IFI TECHNICAL REPORTS

Institute of Computer Science,
Clausthal University of Technology

IfI-05-13

Clausthal-Zellerfeld 2005

Reuse of Test Generation Methods for Embedded Systems

Abdelaziz Guerrouat, Harald Richter
Clausthal University of Technology
Department of Computer Science
Julius-Albert-Str. 4
D-38678 Clausthal-Zellerfeld
{aguerrou, richter}@informatik.tu-clausthal.de

Abstract

In this paper test generation methods and appropriate fault models for testing and analysis of embedded systems described as (extended) finite state machines ((E)FSMs) are presented. Compared to simple FSMs, EFSMs specify not only the control flow but also the data flow. Thus, we define a two-level fault model to cover both aspects. The goal of this paper is to reuse well-known FSM-based test generation methods for automation of embedded system testing. These methods have been widely used in testing and validation of protocols and communicating systems. Dependability is one of the most required properties of safety-critical embedded systems because this property relates to avoidance of faults: fault prevention, fault tolerance, fault removal and fault forecasting. This has, however, an implication that development of such systems is usually error-prone and very costly task. One means to attain this property and reduce the development cost is the introducing of formal methods. Indeed, these substantially contribute in the automation of many tasks of the development cycle and allow an unambiguous system specification and testing. In particular, (E)FSMs-based specification and testing is more advantageous because (E)FSMs support the formal semantic of already standardised formal description techniques (FDTs) despite of their popularity in the design of hardware and software systems. In addition, FDTs allow a user-friendly description of complex specifications by attaching single 'E'FSMs to each other in an hierarchical structure.

Keywords

Embedded systems, formal specification, testing and validation, state/transition-based models, formal description techniques

1 Introduction

An embedded computer system represents a part of a large system performing some of the requirements of that system, for example, a computer system used in a car or in an aircraft. Embedded systems cover a large number of computer systems ranging from small devices to large complex systems for monitoring and controlling. Nowadays, the most number of computer systems belong to embedded systems. They are mainly characterised by real-time and dependability properties. Dependability consists of several attributes such reliability, availability, integrity, safety, confidentiality and maintainability [11].

Due to the growing number and complexity of requirements on embedded systems such as reliability, robustness, availability and other characteristics of dependable systems, the software development consequently becomes a very costly and error-prone task. However, the cost plays a central role in the today's industrial competition. The development of competitive and efficient products implies more and more constraints to the design of embedded systems. Non-formal or semi-formal design procedures have demonstrated their limits to efficiently and industrially support the whole design process and thus to provide competitive products. On the other side, formal methods have been successfully used in other areas for both, software and hardware design and testing [9]. They are usually characterised by qualities like abstraction, understandability, analysis, scalability and non-ambiguity.

In particular, formal description techniques (FDTs) [1] [2] and their semantic model are still to get more consideration at a high level of abstraction in the development life-cycle of embedded systems. In fact, formal description techniques have demonstrated their efficiency in the of analysis of complex systems like protocols and communicating systems [9] [10]. Furthermore, they provide a solid mean for unambiguous specification and rigorous analysis. They are based on 'extended' finite state machines ('E'FSMs) and differ from conventional programming languages by providing not only a formal syntax but also a formal semantic. Moreover, the formal specification and testing increase the confidence in the deduced embedded system implementation. Especially in the area of safety-critical systems, for example, steer-by-wire or brake-by-wire in cars [8], the use of formal techniques is highly recommended [3].

In this paper, we present a testing approach and potential test generation methods that could be reused for embedded systems modelled as 'extended' finite state machines. The respective fault models which describe the appropriate error classes each method is able to detect are given. According to the conformance testing used in the protocol engineering area, which is similar to that adopted here as a validation technique, one may speak about errors, if a test sequence execution (called test suite, i.e. a set of test cases) on the implementation (called IUT: implementation under test) leads to an unexpected behaviour of the implementation w.r.t. the specification. Further, we present the principle of a communication model for embedded systems to explain the meaning of communication using 'E'FSMs and FDTs, respectively.

