generation of software requires a process containing at least three steps (see figure 3):

1st Step: building a model of correct behaviour, m_{ok}, In our case, the OK behavior is (solely) given by the (high-level, functional) requirements,

2nd Step: deriving fault models $\{m_{fail,1}, \ldots, m_{fail,n}\}$ from m_{ok} according to some selected fault classes,

3rd Step: computing the test cases for m_{ok} and $\{m_{fail,1}, \ldots, m_{fail,n}\}$.

The framework presented here can be seen as a refinement of these three steps under our objective, namely supporting the existing work process. Actually, it involves two different types of experts (and, in fact, different departments): requirement engineers and test engineers. The latter needs the results of the former as an input to his work and this exchange happens via documents, discussions, and phone calls and ist time-consuming, error prone, and only weakly supported by software. Our tool can be understood as providing support to both types of experts and a channel for the exchange of well-defined information between them.

In order to develop the foundation for a solution, we participated in both work processes, requirement acquisition and test generation, related to a new version of the Automatic Cruise Control System (ACC). This enabled us to develop the current working hypotheses for an appropriate representation of requirements and for identifying fault models that corresponds to the potential misbehaviors that the engineers hypothesize and their tests check for. This requires two more steps, namely

0th Step: obtaining a formal requirement specification from experts who are accustomed to using natural language for this purpose

4 th Step: presenting the generated tests and explaining the rationale behind them to the test engineers.

Figure 3 also indicates the specific requirements in the process which are:

A. The acquisition of requirements from the requirement engineer must be in a form that is close to their intuitions and the language they use, which is natural language.
B. The model (for m_{ok}) must be a formal one and
C. must preserve the structure of the requirement specification.
D. The model formalism used for the fault models must be expressive enough to cover all relevant requirements and fault types.
E. The purpose of the generated test cases must be comprehensible to the human expert.

Preserving of the structure means that the individual elements of the (manually created) requirement specification can still be identified in the model m_{ok}. The reasons for this, which are explained in detail later, are:

- to apply the same test criteria as in the current manual practice
- to explain the reason for a test case in terms of the requirement specification, which leads to intuitive justifications.

These aims are addressed in this framework by introducing two representations used in different steps in the process:

1. At the user interface: **Template Based Natural Language Specification** (TBNLS), where each requirement is a filled template forming a natural language sentence.

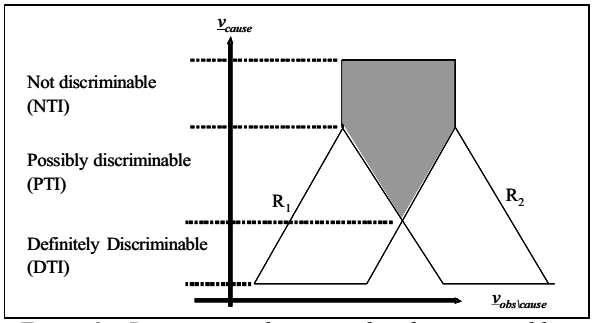


Figure 2 – Determining the inputs that do not, possibly and definitely discriminate between models R_1 and R_2.

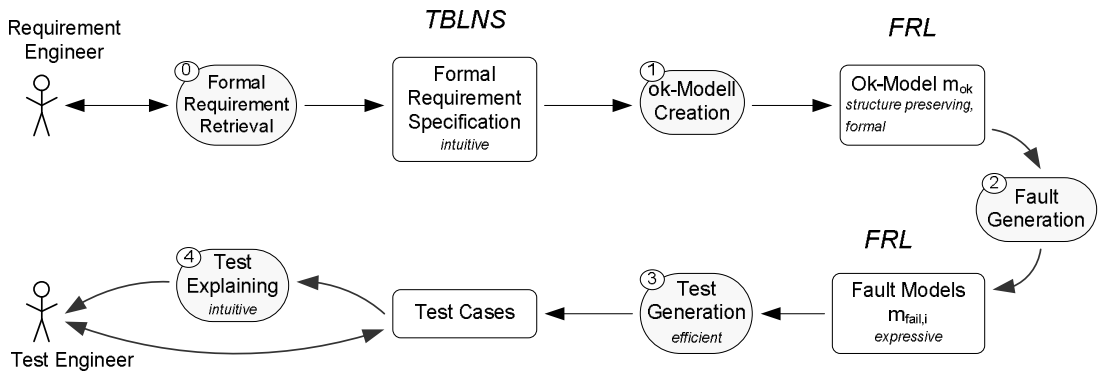


Figure 3 – the steps of fault-based test generation, and the formalisms used in this framework.

