In: 2nd MONET Workshop on Model-Based Systems at IJCAI-05, July 30, 2005.

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 or so, 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 mechanical, electrical, pneumatic, and hydraulic 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, 94a], we presented the theoretical and technical foundations for automated test generation for physical systems based on models of their (nominal and faulty) behavior. Such behaviors are represented as (finite) relations over system variables which characterize the possible states under different modes of behavior. On this basis, tests can be computed as sets of stimuli that trigger disjoint projections of the behavior relations to the space of observables.

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 physical system, 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

In the most general way, testing aims at finding out which hypothesis out of a set H is correct (if any) by stimulating a system such that the available observations of the system responses to the stimuli refute all but one hypotheses (or even all of them).

This is captured by the following definition.

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 $\bot$.

In this definition, condition (i) expresses that there exists an observable system response for each hypothesis under the test input. It also implies that test inputs are consistent with all hypotheses. i.e. we are able to apply the stimulus, because it is causally independent of the hypotheses. Condition (ii) formulates the requirement that the resulting observation **guarantees** that at most one hypothesis will not be refuted, while (ii') states that each hypothesis **may** generate an observation that refutes all others.

While testing, for instance, for fault identification has to discriminate between each single pair of hypotheses (if possible), testing for confirming (or refuting) a particular hypothesis h_0 requires only discrimination between h_0 and any other hypothesis. Usually, one stimulus is not enough to perform the discrimination task which motivates the following definition.

Definition (Confirming Test Input Set)

$\{ti_k\} = \ TI' \subset TI$ is called a discriminating test input set for $H = \{h_i\}$ and $h_0 \in H$ if

$\forall\ h_j$ with $h_0 \neq h_j \quad \exists\ ti_k \in TI'$ such that

ti_k is a discriminating test input for $\{h_0, h_j\}$.

It is called **definitely confirming** if all ti_k have this property, 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. $\vee_i \quad h_i$

Such logical characterizations (see also [McIlraith-Reiter 92]) are too general to serve as a basis for the development of an appropriate representation and algorithms for test generation. Here, the hypotheses correspond to assumptions about the correct or possible faulty behavior of the system to be tested. They are usually given by equations and imple-mented by constraints, and test inputs and observations can be described as value assignments to system variables.

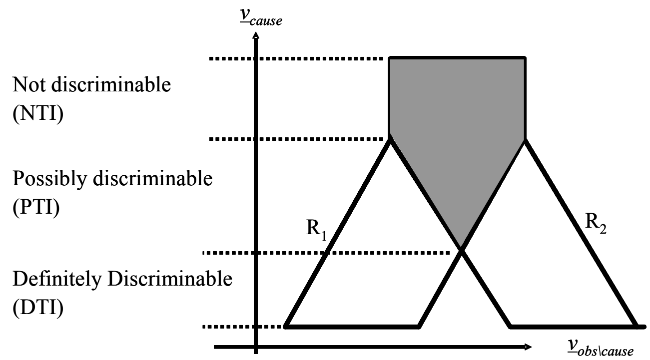

Figure 1 Determining the inputs that do not, possibly and definitely discriminate between R_1 and R_2

The system behavior 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 $DOM(\underline{v}_S) = \ DOM(v_1) \times DOM(v_2) \times \ldots \times DOM(v_n)$. Then a hypothesis $h_i \in H$ is given as a relation $R_i \subseteq DOM(\underline{v}_S)$.

For conformity testing, is h_0 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: $\underline{v}_{cause} \subseteq \underline{v}_{obs} \subseteq \{v_i\}$

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, Figure 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, i.e. stimuli that may lead to an observation in the intersection $p_{obs}(R_i) \cap p_{obs}(R_j)$ shaded in Figure 1. These test inputs we find by projecting the intersection to the causal variables:

$p_{cause}(p_{obs}(R_i) \cap p_{obs}(R_j)\ )$.