The rest of the paper is organized as follows. In Section 2 we review the conventional finite state machines and the extended finite state machines and explain the difference between them. After that, the principle of a communication model for embedded systems using EFSMs and FDTs, respectively, is outlined. Section 3 presents the testing approach. First, the properties of an embedded system to be tested are identified. This is followed by the presentation of the test generation methods to be reused and the corresponding fault models. We give also a short comparison with other specification models commonly used in embedded systems. Finally, Section 4 concludes the paper.

2 Communication and Specification Models for Embedded Systems

In this section we first review the definition of FSMs and EFSMs on which the specification and testing approaches are based. Further, we give the basic structure of a communication model for embedded systems and then we demonstrate how this principally can be specified in (E)FSMs as intermediate model and in FDTs, respectively. We take Estelle as example of FDTs.

2.1 Preliminaries

Definition 2.1 A *finite state machine* (FSM) is defined as a 5-tuple $<S, I, O, T, s_0>$, where S is a non-empty finite set of states, I a non-empty set of inputs, O a non-empty finite set of outputs, $T \subseteq S \times I \times O \times S$ the set of transition relations, and $s_0 \in S$ the initial state of the FSM.

A transition $t \in T$ of an FSM is a 4-tuple $<s, i, o, s'>$, where $s \in S$ is a current state (the edge), $i \in I$ an input, $o \in O$ an output related to s and I, and $s' \in S$ the next state (a tail state) related to s and i.

FSMs are usually used to specify the control flow of a system, however, they are less appropriate for modelling the data flow. To overcome this inconvenient, FSMs are extended by using additional state variables and interaction parameters. Such variables are used in programming languages specifying conditions on transitions and calculations carried out during transitions.

Definition 2.2 An *extended finite state machine* (EFSM) is defined as a 7-tuple $<S, C, I, O, T, s_0, c_0>$ where S is a non-empty set of main states, $C = dom(v_1) \times \ldots \times dom(v_n)$ a non empty countable set of contexts with $v_i \in V$, V the non-empty finite set of variables and $dom(v_i)$ a non-empty countable set referred to as the domain of v_i, I a non-empty finite set of inputs, O a non-empty set of outputs, $T \subseteq S \times C \times I \times O \times S \times C$ the set of transition relations, $s_0 \in S$ the initial main state, and $c_0 \in C$ the initial context of the EFSM.

A main state may consist of sub-states. A context is a specific assignment of values to the variables. A transition $t \in T$ of an EFSM is a 6-tuple $<s, c, i, o, s', c'>$ where $s \in S$ is a current main state, $c \in C$ a current context, $i \in I$ an input, $o \in O$ an output, $s' \in S$ a next main state, and $c' \in C$ a next context.

Compared to FSMs, EFSMs provide a more generalized specification mean. Taking EFSMs as an intermediate specification model is more advantageous for using formal description techniques like Estelle or SDL [1] [2]. Indeed, EFSMs represent the formal semantic of these FDTs and thus the mapping of EFSMs on FDTs can easily take place regarding the behaviour part.

Estelle is a standardised formal description technique (International Standard ISO 9074) based on concepts of structured communicating extended state automata and Pascal. It is oriented towards the specification of complex distributed systems, in particular communicating systems. A specified system is presented as a tree of tasks where each task has a fixed number of input/output access points (interaction points). Within a specified system it exists a fixed structure of subsystems (sub-trees of tasks) and communication links between subsystems.

SDL (Specification and Description Language) is an object-oriented, formal language defined by The International Telecommunications Union Telecommunications Standardization Sector (ITU) (formerly Comité Consultatif International Télégraphique et Téléphonique [CCITT]) as recommendation Z.100. The language is intended for the specification of complex event-driven real-time, and interactive applications involving many concurrent activities that communicate using discrete signals.