2. Internally, this is transformed into a **Formal Requirement Language** (FRL) where requirements are tuples (start-condition, consequence, end-condition). Conditions and consequence are specified in terms of propositions, that characterize system states, time intervals, and temporal constraints.

The human expert interacts only with TBNLS notation and only specifies the correct behavior. From this, the FRL specification is automatically generated and then used to generate fault models (see figure 3).
In the following sections the formalisms are described.

3. Natural-language-like Specification

In current practice, functional requirements (except for a small set of critical applications) are usually stated informally in natural language. A requirement acquisition tool has to allow the expert to stick to this as far as possible. Therefore, we decided to use a natural-language-like specification. However, since it must provide the basis for the following formal representations and algorithms, it has to

- have precise semantics and
- cover at least the most important/common classes of the requirements.

In order to achieve this, we studied current practice requirement documents supplied by the industrial partner. The left-hand column of figure 4 lists three typical examples from such a document. Our analysis showed that almost all requirements can be structured using three elements: *start-condition (if)*, *consequence (then)* and *end-condition (until)*. If a situation matches the start-condition, then also the consequence has to hold. Additionally, the termination of the consequence is specified by an *end-condition*. Thus, e.g. requirement R_2 can be reformulated as follows:

if the system is in mode m_1, lamp L3 is off, and button B4 is released,
then immediately lamp L3 is lit
until button B4 is down again or the system leaves m_1.

The end-condition may be missing, and, hence, the duration of the consequence is unspecified. If the start condition is also omitted, the consequence has to hold universally.
The following description of the template-based natural-language-like specification (TBNLS) comprises

- the set of templates for representing requirements, and
- a domain theory containing additional background information necessary for situating the requirements in a context and for generating tests that reflect and explaining this context.

Sentence Templates for Requirement Rules

A sentence template is a particular grammatical natural language pattern, which can be filled with state expressions and metrical time information. The right-hand column in figure 4 shows the filled in templates for the three sample requirements.
The template itself fixes the *temporal* relationships (e.g. P_1 must occur *before* P_2) between situations characterized by state expressions and classifies the state expressions as start-condition, consequence and end-conditions. The ability to specify exact temporal relationships is missing in many other natural language representations, such as ACE (Attempt to Controlled English, [Fuchs-Schwitter 96]) which is a subset of English restricted in vocabulary and grammar.
A state expression P characterizes a class of situations and is inductively defined from facts $F \in FACTS$:

$P := F \mid (P_1 \text{ AND } P_2) \mid (P_1 \text{ OR } P_2) \mid \text{NOT } P_1$

Facts are atomic propositions and do not have a structure or explicit semantics a priori; a reader must know what they mean. However, dependencies among facts may be defined in the domain theory. E.g. in requirement R_1 the facts are 'Button B_4 is not down', 'Button B_4 is down' and 'Lamp L_3 is lit'. '5 seconds' is metrical time information.
The choice to use unstructured propositions was made in order to avoid putting too much burden on the requirement engineer by forcing him to introduce certain predicates, variables with associated domains etc. While it is straightforward to assign a value '30km/h' to the quantity 'velocity', in other cases, it would be unnatural for requirements like 'Transition from mode m1 to m2 must be comfort-

able'. Later, we plan to explore the gain of a more structured representation of the facts. A more complex representation (which means more *work!*) may be more acceptable if the resulting benefit can be demonstrated and quantified.

In a template, 'A occurs' always means that A was false before and changed to true now, and 'A holds' means, that A is true independent of its value before or after.

