In: International Joint Conference on Artificial Intelligence (IJCAI). 6th January through 12th January 2007,
Hyderabad, India

Fault-model-based Test Generation for Embedded Software

M. Esser[1], P.Struss[1,2]
[1]Technische Universität München, Boltzmannstr. 3 D-85748 Garching, Germany
[2] OCC'M Software, Gleissentalstr. 22, D-82041 Deisenhofen, Germany
{esser, struss}@ in.tum.de, struss@ occm.de

Abstract

Testing embedded software systems on the control units of vehicles is a safety-relevant task, and developing the test suites for performing the tests on test benches is time-consuming. We present the foundations and results of a case study to automate the generation of tests for control software of vehicle control units based on a specification of requirements in terms of finite state machines. This case study builds upon our previous work on generation of tests for physical systems based on relational behavior models. In order to apply the respective algorithms, the finite state machine representation is transformed into a relational model. We present the transformation, the application of the test generation algorithm to a real example, and discuss the results and some specific challenges regarding software testing.

1 Introduction

Over the last decade, cars have become a kind of mobile software platform. There are tens of processors (Electronic Control Units, ECU) on board of a vehicle; they are communicating with each other via several bus systems, and software has a major influence on the performance and safety of a vehicle. The software embedded in the car subsystems becomes increasingly complex, and it comes in many variants, reflecting the context of different types of vehicles, the manufacturer-specific physical realization, versions over time etc. Testing such embedded software becomes increasingly challenging and has been moving away from test drives under various conditions to automated tests performed on test benches which can partly or totally simulated the car as a physical system.

But for the reasons stated above, namely complexity of the software and its variation, generating the test suites becomes demanding and time consuming and demands for computer support. Automating the generation of such tests based on a specification of the desired behavior of the software together with the physical system promises benefits regarding both the required efforts and the completeness of the result.

In [Struss 94], we presented the theoretical and technical foundations for automated test generation for physical systems based on (relational) models of their (nominal and faulty) behavior.

An extension of this approach to cover also software would be highly beneficial, because it would provide a coherent solution to testing both physical systems and their embedded software. More concretely, the software test could start from a specification of the intended behavior of the entire system (including physical components and software), and also the tests could reflect the particular nature of the **embedded** software, namely using stimuli and observations of the physical system rather than directly of the software system.

The case study described in this paper concerns a real-life example (the measurement and computation of the fuel level in a vehicle tank) based on the requirement specification document of a car manufacturer.

We continue by summarizing the basis for our relation-based implementation of test generation. In order to extend it to software, the requirement specification has to be turned into a relational representation. In the respective document, the skeleton of this specification is provided in a state-chart manner. Therefore, section 3 of this paper proposes a behavior specification as a special finite state machine, and section 4 presents the transformation into a relational representation.

A major challenge in the application of the test generation algorithm to software is to provide relevant and appropriate fault models against which the software should be tested (section 5). The final sections present results of the case study and discuss problems and insights.

2 The Background: Model-based Test Generation

The goal of this work is not to develop a new test generation algorithm, but apply it to (devices with embedded) software systems. Therefore, we only briefly summarize the theoretical and technical foundations and refer to [Struss 94, 07] for more details.
Testing attempts to influence a system in a way that reveals information for discriminating between different hypotheses about the system (e.g. about the kind of fault that is present).

Definition (Discriminating Test Input)

Let $TI = \{ti\}$ be the set of possible test inputs (stimuli), $OBS = \{obs\}$ the set of possible observations (system responses), and $H = \{h_i\}$ a set of hypotheses.

$ti \in TI$ is called a **definitely** discriminating test input for H if

(i) $\forall\ h_i \in H \quad \exists\ obs \in OBS \qquad ti \wedge h_i \wedge obs$ O $\bot$, and
(ii) $\forall\ h_i \in H \quad \forall\ obs \in OBS$
if $\quad ti \wedge h_i \wedge obs$ O $\bot$
then $\forall\ h_j \neq h_i\ ti \wedge h_j \wedge obs$ P $\bot$.

ti is a **possibly** discriminating test input if
(ii′) $\forall\ h_i \in H \quad \exists\ obs \in OBS$ such that
$ti \wedge h_i \wedge obs$ O $\bot$ and $\quad \forall\ h_j \neq h_i \quad ti \wedge h_j \wedge obs$ O $\bot$.