2.2 Communication

An embedded system is any computer system or computing device that performs a dedicated function or is designed for use with a specific embedded software application, e.g. Car, PDA (Personal Data Assistant), Mobile Phone, E-Book (Electronic Book), Robot, etc. That is, an embedded system is a special-purpose system built into a larger device. It is embedded as a subsystem in a larger system which may or may not be a computer system. An embedded system is typically required to meet specific requirements. In an embedded mechatronic system, a microcontroller or computer system performs a dedicated function for an appliance or a gadget such as a car's brake or a steering wheel [8].

Embedded systems must usually be dependable, efficient and must meet real-time constraints. Be 'dependable' means that an embedded system must be reliable, available and safe. The efficiency mostly concerns properties like energy, code-size, run-time, weight and cost. An embedded system is dedicated for a certain application and characterized also by a dedicated user interface. Thus, knowledge about future behaviour at design time can be used to minimize resources and to maximize robustness. Many embedded systems must meet real-time constraints. A real-time system must react to stimuli from the controlled object (or the operator) within the time interval dedicated by the environment.

Embedded systems are frequently connected to a physical environment through sensors and actuators. They are typically reactive systems. A reactive system is in continuous interaction with its environment and executes at a pace determined by that environment. The behaviour depends on input and current state for which the automata model is often most appropriate.

Figure 1 Basic structure of an embedded system

Figure 1 illustrates the basic structure of an embedded system comprising an *external process*, *sensors*, *actuators*, and a *controller*:

The *external process* is a process that can be of physical, mechanical, or electrical nature. *Sensors* provide information about the current state of the *external process* by means of so-called *monitoring events*. They are communicated to the *controller*. For the *controller*, they represent input events. They are considered as stimuli for the *controller*. The *controller* must react to each received event, i.e. input event. Events originate usually from *sensors*. Depending on the received events from sensors, corresponding states of the *external process* will be determined. *Actuators* receive the results determined by the *controller* which are communicated to the *external process* by means of so-called *controlling events*.

The external process is usually given in advance. In contrast, the controller is often implemented by real-time hardware and software. This should allow each modification of the controller algorithm in a straightforward way each time this is needed. The controller's behaviour is depending on that of the external process. The controller commands the behaviour of the external process taking into consideration requirements on the process and its characteristics, such as physical laws, real time and other constraints.

2.3 Specification

From the point of view of communication an embedded system specification consists of the specification of its environment and its controller. We assume that the embedded system is state-transition based because the automata model is efficiently more appropriate. Thus, its behaviour description will be based on the EFSM model. This consists of a set of modules where each module describing a given function is modelled as one or many EFSMs (Figure 2). These modules are attached to each other by means of channels and interact with each other via broadcasting events. However, a sequence and an hierarchy have to be respected in this communication. For instance, the direct communication of a module of an actuator with a sensor is not allowed.

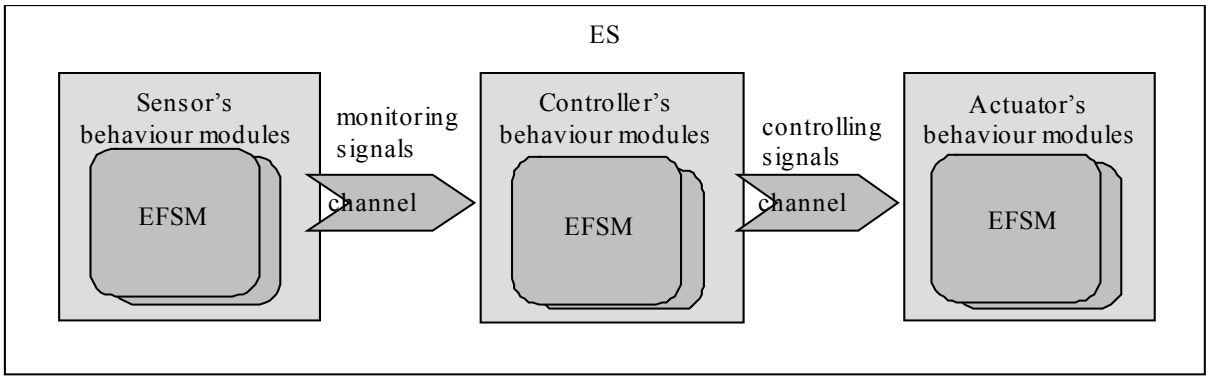


Figure 2 Survey of the communication model for embedded systems

The most important component of an embedded system consists of the controller which communicates with its environment, i.e. sensors and actors, via signals (i.e. events). To be recognized by all components, these events have to be declared as global variables for adjacent EFSMs. The events output from sensors represents input events for the controller. The events from the controller to the actuators are output events and represent input events for the actuators. They result from new computations performed by the controller that is triggered by the received input events.