The user can choose to activate a default rule, stating that a fact persists, unless specified otherwise in the template (which addresses the frame problem). However, this leads to disadvantages compared to specifying the behavior for entire time intervals explicitly: leaving the behavior unspecified for certain intervals, which may be adequate during design stage, is not possible.

Domain Theory

The functional requirements are considered as a description of the intended behavior of the respective subsystem. However, it is only a partial description and is likely to be insufficient for generating reasonable tests for several reasons. Here, we are not referring to the fact that the requirement engineer may have forgotten to specify some relevant aspects of the desired system behavior. The sources of incompleteness are more fundamental:

- The collection of requirements may **not** contain the most **basic** ones, because they are obvious to anybody involved in the process. (Actually, sometimes, they will only be assumed to be obvious, and e.g. the test engineer may not be aware of them, which creates a problem). For instance, the fundamental function of a brake, namely to decelerate the vehicle will probably not be subject to an explicit requirement.
- The collection of requirements concerns only **one subsystem**, and, more specifically, its software and does not specify the behavior of the context, i.e. the physical components of this subsystem and other subsystems it is interacting with. For instance, the Automatic Cruise Control (ACC) interacts with the braking system, whose function will produce in turn an impact on the ACC (by reducing the speed and potentially increasing the distance to the preceding vehicle).
- Also, the **environment** of the **vehicle** (road conditions, the driver's actions, other vehicles etc.) will usually not show up in the requirements, but may be relevant to test generation. For instance, acceleration of the preceding vehicle influences the distance to it, which is a measured input to the ACC. Also, basic physical constraints will be missing, such as the fact that the vehicle cannot brake and accelerate at the same time.

If this background knowledge is not available to the test generation algorithm, it may still be able to produce tests, but they may be unintuitive and overly complicated, and miss some "obvious" solutions. Complementing the model obtained from the requirements by this kind of knowledge will make automated test generation more powerful and provide results with higher acceptance. We expect that a very basic and qualitative model of vehicle functions and

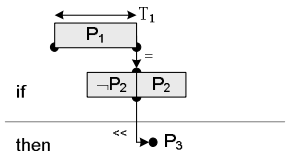
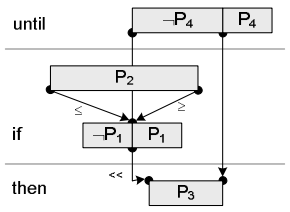
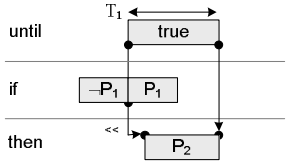
	Prosa Requirement	Requirement Sentence Template	In FRL $R=\left(A \overset{\text{then}}{\underset{\text{first}}{\rightarrow}} B \overset{\text{until}}{\rightarrow} C\right)$	Graphical
R_1	If button B4 has not been down during the last 5 seconds, then lamp L3 must be lit immediately after button B4 changed to down.	If Button B_4 is not down P_1 held the last 5 seconds T_1 and now Button B_4 is down P_2 occurs, then must Lamp L_3 is lit P_3 follow immediately.	$A=[P_1]_{t1}^{t2} \wedge [[P_2]_{t3}^{t4} \wedge (t_2 \mid t_3) \wedge (t_1 = t_2 - T_1)$ $B=[P_3]_{t5}^{t6} \wedge (t_3 \mid t_5)$	
R_2	Immediately after button B4 Is released while the system is in mode m_1 and lamp L3 is not lit, the lamp L3 must be lit until button B4 is down again or the system leaves m_1.	If Button B_4 is down P_1 occurs, during System is in mode m_1 AND Lamp L_3 is not lit P_2 holds, then Lamp L_3 is lit P_3 must hold immediately, until Button B_4 is up OR System is not in mode m_1 P_4 hold.	$A=[[P_1]_{t_1}^{t_2} \wedge [P_2]_{t_3}^{t_4} \wedge (t_3 < t_1 < t_4)$ $B=[P_3]_{t_5}^{t_6} \wedge (t_1 \mid t_5)$ $C=[P_4]_{t_7}^{t_8} \wedge (t_6 \mid t_7)$	
R_3	Immediately after button B4 Is pressed, Lamp L3 must be red for 10 seconds.	If Button B_4 is down P_1 occurs, then Lamp L_3 is red P_2 hold immediately, until 10 seconds T_1 elapsed.	$A=[[P_1]_{t_1}^{t_2}$ $B=[P_2]_{t_3}^{t_4} \wedge (t_2 \mid t_3)$ $C=(t_4 = t_3 + 10s)$	