The complement of this is the complete set of all test inputs that are guaranteed to produce different system responses under the two hypotheses:

$DTI_{ij} = DOM(\underline{v}_{cause}) \setminus p_{cause}(p_{obs}(R_i) \cap p_{obs}(R_j))$.

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} is the set of all 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 (an assumption which may be relaxed in the otherwise identical discriminability/detectability analysis presented in [Dressler-Struss 03]).

We only mention the fact, that, when applying tests in practice, one may have to avoid certain stimuli because they carry the risk of damaging or destroying the system or to create catastrophic effects as long as certain faults have not been ruled out. In this case, the admissible test inputs are given by some set $R_{adm} \subseteq DOM(\underline{v}_{cause})$, and we obtain
$DTI_{adm,\,ij} = R_{adm} \setminus p_{cause}(p_{obs}(R_i) \cap p_{obs}(R_j))$.

In a similar way as DTI_{ij}, we can compute the set of test inputs that are guaranteed to create indistinguishable observable responses under both hypotheses, i.e. they cannot produce observations in the difference of the relations:

$(p_{obs}(R_i) \setminus \ p_{obs}(R_j)) \cup (p_{obs}(R_i) \setminus \ p_{obs}(R_i))$.

Then the non-discriminating test inputs are

$NTI_{ij} =$
$DOM(\underline{v}_{cause}) \setminus p_{cause}((p_{obs}(R_j) \setminus p_{obs}(R_i)) \cup (p_{obs}(R_i) \setminus p_{obs}(R_j)))$

All other test inputs may or may not lead to discrimination.

Lemma 2

The set of all possibly discriminating test inputs for a pair of hypotheses $\{h_i, h_j\}$ is given by

$PTI_{ij} = DOM(\underline{v}_{cause}) \setminus (NTI_{ij} \cup DTI_{ij})$.

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

Lemma 3

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 3 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. However, it is worth noting that the test input sets are the minimal ones that **guarantee** the discrimination of h_0 from the hypotheses in H. In practice, only a subset of the tests may have to be executed, because some of them refute more hypotheses than guaranteed (because they are a possibly discriminating test for some other pair of hypotheses) and render other tests unnecessary.

The algorithm has been implemented based on software components of OCC'M's RAZ'R ([OCC'M 05]) which provide a representation and operations of relations as ordered multiple decision diagrams (OMDD).

Finally, we mention that probabilities (of hypotheses and observations) can be used to optimize test sets ([Struss 94a], [Vatcheva-de Jong-Mars 02]).

3 State Charts for Specification of Software Requirements

State charts and finite state machines (FSM) are commonly used in specifications of software requirements. **Figure 2** shows a FSM extracted from a requirement specification delivered 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*.

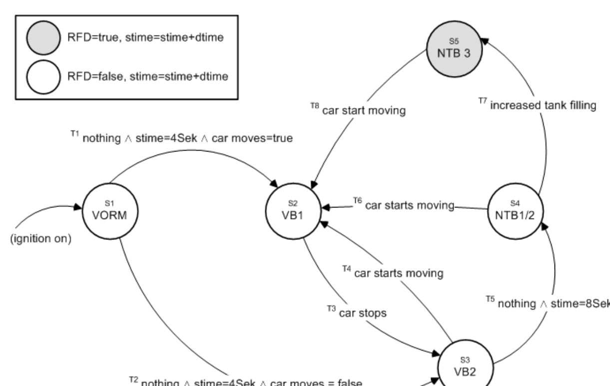

Figure 2: FSM describing a refilling detection in a personal car

5. Set t=t+1
6. 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 a event *car starts moving* can not occur next. In our example, the following rules are necessary:

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

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

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 "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 vector o^{t+1}, reproducing the calculations of the FSM in one step (an 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 $R_{\mathrm{Ca}i}$ 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 $\mathrm{CT}_{\mathrm{Default}}$ (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 $R_{\mathrm{CT}i}$ of these