Depending on the nature of sensor events (e.g. indicating the power on/off state for an electrical unit, the speed of a mobile object such as a car, etc.) the corresponding EFSM of this component is triggered and the concerning transition(s) are performed. This triggers the EFSMs of the controller whose states change. Depending on the received events, transitions in the EFSMs are executed. Note, that transitions in the controller can spontaneously be triggered by other events, e.g. time out. The modelled subsequent state of the external process is computed and communicated as output events via the actuators.

To provide an intermediate specification model which better fits the behaviour part of the considered FDT, i.e. Estelle, we introduce a new EFSM, called p-EFSM (p stands for 'predicated'). This is defined as follows:

Definition 2.3 A *predicated extended finite state machine* (p-EFSM) is an 8-tuple $<S, C, I, P, O, T, s_0, c_0>$ where S is a non-empty set of main states, $C=dom(v_1)$ x … x $dom(v_n)$ a non-empty countable set of contexts with $v_i \in V$, V a non-empty finite set of variables, and $dom(v_i)$ a non-empty countable set referred to as the domain of v_i, P a countable set of predicates (possibly empty), I a non-empty finite set of inputs, O a non-empty finite set of outputs, $T \subseteq S$ x C x I x P x O x S x C a set of transition relations, $s_0 \in S$ the initial main state, and $c_0 \in C$ the initial context of the p-EFSM.

p-EFSM extends a bit the conventional EFSMs for FDT mapping purposes as we will see later. p-EFSM is similar to EFSM except that in a p-EFSM the conditions on transitions are explicitly specified. This is just a notation facility and functionally and conceptually there is no difference between both models. In the rest of the paper, we indifferently address both models.

Thus, a transition $t \in T$ of a p-EFSM is a 7-tuple $<s, c, I, p, o, s', c'>$ where $s \in S$ is a current main state, $c \in C$ a current context, $i \in I$ an input, $p \in P$ a enabling predicate which depends on the context c, $o \in O$ an output, $s' \in S$ a next main state, and $c' \in C$ a next context.

We consider one or more p-EFSMs for each component of the system and denote them with indices s, c and a for *sensors*, *controller*, and *actuators*.

Interdependencies between these components are described as follows:

- Let t_s be a transition
  $t_s \in T_s$: $t_s = <s_s, c_s, i_s, p_s, o_s, s'_s, c'_s>$ with
  $s_s \in S_s$, $c_s \in C_s$, $i_s \in I_s$, $p_s \in P_s$, $o_s \in O_s$, $s'_s \in S_s$, $c'_s \in C_s \Rightarrow \exists\, t_c \in T_c \mid o_s \equiv i_c$

  That is, each output event generated by sensors must trigger a transition of the controller. This event represents an input event for the triggered transition. We assume here that the predicates related to the transitions are satisfied by the actual context.

- Let $t_c \in T_c$ be a transition with $s_c \in S_c$, $c_c \in C_c$, $i_c \in I_c$, $p_c \in P_c$, $o_c \in O_c$, $s'_c \in S_c$, $c'_c \in C_c$,
  if $i_c \in \mathrm{O_s} \Rightarrow \exists\, t_s \in T_s$ and $i_c \equiv o_s$.

  This means that, if there exists a transition of the controller whose input event belongs to the set of output events of the sensors then it must exist a transition of the sensors whose output event is identified with the given event.