Testing for confirming (or refuting) a particular hypothesis h_0 out of the set H requires only discrimination between h_0 and any other hypothesis.

Definition (Confirming Test Input Set)

$\{ti_k\} = \quad TI' \subset TI$ is called a discriminating test input set for $H = \{h_i\}$ and $h_0 \in H$ if
$\forall\ h_j$ with $h_0 \neq h_j \qquad \exists\ ti_k \in TI'$ such that
ti_k is a (definitely or possibly) discriminating test input for $\{h_0, h_j\}$.

It is called **definitely confirming** if all ti_k are definitely discriminating, and **possibly confirming** otherwise. It is called **minimal** if it has no proper subset $TI'' \subset TI'$ which is discriminating.

Remark

Refutation of all hypotheses $h_j \neq h_0$ implies h_0 only, if we assume that the set H is complete, i.e. $\vee_{\mathrm{i}} \quad h_i$

[Struss 94] treats test generation for physical systems, with hypotheses concerning their (correct or possible faulty) behavior, which is assumed to be characterized by a vector
$\underline{v}_S = (v_1, v_2, v_3, \ldots, v_n)$ of system variables with domains
$DOM(\underline{v}_S) = \quad DOM(v_1) \times DOM(v_2) \times \ldots \times DOM(v_n)$.
Then a hypothesis $h_i \in H$ is given as a relation
$R_i \subseteq DOM(\underline{v}_S)$.
For conformity testing, h_0 is given by $R_0 = R_{OK}$, the model of correct behavior. Observations are value assignments to a subvector of the variables, $\underline{v}_{obs}$, and also the stimuli are described by assigning values to a vector $\underline{v}_{cause}$ of susceptible ("causal" or input) variables. We make the reasonable assumption that we always know the applied stimulus which means the causal variables are a subvector of the observable ones: $\underline{v}_{cause} \subseteq \underline{v}_{obs}$.

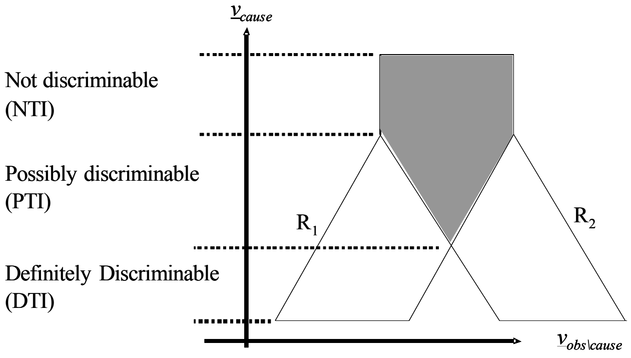


Figure 1 Determining the inputs that do not, possibly and definitely discriminate between R_1 and R_2

The basic idea underlying model-based test generation ([Struss 94]) is that the construction of test inputs is done by computing them from the observable differences of the relations that represent the various hypotheses. **Figure 1** illustrates this. Firstly, for testing, only the observables matter. Accordingly, Fig. 1 presents only the projections, $p_{obs}(R_i)$, $p_{obs}(R_j)$, of two relations, R_1 and R_2, (possibly defined over a large set of variables) to the observable variables. The vertical axis represents the causal variables, whereas the horizontal axis shows the other observable variables (representing the observable system response).

To construct a (definitely) discriminating test input, we have to avoid stimuli that can lead to the same observable system response for both relations (the shaded region in Fig. 1). This provides the intuition behind

Lemma 1

If $h_i=R_i$, $h_j=R_j$, $TI=DOM(\underline{v}_{cause})$, and $OBS=DOM(\underline{v}_{obs})$, then
$DTI_{ij} = DOM(\underline{v}_{cause}) \setminus p_{cause}(p_{obs}(R_i) \cap p_{obs}(R_j))$
is the set of definitely discriminating test inputs for $\{h_i, h_j\}$.

Please, note that we assume that the projections of R_i and R_j cover the entire domain of the causal variables which corresponds to condition (i) in the definition of the test input.