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

atomic components are:

$$\begin{aligned} R_{CT_i} : \Big\{ \Big(\left(s^{t}, e^{t+1}, i^{t+1}, l^{t}\right), s^{t+1} \Big) \ \Big| \\ \exists_{T_i \in T} \ \Big(T_i = \left(s^{t}, IF, s^{t+1}\right) \wedge \left((e^{t+1}, i^{t+1}, l^{t}) \in IF \right) \Big) \vee \\ \Big(T_i = \left(\%_0^{t}, IF, \%_0^{t+1}\right) \wedge \left(s^{t} \neq \%_0^{t} \vee (e^{t+1}, i^{t+1}, l^{t}) \notin IF \right) \Rightarrow s^{t+1} \in S \Big) \Big\} \end{aligned}$$

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

$$R_{CT_{Default}} = \left\{ \begin{array}{l} \Big(\left(s^{t}, e^{t+1}, i^{t+1}, l^{t}\right), s^{t+1} \Big) \mid \\ \qquad \forall_{T_i = (\%_0^{t}, IF, \%_0^{t+1})} \left(s^{t} \neq \%_0^{t} \vee (e^{t+1}, i^{t+1}, l^{t}) \notin IF \right) \end{array} \right\}$$

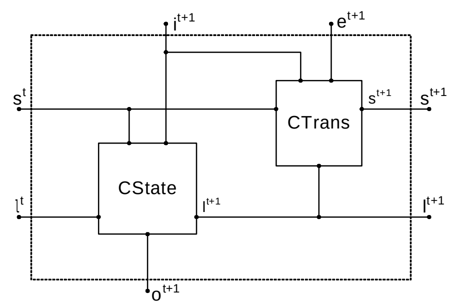

Figure 3: C1Step and its internal structure.

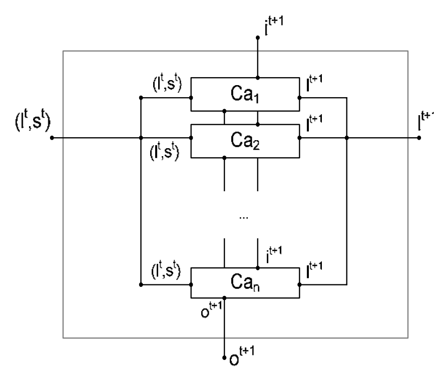

Figure 4: CState and its internal structure.

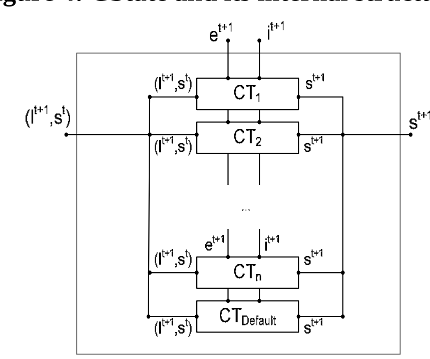

Figure 5: CTrans and its internal structure.

Now one iteration of the operating sequence can be simulated. To simulate n iterations, C1Step is copied n times and placed in serial. But this shows also a limitation: the model can simulate only a fixed number of steps, and the more C1Steps 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 for this problem could be to start with a small amount of steps and increase it until the system produces some tests.

A violations of locality becomes evident when the (set of) transitions are 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, 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. 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. For the former, we can usually assume it was designed correctly (which is why correct components together will perform correctly), but for the software 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 $m^{a'}$ which is equal to m^a except that element A is substituted with B.

Another faulty behavior is that after waiting 8sec in standstill the software behaves the same way both detecting an increased tank fill and detecting that the car starts moving. Looking at 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 $T_i=(s^t, IF_i, s^{t+1})$.

With carefully chosen sets of failure models, one can also generate tests that achieve classical coverage criteria [Beizer95]. To get a state coverage, for example, a set of failure models M_{fail} could be constructed as following. 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 automata, the equivalence is proven in [Esser05].