- Let $t_a \in T_a$ be a transition: $t_a = <s_a, c_a, i_a, p_a, o_a, s'_a, c'_a>$ with $s_a \in S_a$, $c_a \in C_a$, $i_a \in I_a$, $p_a \in P_a$, $o_a \in O_a$, $s'_a \in S_a$,
  $c'_a \in C_a \Rightarrow \exists\, t_c \in T_c$: $t_c = <s_c, c_c, i_c, p_c, o_c, s'_c, c'_c>$ and $o_c \equiv i_a$.

  Each transition of actuators must be only triggered by the controller and must match the output event of the triggering transition of the controller.

(p)-EFSMs describe thus system components that may be blocks or modules depending on the used formal description technique[1] which are linked together by means of channels via interaction points (IP) to build the whole embedded system specification (s. Figure 3).

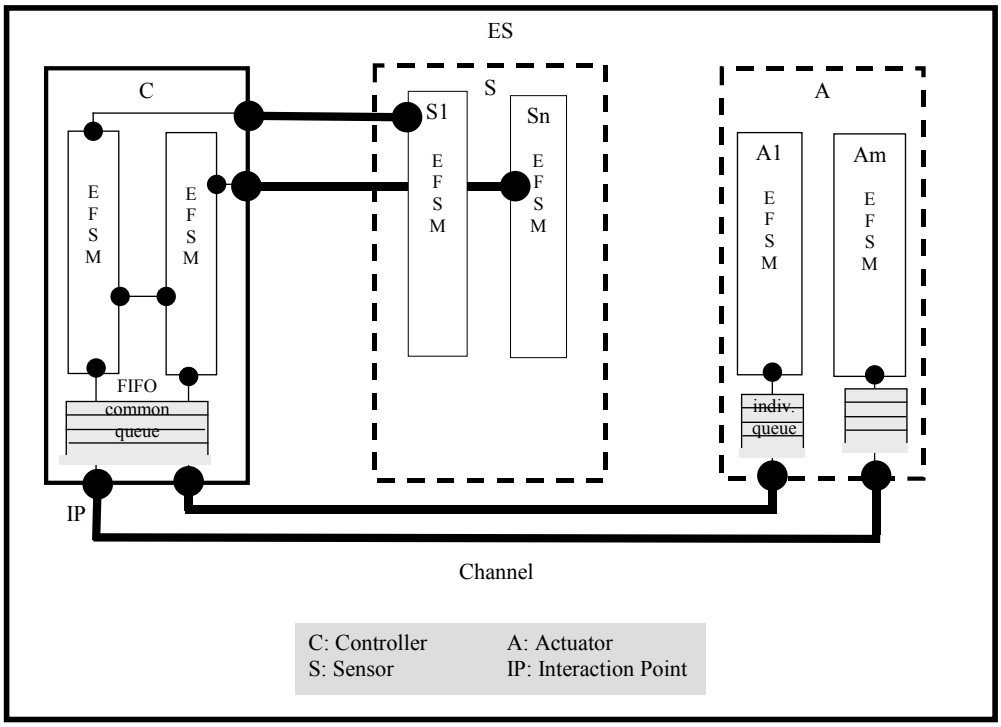


Figure 3 ES specification based on Estelle

A specified embedded system is a tree of tasks ((p)-EFSMs) which can be categorized in three classes corresponding to controller, sensors and actuators modules. They are organized in a hierarchical structure (parent-son-relationship) w.r.t. the principal structuring concepts of Estelle[2] (Figure 3).

[1] SDL uses the 'block' concept whereas Estelle 'module'
[2] In a similar way one can use SDL.

Each task has a fixed number of Input/Output access points (interaction points) which can be associated to controller, sensors or actuator modules. Bidirectional communication links may exist between tasks (between their interaction points).

Within a specified embedded system there exists a fixed structure of subsystems (sub-trees of tasks), corresponding to controller, sensors or actuators, and of communications links (between them). Within a subsystem both structures (of tasks and communication links) may change dynamically. Tasks exchange interactions in the following way:

- A task may send an interaction through its interaction point to a task linked to it, e.g. from C to A (controlling events) via the interaction points in C and A which are link to each other.
- An interaction received by a task, as its interaction points, is appended to a FIFO queue associated to this interaction point. A FIFO queue may be either associated to one interaction point (individual queue) or to many interaction points (common queue).