The sets DTI_{ij} for all pairs $\{h_i, h_j\}$ provide the space for constructing (minimal) discriminating test input sets.

Lemma 2

The (minimal) hitting sets of the set $\{DTI_{0j}\}$ are the (minimal) definitely confirming test input sets for H, h_0.

A hitting set of a set of sets $\{A_i\}$ is defined by having a non-empty intersection with each A_i. (Please, note that Lemma 2 has only the purpose to characterize all discriminating test input sets. Since we need **only one** test input to perform the test, we are not bothered by the complexity of computing all hitting sets.)

This way, the number of tests constructed can be less than the number of hypotheses different from h_0. If the tests have a fixed cost associated, then the cheapest test set can be found among the minimal sets.

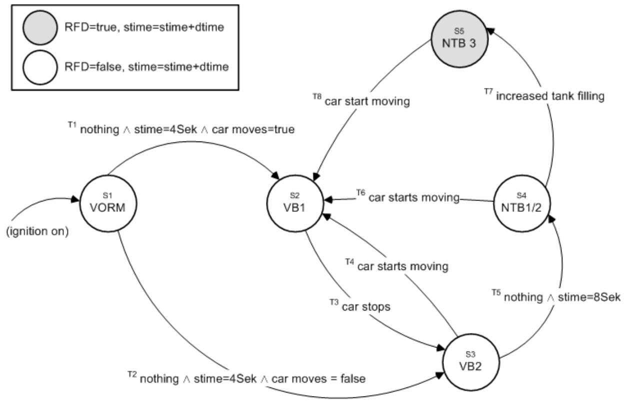

RFD=true, stime=stime+dtime
RFD=false, stime=stime+dtime
S5
NTB 3
T8 car start moving
T7 increased tank filling
T1 nothing ∧ stime=4Sek ∧ car moves=true
S1
VORM
S2
VB1
T6 car starts moving
S4
NTB1/2
(ignition on)
T4 car starts moving
T3 car stops
T5 nothing ∧ stime=8Sek
S3
VB2
T2 nothing ∧ stime=4Sek ∧ car moves = false

Figure 2: FSM describing a refilling detection in a personal car

3 State Charts for Specification of Software Requirements

Extending the solution sketched above to software testing raises some fundamental issues. Firstly, it assumes a behavior description in terms of relations. (Embedded) software, however, is usually specified and described in terms of its (discrete) dynamic features. Secondly, and even more fundamentally, in this case testing is not concerned with faults in a physical device, but **bugs in the design** of an artifact. In the work described here, we address the second problem by

- starting from the specification and model faults as (classes of) **deviations from this specification**, and the first one by
- **transforming** a widely used representation of such specifications into a relational representation.

State charts and finite state machines (FSM) are frequently used in specifications of software requirements. **Figure 2** shows a FSM extracted from a requirement specification produced by an automotive manufacturer. The machine describes a process to detect refueling of a passenger car: if the car stops for more than 8 seconds and if a remarkably higher tank filling is detected then the software sets the output flag RFD (ReFilling Detected) to true. Otherwise RFD is always false.

Let us define the used type of FSM in a more formal way: an automata $m^a = (E,(I,O,L),(S,A),T,s^0,l^0)$ is described by

- the set E of events $e_1, \ldots, e_{nE}$,
- the ordered set I of input variables $i_1, \ldots ,i_{nI}$,
- the ordered set O of output variables $o_1, \ldots, o_{nO}$,
- the ordered set L of local variables $l_1, \ldots, l_{nL}$,
- the set S of control states $s_1, \ldots, s_{nS}$,
- the set A of state expressions $a_1, \ldots, a_{nS}$ defining a relation $\delta_{a,i} \subset dom(I) \times dom(L) \times dom(O) \times dom(L)$ for each state s_i,
- the set T of transitions $T_1, \ldots, T_{nT}$ with $T_i \subset S \times P(dom(E) \times dom(I) \times dom(L)) \times S$ where $P(X)$ denotes to the power set of X,
- the initial control state s^0 and
- the vector l^0 with the initial values of L.