Figure 4 – Examples of natural-language requirements and the respective templates, their notation in FRL, and graphical representation (from left).

its context will suffice to achieve this and, therefore, be highly reusable for different subsystems (and, in fact, for different work processes).
There is another limitation that is due to the chosen granularity of the current requirement representation:

- Since the entries in the templates are treated as elementary propositions, their ontological relations (e.g. exclusiveness or a taxonomy) are not made explicit in the set of requirements. In our example, 'Button is down' is the negation of 'Button is up'.

To exclude situations which are impossible in reality, such kind of dependencies have to be included in the domain theory. Otherwise, test generation, may produce test cases that cannot be executed.
The axioms of the domain theory should be expressed in the same way as the requirements (e.g. for defining that the 'Button is pushed' holds, if and only if 'Button is down' holds now and 'Button is up" was true before), but represented separately from them, because

- the requirement engineer is only interested in the requirements and should not be forced to deal with the domain theory (which is obvious background knowledge to him)
- the domain theory plays a different role in the process, in that its interdependencies are not subject to testing,.

At present, we use 15 different requirement sentence templates. They suffice to express the set of requirements in the current project. Of course, additional templates may be added if needed. Because it might be difficult to pick the right template from a larger set and because there may be a need to modify the template while formulating a requirement, in the future, we may consider supporting the construction of templates from elementary fragments (logical connectors, temporal constraints) similar to configuring functions in Excel. In this paper, we refer only to the templates underlying R_1, R_2 and R_3.

4. Formal Requirement Language

In order to build the model, the first step after acquiring the requirements in template form is to transform each template instance into a requirement in the formal language (FRL). Figure 4 shows the example rules in FRL notation (left-hand column) and their graphical illustrations.
The set of FRL requirement rules over a set of facts FACTS is defined inductively.

Definition – FRL Requirement
Let ε_{state}=P be the set of state expressions defined in section 3 and ~ be one of the relations =, <, >, ≤, ≥, and |. Then

$$\begin{aligned}
\varepsilon_{interval} &= \left[\varepsilon_{state}\right]_{t1}^{t2} \mid \left[\left[\varepsilon_{state}\right]_{t1}^{t2} \mid \left[\varepsilon_{state}\right]\right]_{t1}^{t2} \\
&\mid \varepsilon_{interval} \wedge \varepsilon_{interval} \mid \varepsilon_{interval} \vee \varepsilon_{interval} \\
&\mid (\varepsilon_{interval}) \mid \varepsilon_{interval} \wedge \varepsilon_{quant} \mid \neg\varepsilon_{interval} \\
\varepsilon_{quant} &= \exists\, \varepsilon_{unquant} \mid \neg\exists\, \varepsilon_{unquant} \\
&\mid \varepsilon_{quant} \wedge \varepsilon_{quant} \mid \varepsilon_{quant} \vee \varepsilon_{quant} \\
\varepsilon_{unquant} &= (\varepsilon_{interval} \wedge \varepsilon_{constr}) \mid \neg\varepsilon_{unquant} \\
\varepsilon_{constr} &= (t_i \sim t_j) \mid (t_i \sim t_j + x) \mid (t_i \sim t_j - x) \\
&\mid \varepsilon_{constr} \wedge \varepsilon_{constr} \mid \varepsilon_{constr} \vee \varepsilon_{constr} \\
\varepsilon_{requ} &= \left(A \overset{\text{then}}{\rightarrow} \left[\varepsilon_{state}\right]_{ta}^{tb} \wedge \varepsilon_{constr} \overset{\text{until}}{\underset{\text{first}}{\rightarrow}} C \right) \\
&\mid \left(A \overset{\text{then}}{\rightarrow} \left[\varepsilon_{state}\right]_{ta}^{tb} \wedge \varepsilon_{constr} \right)
\end{aligned}$$

where A,C ∈ $\varepsilon_{unquant}$.
The informal semantics is:

- $\left[\varepsilon_{state}\right]_{t1}^{t2}$ specifies that ε_{state} holds all the time during the (closed) interval $[t_1,t_2]$, where ε_{state} may but not need to hold before and after that interval.
- $\left[\left[\varepsilon_{state}\right]_{t1}^{t2}$ specifies a left-max interval where ε_{state} must hold, i.e. additional to the above expression, ε_{state} must not hold in the interval right before t_1. The analogue holds for $\left[\varepsilon_{state}\right]\right]_{t1}^{t2}$.
- $t_1 \mid t_2$ means t_2 follows t_1, i.e. there is an infinitely short time between t_1 and t_2.

The formal semantics is given in predicate logic, where F(t) means, that fact F holds at time t:

$$\left[\varepsilon_{state}\right]_{t1}^{t2} := \forall_{t,t1\le t\le t2} \varepsilon_{state}(t)$$

$$\left[\left[\varepsilon_{state}\right]_{t1}^{t2} := \exists_{t',t0} \left[\neg\varepsilon_{state}\right]_{t'}^{t0} \wedge \left[\varepsilon_{state}\right]_{t1}^{t2} \wedge (t_0 \mid t_1)$$

$$t_1 \mid t_2 := \forall_{t0<t1,t3>t2} \left(\left[t_0;t_1\right] \cup \left[t_2;t_3\right] = \left[t_0;t_3\right]\right)$$

where [a;b] is the closed interval between a and b. Figure 5 shows the semantics of a requirement in predicate logic. $\overline{C}$ is a copy of C, where each variable name v in v(C), except those occurring also in formula A, is renamed into $\overline{v}$. The same holds for $\overline{B}$.

Preservation of Structural Information

Requirements are constraints on the intended behavior of the system. While in verification one would simply check whether or not the set of constraints are fulfilled, test generation and test justification (section 6) requires the preservation of some structural information. From a purely logical perspective, one could transform an FRL requirement into an implication and, thus, into a single constraint and a set of requirements into a constraint network. While this can be done to specify the semantics of
$\overset{\text{then}}{\rightarrow}$ and $\overset{\text{until}}{\underset{\text{first}}{\rightarrow}}$,
FRL maintains

- the individual requirements as units and
- within a single requirement, the distinction between start-condition, consequence and end-condition.

This is essential because the tests have to be generated for each single requirement, and conditions and consequence play a different role in test generation. While the former (and their mutations) need to be **established** by other **actions**, the latter is what must be established by the system, can be affected by faults, and, hence, needs to be **checked**. Section 5 will show how this structure is exploited for generating fault models.

$$R = \left(A \xrightarrow{\text{then}} B \xrightarrow[\text{first}]{\text{until}} C \right) := \forall_{\text{fv}(A)} \left(A \Rightarrow BC_1 \vee BC_2 \right),$$

$$R = \left(A \xrightarrow{\text{then}} B \right) := \forall_{\text{fv}(A)} \left(A \Rightarrow C \right)$$

with

$$BC_1 \Leftrightarrow \exists_{ta,tb} \left(B \wedge \left(\exists_{\text{fv}(C)} C \right) \wedge \left(\neg \exists_{\text{v}(\overline{C}),\overline{ta},\overline{tb}<tb} \left(\overline{C} \wedge c_{\overline{B}} \right) \right) \right)$$

$$BC_2 \Leftrightarrow \left(\exists_{t_a,t_b} \left(B \wedge \left(t_b = \infty \right) \right) \wedge \forall_{t_a,t_b} \neg \exists_{fv(C)} \left(C \wedge c_B \right) \right)$$

$$B := \left[P \right]_{t_a}^{t_b} \wedge c_B$$

Figure 5 – Semantic of a FRL requirement rule in predicate logic.

Other possible representations, such as finite state machines representing the whole functionality of the system or formulae of the Duration Calculus (DC, see [Chaochen-Hansen 03]) do not contain information about start-, end-condition or consequence of the original requirements.
Requirements in FRL do not only contain this structural information, but also provide a 1:1 mapping between each state expression in the template and in the formal requirement. This is relevant for the tool, because it forms the basis for presenting comprehensible justifications for the generated tests to the test engineer, which can be stated in terms of the elements of the original requirements (formulated in a template).