6 Results and Discussion

In this section the discrimination of two failure models from the ok model is discussed. These failures are:

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

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

Discrimination between m_{ok} and m_{delT3}

Only two types of tests are generated to discriminate these two models. 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.

Discrimination between m_{ok} and m_{delT5}

To discriminate these two models, 36 different types of tests will be generated. The two tests of the previous discriminations are also among them. Some of them are shown in Figure 7. 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

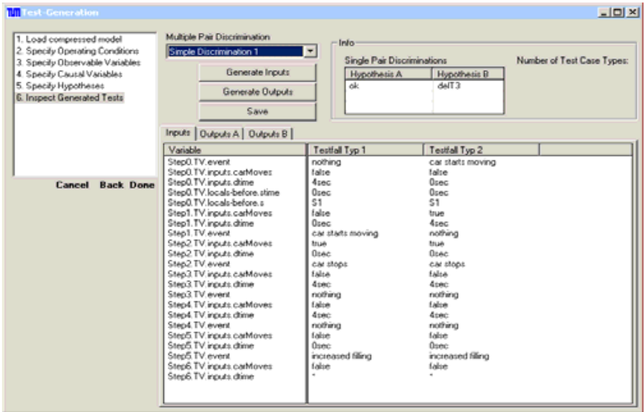

Figure6: tests discriminating m_{ok} from m_{delT3}

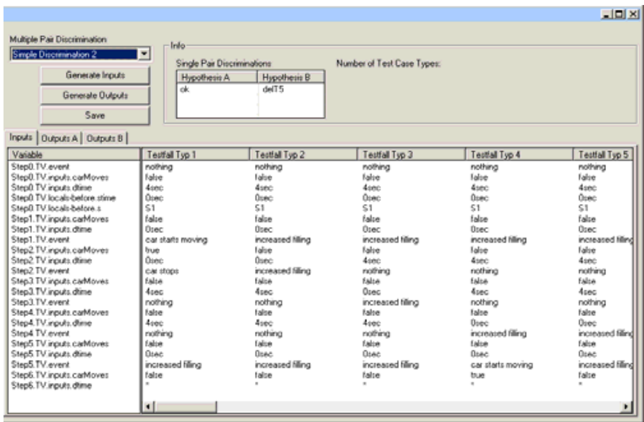

Figure7: tests discriminating m_{ok} from m_{delT5}

having no effects but serving as placeholder. On this account so many different tests are generated.

Discrimination of both pairs

Tests discriminating between both pairs (m_{ok} from m_{delT3} as well from m_{delT5}) are the two from the first 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 a transition is deleted, one of these both has always to reach state S_6, because only there the output is different to the one of the others.

Discussion

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**.

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).

Acknowledgements

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

References

[Dressler-Struss 03] Dressler, O., Struss, P.: *A Toolbox Integrating Model-based Diagnosability Analysis and Automated Generation of Diagnostics*.
In: Proceedings of the 14th International Workshop on Principles of Diagnosis (DX03), pages 99-104, Washington, D.C., USA, June 2003

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

[Beizer95] Beizer, B.: *Black-Box Testing*, John Wiley and Sons, New York, NY, 1995

[McIlraith-Reiter 92] McIlraith, S., Reiter, R.: *On Tests for Hypothetical Reasoning*. In: W. Hamscher, J. de Kleer und L. Console (Hg.). Readings in Model-based Diagnosis. Morgan Kaufmann, San Mateo, 1992

[OCC'M 05] www.occm.de

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

[Struss 94a] Struss, P.: *Testing for Discrimination of Diagnoses*. In: Working Papers of the 5th International Workshop on Principles of Diagnosis (DX-94), New Paltz, USA, 1994.

[Vatcheva-de Jong-Mars 02] Vatcheva, I., de Jong, H., Mars, N.: *Selection of Perturbation Experiments for Model Discrimination*. Proceeddings of ECAI-02, 2002