Each machine has a special local variable l_1 called *stime* indicating the time elapsed since the machine has entered the actual control state. It is special because each time the control state is switched, the variable is reset automatically. Every variable v in I, O or L has a finite domain $dom(v)$.

With the inputs $(i^1, \ldots, i^n)$ and the events $(e^1, \ldots, e^n)$ the machine produces the outputs $(o^1, \ldots, o^n)$ according to the following operating sequence:

1. Set t = 0.
2. Evaluate the state expression a_i of the current state $s^t=s_i$ to calculate the new values of the output and local variables : $(i^{t+1}, l^t, o^{t+1}, l^{t+1}) \in \delta_{a,i}$
3. If T contains a transition $T_i=(s_{src}, IF, s_{dst})$ with $s_{src}=s^t$ and $(e^{t+1},i^{t+1},l^{t+1}) \in IF$ then set $s^{t+1}=s_{dst}$, otherwise set $s^{t+1}=s^t$.
4. If $s^{t+1} \neq s^t$ then reset *stime*.
5. Set $t=t+1$
1. Jump to Step 2.

In our example, the FSM has two input variables *car moves* and Δ*time*, one output variable *RFD*, *stime* as a the only local variable and the events *nothing*, *car starts moving, car stops* and *increased tank filling*. The variable Δ*time* is set according to the time elapsed since the previous event occurred. Its value is always added to the *stime* variable, which could be used in a precondition of a transition.

Dependent on the chosen set of input variables I and the events E, the test generation system needs more information in order to produce meaningful tests, because the values of some variables might depend on the occurrence of an event. E.g. if *car moves*^{t}*=true* then the event *car starts moving* can not occur next. In our example, the following rules are necessary:

$$\text{car moves}^t = \text{false} \quad \wedge \quad \text{car moves}^{t+1} = \text{true} \Leftrightarrow e^t = \text{car starts moving}$$

$$\text{car moves}^t=\text{true} \wedge \text{car moves}^{t+1}=\text{false} \Leftrightarrow e^t=\text{car stops}$$

In the next section, we describe how the FSM is transformed into a relational representation.

4 Transformation of a FSM into a Compositional, Relational Representation

The conversion of a FSM of the described type produces a compositional model, i.e. a model that preserves the structure and the elements of the FSM. As a consequence, a modification of one part of the FSM results in the modification of only one part of the compositional model (As it will turn out this is not fully accomplished for fundamental reasons). The compositional model also provides the possibility of relating

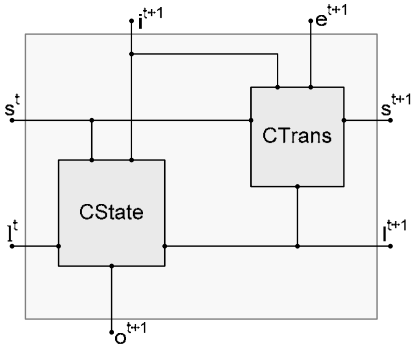


Figure 3: C1Step and its internal structure

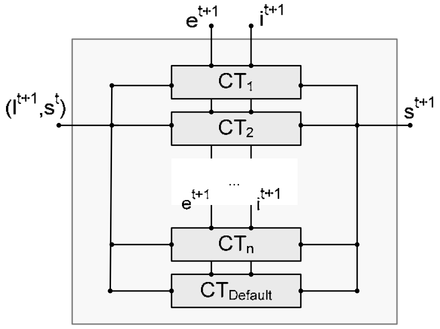


Figure 5: CTrans and its internal structure

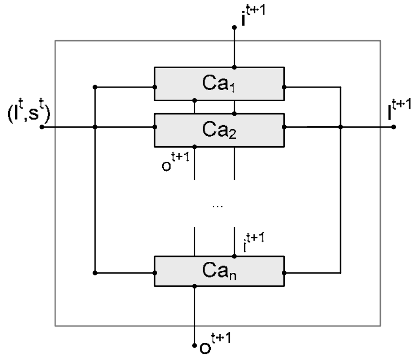


Figure 4: CState and its internal

"defects" to the various elements (and also to record and trace their effects e.g. in diagnosis).