Expressiveness of FRL

The formal requirement language is quite expressive, and, in fact, it is overly expressive from the application point of view. The included continuous time model results in undecidability. For instance, consider the simple rule: '5 seconds after pressing a button, the lamp must be lit'. The button could be pressed infinitely often during 5 seconds, and each time the requirement would have to hold. Thus during this period, an infinite amount of memory would be necessary to keep track of the individual time points the lamp must be lit. Of course, this is irrelevant under practical consideration, because the button may be pushed often, but not infinitely often.
There are two ways to cope with this issue. One could restrict the language explicitly to a discrete time model with finite granularity. Alternatively, one can restrict the use of the language. This is what is currently guaranteed by the finite sets of templates (and requirements), the way fault models are constructed, and the fact that test generation considers a finite set of steps only.

5. Fault Model Types

We also analyzed how test cases of an existing test suite were generated and identified the motivation behind them. It is not surprising that tests are explicitly or (often) implicitly based on hypothesizing "what may go wrong" and designed to detect a certain type of misbehavior (deviation from the specification). It turned out that many of these fault types can be represented as defects in the requirement specification and can be described in terms of start-condition, end-condition and consequence that are "mutations" of the ones occurring in the requirements.
In the following, we discuss some fault types that were used most frequently in the analyzed documents. We present examples and the fault types informally and in FRL.
Two of the most obvious fault types are, stated intuitively,

- The conditions are satisfied, but the consequence does not occur.
- The condition is not satisfied, but the consequence occurs, anyway. (At a second glance, this is not necessarily a fault, unless there are other requirements contradicting this).

To illustrate the first case, consider the example requirement R: "if button B3 is down, lamp L1 is lit and the speed is below 130km/h, lamp L2 must be on for 5 seconds".
Then the first fault type states that in any situation matching the start-condition (*positive* case), the lamp L2 is not on for 5 seconds (but may be on for less than 5 seconds).

Simple Positive Fault

Contrary to the specification, in all conditions that satisfy the start condition of a requirement, its consequence does not occur, i.e. the negated consequence occurs

For a requirement R_i one fault model $m_{\text{fail,i}}$ is created which differs from the correct specification m_{ok} only in R_i by replacing B with its negation:

$$m_{\text{fail},i} = m_{\text{ok}} \left[R_i \mapsto R_{i,\text{fail}} \right] \wedge$$

$$R_{i,\text{fail}} = R_i \left[\left(B \xrightarrow{\text{until}} C \right) \mapsto \neg \left(B \xrightarrow{\text{until}} C \right) \right].$$

where $X[Y \mapsto Z]$ means that Y is replaced by Z within X.

Here, and in the following, the respective requirements are stated as

$$R_i = \left(A \xrightarrow{\text{then}} B \xrightarrow[\text{first}]{\text{until}} C \right)$$

For the next fault type assume that it is known (via other requirements) that if even only one of the conjuncts in the start-condition of R is not true (*negative* case), e.g. the button is not down but L1 is off and speed is below 130km/h, then the lamp L2 is not on for 5 seconds. In contrast the fault states that L2 *is lit* in such negative cases.

Simple Negative Fault

Contrary to the specification, in all conditions that differ from the start-condition of a requirement by exactly one negated conjunct, the requirements consequence does occur, although other requirements R_2, … R_n implies a different consequence.

For each fact occurrence FO_j in start-condition A of a requirement R_i, a fault model $m_{\text{fail,i,j}}$ is created which differs from the ok model by removing FO_j from A:

$$m_{\text{fail},i,j} = m_{\text{ok}}\left[R_i \mapsto R_{i,\text{fail},j}\right] \wedge$$
$$R_{i,\text{fail},j} = R_i\left[A \mapsto A_{\text{fail},j}\right] \wedge$$
$$A_{\text{fail},j} = A\left[FO_j \mapsto \text{true}\right].$$

The next fault type leads to a boundary value analysis. The fault states that in any borderline situation, like the speed is 129,5km/h, the consequence does not occur.