The basic step is the transformation of the entire FSM into a component C1Step and its internal structure (Figure 3). C1Step takes the state s^t, values of local variables l^t, the input vector i^{t+1}, and the event e^{t+1} and generates the subsequent state s^{t+1}, the new values of local variables l^{t+1}, and the output

$$R_{Ca_j} = \left\{ \begin{array}{r} \left(\left(s^t, i^{t+1}, l^t\right), \left(l^{t+1}, o^{t+1}\right)\right) | \left(s^t = s_j \wedge (i^{t+1}, l^t, o^{t+1}, l^{t+1}) \in \delta_{a_j}\right) \vee \\ \left(s^t \neq s_j \wedge l^{t+1} \in L \wedge o^{t+1} \in O\right) \end{array} \right\}$$

vector o^{t+1}, reproducing the calculations of the FSM in one step (one iteration in the listed operation sequence). C1Step consists of the two components CState and CTrans. The former encodes the state expressions $\delta_{a,i}$, while CTrans represents the transitions T_i.

CState constrains s^t, l^t, l^{t+1}, o^{t+1} and i^{t+1} independently from the next event e^{t+1}. It contains nS atomic components Ca_i, one for each state expression a_i, which are placed in parallel (Figure 4). The expressions are conditioned by their respective state and, hence, exactly one component Ca_i defines the proper values of the variables. Hence, a change in one a_i results in the modification of only one component and a maximum of locality is achieved.

Ca_i determines l^{t+1} and o^{t+1} depending on s^t, l^t and i^{t+1} according to a_i. The relational model R_{Cai} of such an atomic component is:

CTrans correlates all the variables except the output o^{t+1} and consists of nT parallel atomic components CT_i, one for each transition, and a component $CT_{Default}$ (Figure 5). Exactly one component CT_i determines s^{t+1} depending on s^t, l^{t+1}, i^{t+1} and e^{t+1} according to T_i. The relational model R_{CTi} of these atomic components are:

$$\begin{array}{l} R_{CT_i} : \left\{\left(\left(s^t, e^{t+1}, i^{t+1}, l^t\right), s^{t+1}\right) \;\; | \right. \\ \qquad \exists_{T_i \in T} \;\left(T_i = \left(s^t, IF, s^{t+1}\right) \wedge \left((e^{t+1}, i^{t+1}, l^t) \in IF\right)\right) \vee \\ \qquad\qquad \left.\left(T_i = \left(\tilde{s}^t, IF, \tilde{s}^{t+1}\right) \wedge \left(s^t \neq \tilde{s}^t \vee (e^{t+1}, i^{t+1}, l^t) \notin IF\right) \Rightarrow s^{t+1} \in S\right)\right\} \end{array}$$

In all cases where no transition is executed, the atomic component $CT_{Default}$ defines the values according to the automata definition: the state does not change, $s^{t+1} = s^t$. Therefore its relation is

Now one iteration of the operating sequence can be simulated. To simulate n iterations, C1Step is copied n times and placed in series. But this shows also a limitation: the model can simulate only a fixed number of steps, and the more C1Step components are interconnected the bigger the model (the relation of the entire model) grows.

The number of steps needed for test generation depends on the respective FSM and the failure. In order to discriminate the ok-model from the failure model, n has to be at least as long as the shortest path in which effects of the fault becomes observable. One solution to this problem could be to start with a small number of steps and increase it until the system produces some tests.

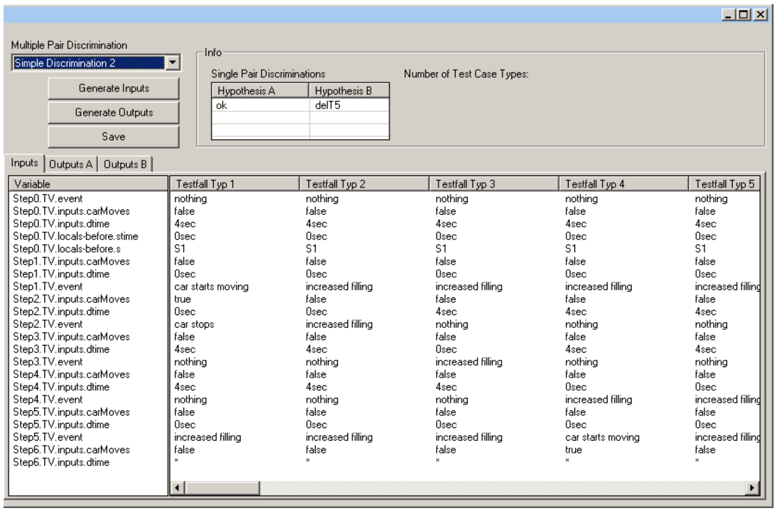
Each vector $(s^0,l^0,i^1,e^1,...,i^n,e^n)$ represents a possible test input ti of a n-step-model. Thus, ti comprises n parts, one for each time step and, hence, expresses a **temporal sequence** of stimuli. The same holds for every observation: obs = $(o^1,...,o^n)$.

A violation of locality becomes evident when the set of transitions is changed, e.g. by deleting, adding, or modifying one. In such cases, not only the respective CT_i component has to be removed, added, or changed, but also the default behavior in $CT_{Default}$ has to be updated.

5 Fault Models

As described in section 2, our approach to testing is based on trying to confirm the correct behavior by refuting the models of possible faulty behaviors. When testing systems that are composed of physical components only, these models are obtained in a natural way from the fault models of elementary components, which usually have a small set of (qualitatively different) foreseeable misbehaviors due to the underlying physics. Faults due to additional interactions among components are either neglected or have to be anticipated and manifested in the model. In summary, for physical systems, the specific realization of the system determines the possible kinds of misbehavior, and testing compares them to a situation where all components work properly.

In software testing (but also in debugging designs of hardware), this does not apply. First, the space of possible faults is not restricted by physical laws, but only by the creativity of the software developer when making mistakes.



Hypothesis A	Hypothesis B
ok	delT5

Variable	Testfall Typ 1	Testfall Typ 2	Testfall Typ 3	Testfall Typ 4	Testfall Typ 5
Step0.TV.event	nothing	nothing	nothing	nothing	nothing
Step0.TV.inputs.carMoves	false	false	false	false	false
Step0.TV.inputs.dtime	4sec	4sec	4sec	4sec	4sec
Step0.TV.locals-before.stime	0sec	0sec	0sec	0sec	0sec
Step0.TV.locals-before.s	S1	S1	S1	S1	S1
Step1.TV.inputs.carMoves	false	false	false	false	false
Step1.TV.inputs.dtime	0sec	0sec	0sec	0sec	0sec
Step1.TV.event	car starts moving	increased filling	increased filling	increased filling	increased filling
Step2.TV.inputs.carMoves	true	false	false	false	false
Step2.TV.inputs.dtime	0sec	0sec	4sec	4sec	4sec
Step2.TV.event	car stops	increased filling	nothing	nothing	nothing
Step3.TV.inputs.carMoves	false	false	false	false	false
Step3.TV.inputs.dtime	4sec	4sec	0sec	4sec	4sec
Step3.TV.event	nothing	nothing	increased filling	nothing	nothing
Step4.TV.inputs.carMoves	false	false	false	false	false
Step4.TV.inputs.dtime	4sec	4sec	4sec	0sec	0sec
Step4.TV.event	nothing	nothing	nothing	increased filling	increased filling
Step5.TV.inputs.carMoves	false	false	false	false	false
Step5.TV.inputs.dtime	0sec	0sec	0sec	0sec	0sec
Step5.TV.event	increased filling	increased filling	increased filling	car starts moving	increased filling
Step6.TV.inputs.carMoves	false	false	false	true	false
Step6.TV.inputs.dtime	*	*	*	*	*

Figure 6: tests discriminating m_{ok} from m_{delT5}

This space is infinite, and the occurrence of structural faults is the rule rather than an exception. Second, the assumption that correct functioning of all (software) components assures the achievement of the intended overall behavior does not hold. This marks an important difference between testing physical artifacts and software (and also hardware design). For the former, we can usually assume it was designed correctly (which is why correct components together will perform correctly), but for the latter we cannot. It is just the opposite: testing aims at revealing design faults.

In our application, the situation is complicated by the fact that it starts from the functional requirements rather than a detailed software design or even the code which might suggest certain types of bugs to check for (e.g. no termination of a while loop). On the positive side, this may lead to a smaller, qualitatively characterized set of possible misbehaviors.