We therefore define that a value assignment v='(v_1,…v_n) for all variables V=(V_1,…V_n) is called a positive borderline situation of an expression expr, iff the expression is true under v and there exists a i≤n such that expr becomes false when in- or decreasing (only) the value v_i.

Boundary Value Positive Fault

Contrary to the specification, for each positive borderline situation of the start-condition, the consequence does not occur.

Analogously, a negative borderline can be defined, where the expression is false but becomes true with an in-/decrease, leading to the definition of a Boundary Value Negative Test.

Representing this fault type is fairly clumsy using propositions only. It will be easier when we introduce value assignments to variables as state expressions..

Although the simple positive fault type appears very simple, there can be many state expressions that satisfy the respective condition (conjunctions that subsume the condition), and only some of them are reasonable to consider, because additional conjuncts interact with the other ones. This interaction can be due to shared resources. E.g. assume that both the lamp L1 and the window lifter consume a significant amount of power from the battery. A faulty behavior because of resource shortage may occur if both, 'L1 is lit' and 'window lifter is on' hold at the same time.

Resource Shortage Fault

The starting conditions of two requirements are satisfied such that both consequences must hold during the same time, where both shares a common resource, one of the consequences does not occur.

For each tuple (F_1,F_2)∈FACTS^2 where both facts share the same resource, *share-resource(F_1, F_2)*, two fault models $m_{\text{fail},1}$ and $m_{\text{fail}2}$ are created. The first differs from m_{ok} in replacing each requirement R_i with consequence

$$B = \left(\left[F_1\right]_{ta}^{tb} \wedge c_B\right)$$

by $R_{i,\text{fault}}$ with:

$$R_{i,\text{fault}} = R_i\left[B \rightarrow B'\right],$$
$$B' = \left(\left[\left(F_1 \wedge \neg F_2\right) \vee \left(\neg F_1 \wedge F_2\right)\right]_{ta}^{tb} \wedge c_B\right).$$

Replacing each F_1 in the formulas above by F_2 and vice versa leads to the second model $m_{\text{fail},2}$.

Note that all fault types above refer to the structure of the requirements stated as templates.

There are more plausible fault types, and we list some of them:

Unwanted Temporal Ordering Dependency

Contrary to the specification, if the events of the start condition occur in a specific order, the consequence of a requirement does not occur.

Example: The specification contains the requirement „When buttons A and B are both down for 5 seconds, then X must occur immediately" According to this requirement the consequence must occur independently of the order of pressing A and B.

Wrong Requirement Priority

Contrary to the specification, a requirement has a lower priority than another requirement (if a requirement has a higher priority than another, it over-writes it, i.e. in a condition matching both start conditions, only the consequence of the requirement with the higher priority if they are contradictory).

Example: assuming the following two requirements, where the first is higher prioritized than the second. Requirement 1: "If the main switch is turned to position ‚off', the system must immediately turn off itself". Requirement 2: "if the pedal is released in mode $active_1$, the system must immediately switch to mode $active_2$". Here a Wrong Requirement Priority Fault exists, if the system switches to $active_2$ instead of deactivating itself, if the pedal and the switch are pressed simultaneously in $active_1$.

Requirement Violated in Temporary States

In a specific temporary state the consequence of a requirement does not hold.

Example: assume in some situations mode m_1 is active for 500ms only and then becomes inactive without further interaction. The fault type states that the consequence of some selected or all applicable requirements do not hold during m_1 although the start-conditions are fulfilled. The underlying hypothesis is that such short states may be easily overlooked during testing.

Incorrect Bracketing

Wrong brackets in a state expression

For instance, the text "…button A or button B and button C is pressed…" in a requirement document could be interpreted as (A∨B)∧C is pressed or A∨(B∧C) is pressed. This fault type assumes that the wrong interpretation was chosen for implementation.

Note that for some fault types additional knowledge not contained in the requirement specification is needed, e.g. for an Unwanted Resource Shortage Fault, information about dependencies between facts (or requirements) and the resource are needed.