In our example about the detection of fuel refilling, a failure one might think of is that the software does not poll the car's movement during driving and therefore does not detect a stop. This means the machine stays in its current control state instead of performing T_3. The Transition T_3 could be seen as deleted. The construction of such a failure model could be achieved by applying the following operator on the ok-model:

remove-if-condition: $(m^a, T_i) \rightarrow (m^{a\,\prime})$
where $m^{a\,\prime} = m^a[IF_i \rightarrow \varnothing]$ and $T_i=(s^t, IF_i, s^{t+1})$.
Operation $m^a[A \rightarrow B]$ results in a FSM $m^{a\,\prime}$ which is equal to m^a except that element A is substituted with B.

Another faulty behavior would occur, if the software treats an increased tank level after 8sec in standstill exactly as if the car starts moving. W.r.t. the FSM, this means executing T_6 instead of T_7. The proper failure model can be constructed by the operator

move-if-condition-to: $(m^a, T_i, T_j) \rightarrow (m^{a\,\prime})$
where $m^{a\,\prime} = m^a[IF_i \rightarrow IF_i \cup IF_j,\ IF_j \rightarrow \varnothing]$ and $T_i=(s^t, IF_i, s^{t+1})$.

Similarly, other faults can be specified, which, for instance, change the source or destination state, or modify the state expression.

6 Results

In this section, the discrimination of the failure models

- m_{delT3} = remove-if-condition(m_{ok}, T_3)
- m_{delT5} = remove-if-condition(m_{ok}, T_5)

from the ok model is discussed. A relational model that simulates 7 steps of the FSM is used here.

To **discriminate the two models m_{ok} and m_{delT5}**, 36 different types of tests are generated. Figure 6 lists them, where '*' stands for any value in the domain of the respective variable. The input sequence of the first test could be formulated more naturally as following:

1. starting from the initial state one waits 4s long,
2. then the car starts moving and
3. directly after this, it stops again and
4. one waits again 4s.
5. After waiting a third time 4s,
6. a significant increase of the tank filling is detected.

In test 2, the second and the third event occurring are "increased tank filling". These events are unnecessary. Without these two steps the test input still discriminates the fault from the ok model. The reason that the system generates these is the fixed number of steps of the relational model. So some steps have to be filled with events having no effects but serving as placeholders. This explains why so many different tests are generated. Eliminating unnecessary stimuli is addressed in [Struss 07].

Only two types of tests are generated **to discriminate the two models m_{ok} and m_{delT3}**. They are also among the tests of the previous discrimination: one is the first test discussed above, the other causes the trajectory shown in Figure 7.

Tests discriminating between both pairs (m_{ok} from m_{delT3} as well from m_{delT5}) are the two from the second discrimination, because these are also in the generated set of the second one. In our example, this is not surprising. To distinguish between an ok automata and a fault automata where any transition is deleted, one of these both has always to reach S_6, because this is the only state where the output is different to the one of the others.

7 Related Work

Classical approaches generates tests optimized in respect to a certain coverage criteria like state, transition or MCDC coverage [Beizer 95]. In our approach, with carefully chosen sets of failure models tests will be generated that achieve also classical coverage criteria.

To obtain a state coverage, for example, a set of failure models M_{fail} could be constructed as follows. For each state s_i, there exists one failure model $m_{fail,i}$ in M_{fail} which differs from the ok-model in the output of state expression a_i only. The outputs of these two models are complementary. For the case that m_{ok} is a deterministic automaton, the equivalence is proven in [Esser 05].

Also the diagnoser of [Sampath 96] could be used for test generation (although we are not aware of any publication describing this): In this approach a diagnoser is generated from the system model, both FSMs, for calculating diagnosis and diagnosability. The transitions of a diagnoser are labeled with observable events, whereas the states are labeled with the behavior modes consistent with the events that occurred so far. For test generation, the set of observable events could be split into causal and non-causal observable events. Then, the task is to find a causal event sequence where each diagnoser path consistent with these causal events and any possible non-causal observables has either only a ok-label or no ok-label at all. Each sequence of causal and observable events is a valid test for the modeled failures. We expect that the two approaches can be transformed into each other. An analysis and comparison of the efficiency would be interesting.