Note that a faulty requirement may contradict other requirements, thus it has to override them in order to get a faulty but consistent specification (e.g. by assigning each requirement a priority).

In summary, we tried to illustrate in this section that

- there exist intuitive concepts of fault types that motivate tests,
- these fault types are obtained as transformations of the respective requirements (and that this re-

quires the preservation of the structure of the requirements)
- they can be stated in the same language as the requirements, FRL.

6. Input and Output of Test Generation

Based on the described foundations, a test generation specification TGS contains all information, besides the specification $model_{ok}$ of the system-under-test, needed to successfully perform fault-based test generation. A TGS is represented as a tuple:

$TGS = (TESTBENCH, SET_{TESTIDEAS})$,

where

$TESTBENCH = (FACTS_{obs}, FACTS_{causal})$

describes the attributes of the test bench, which is the technical interface to the system-under-test. When switching to another test bench, TESTBENCH must be adopted properly. It declares the set $FACTS_{obs}$ of facts observable and the set $FACTS_{causal}$ of facts that can be manipulated by the tester. Optionally the test bench's maximal temporal resolution of the observations, Δt_{obs}, and of the stimuli, Δt_{obs}, may be stated. For instance, Δt_{obs} =10ms stating that events lasting less than ten milliseconds are not visible, becomes handy if the stimuli is observed only by a human tester.

The set $SET_{TESTIDEAS}$ consists of test ideas

$TESTIDEA = (TYPE_{fault}, REQUIREMENT)$.

A test idea specifies that the fault type $TYPE_{fault}$ has to be applied to REQUIREMENT, which results in one or several fault models. Fault specific parameters, such as a reference to resource sharing for Resource Shortage Faults, may be given in addition.

Since the test generation algorithm produces tests in order to discriminate the OK behavior from the various fault models, the purpose of the test can be explained to the test engineer by referring to a specific requirement and to certain fault types, i.e. at a conceptual level he is familiar with. One could also try to display the hypothesized fault types in terms of templates. Because there is no guarantee that the FRL fault model corresponds to any of the templates offered for requirement acquisition, this would only work if such templates are generated automatically from FRL.

7. Future Work

The project described here is driven by the application requirements. Therefore, we have to avoid overloading the acquisition of the requirements and the domain theory. This is why we have to allow a rather coarse level for expressing FACTS, with the obvious drawback that at this propositional level many of the interrelations of various requirements, including inconsistency, refinement, redundancy, remain implicit and cannot be exploited by the algorithm, which weakens its results. Such interrelations can be made explicit in the domain theory, causing higher efforts on this side. The alternative is to allow for a more structured representation. An obvious extension is to introduce variables and assignment of values or ranges to them. Currently, we are performing an initial evaluation, which will provide us with feedback from the test engineers and with hints on an appropriate trade-off between limiting efforts on the requirement acquisition side and the quality and utility of the generated tests. At least, we will be able to demonstrate what can be gained by a more structured and systematic requirement acquisition.

Acknowledgements

Thanks to the Model-based Systems and Qualitative Modeling Group at the Technical University of Munich, Oskar Dressler, Martin Sachenbacher, and the reviewers for their helpful comments. We also thank Audi AG, Ingolstadt, and, in particular, Reinhard Schieber for support of this work.

References

[Boot et al. 99] Boot, R., Richert, J., Schutte H. and Rukgauer, A. 1999, *Automated test of ECUs in a hardware-in-the-loop simulation environment*. Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design.

[Chaochen-Hansen 03] Chaochen, Z., and Hansen, M. R. 2003, *Duration Calculus. A Formal Approach to Real-Time Systems*. Springer.

[Esser-Struss 07] Esser, M. and Struss, P. 2007, *Fault-model-based Test Generation for Embedded Software*, IJCAI2007, Hyderabad, India.

[Fuchs-Schwitter 96] Fuchs, N. E., Schwitter, R. 1996, *Attemp to Controlled English (ACE)*, CLAW 96, First International Workshop on Controlled Language Applications, University of Leuven, Belgium

[Struss 94] Struss, P. 1994, *Testing Physical Systems.* In Proceedings of AAAI-94, Seattle, USA.