8 Discussion

The problem which is central to our approach is finding appropriate fault models representing realistic and relevant faults. On the one hand, they are difficult to obtain for software and even more so, when one starts from a functional specification, as we do. This may seem to be a disadvantage in comparison with the other testing heuristics, like coverage criteria. However, it is not true that they do not involve fault models. In fact, they **are based** on assumptions about possible faults, but these are **implicit**. The fact that our approach makes them explicit is a major advantage and the basis for more progress. It also bears the potential to generate tests whose power and coverage grows together with the refinement of the specification during the development process.

We consider the results of this experiment as encouraging and will continue this work in a project with Audi AG. It has raised a number of issues that need to be addressed in this project.

A basic one concerns the question whether the current modeling formalism, a specific type of finite state machine, is appropriate. This has several aspects: First, it has to be checked whether it is **expressive** enough to capture the requirements on embedded software. Second, the impact of the representation on the **complexity** of the algorithm has to be analyzed (Handling absolute time is an important issue, as stated below). These aspects have to be confronted with the most important guideline: appropriateness for **current practice**.

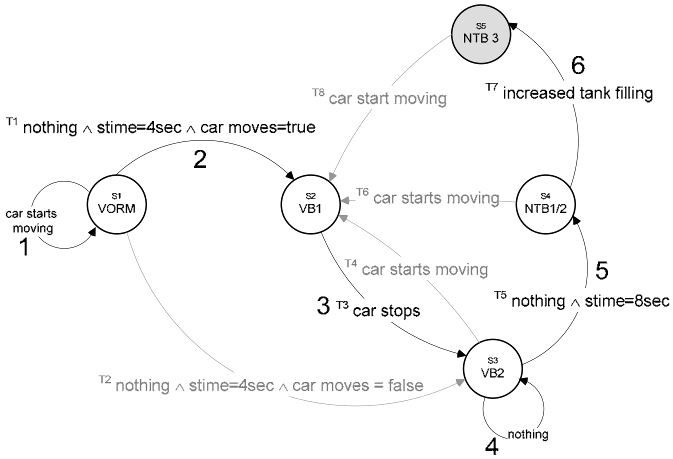


Figure 7: trajectory of the FSM for a test input discriminating m_{ok} from m_{delT3}

Our project is not an academic exercise, but aims at tools that can be easily used in the actual work process. Current requirement specifications at the development stage that matters in our context comprise mainly natural language text together with a few formal or semi-formal elements, such as state charts (provided they are written at all!). Assuming the existence of formal, executable specifications is unrealistic. Any formal representation of the requirements as we need them as an input to our tools needs to take into account whether they can be produced in the current process, by the staff given its education and background, and the limited efforts that can be spent in a real project where meeting deadlines and reducing development time has top priority. Whenever the use of new tools and additional work is required, this needs a rigorous justification by a significant pay-off (in our case in the time spent on testing and the quality of its results).

On the technical side, an adequate handling of time is needed. In our example, time elapsing in a particular state (e.g. “8s with no motion”) seems to be local. However, the respective event has to be stated in a way that can be interpreted properly in other states as well, which may have been reached due to a fault. Introducing global absolute time tends to enforce using the smallest time increments required for some state and event, which appears prohibitive.

Acknowledgements

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

References

[Esser 05] Esser, M: *Modellbasierte Generierung von Tests für eingebettete Systeme am Beispiel der Tankanzeige in einem Kraftwagen*, Technical University of Munich, 2005

[Beizer 95] Beizer, B.: *Black-Box Testing*, John Wiley and Sons, New York, NY, 1995

[Sampath 96] Sampath, M., Senupta, R., Lafortune, S., Sinnamohideen, K., Teneketzis, D.: *Failure Diagnosis using Discrete Event Models*. In: IEEE Transactions on Control Systems Technology, 4(2) 1996, pp. 105-124

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

[Struss 07] Struss, P.: *Model-based Optimization of Testing through Reduction of Stimuli. 20*th International Joint Conference on Artificial Intelligence IJCAI07, Hyderabad, India, 2007