It is easy to map a p-EFSM specification on the behavioural part (transition part) of an Estelle module. The later has the following structure which is composed of two parts - condition clauses and actions:

```
WHEN clause
      when interaction_point_id.interaction_name
FROM clause
      from state
PROVIDED clause
      provided Boolean expression
DELAY clause
      delay (integer_expression)
TO clause
      to state
      output
```

To map a given (p)-EFSM on an Estelle module (behaviour part), one has to match each transition of the (p)-EFSM with the corresponding transition part of the given Estelle module. Thus, the *when clause* corresponds to an input event in p-EFSM (for a given transition), *from* to edge state, *provided* to the condition on the transition (predicate), possibly *delay* to a timing special input event, *to* to the tail state and *output* to output event.

3 Fault Models and Test Generation Methods

The main validation techniques that can be used in or adapted to the embedded system engineering are of two kinds: verification and conformance testing. The verification approach deals with system specification and tries to prove its correctness based on the so-called white box. In this case, the user properties are specified by another formalism as temporal logic and verified by commonly using model-checkers. The second approach, the conformance testing, deals with a system implementation and tires to find incorrectness on it without considering its intern structure, a so-called black-box testing. Test sequences (called test cases) are derived from the specification and are executed on the implementation (called IUT: implementation under test). According to the test purposes of the executed test cases and the analysis of the verdict of the test execution, it will be stated whether the implementation is conform to the specification or not. A test case is composed of a test preamble (a sequence of actions leading to a given state or action), test body (actions to test), and of a test postamble (sequence of action allowing to go back to the initial state).

The present work is based on the second validation technique and thus we want to adopt the conformance testing approach for the validation of embedded systems. This conducts us to define what an error is and how test sequences can be extracted from a specification assuming (E)FSMs as intermediate specification models for embedded systems.

3.1 Fault Models

Analysis and Testing of embedded systems have to prove correctness, completeness and consistency in early phases of system development. Correctness means the fulfilment of the required services and its providing within a given time period. Completeness is its act of reaction to all possible events and carrying out all services. Consistency relates to the interior contradiction freeness of the specification.

There are two kinds of testing, general and special. The first one consists of testing of properties that must be held independently of special semantics of the developed system (consistency), such as livelock and deadlock-freeness, limitedness and resynchronization. The second aims at properties that are determined by the semantics of the designed system.

Properties that are commonly addressed by analysis and testing are summarized as follows:

- *The non-existence of non-executable actions:* The system comprises no actions that cannot be executable under normal conditions.
- *Liveliness:* Each state of the system is reachable from the initial state.
- *Deadlock-freeness:* The system reaches no state that does not allow to interact with the environment and never leaves it.
- *Livelock-freeness:* The system comprises no non-productive cycles.
- *Error tolerance and resynchronization:* The system reaches a normal state within a limited time period after an error leading to an abnormal state has been occurred.
- *Safety:* The system comprises no unspecified events.
- *Partial correctness:* The system provides a special service when it terminates.
- *Termination:* The system reaches each time the final state(s), or the initial state for cyclic systems.
- *Error behaviour freeness:* This is performed by testing the implementation and comparing the result behaviour with the specification.

Precise specifications are essential to allow the analysis of embedded systems. The use of formal methods enables the automation of most aspects. In this work we are particularly interested in the problem of detecting erroneous behaviour in embedded system implementations which are state-transition-based. This problem is defined as follows:

Definition 2.3 The problem of erroneous behaviours detection in an embedded system implementation is the problem for deciding whether the corresponding (E)FSMs contains errors by means of testing based on an the appropriate fault models.

The large number and the complexity of embedded systems failures requires from a practical testing approach to avoid working directly with failure cases. Indeed, in most cases, one is most concerned with detecting the presence or absence of any failure. Many failures may very well cause the same error for a given test or set of tests. One method to resolve this problem is the use of fault models to describe the effects of failures at some higher level of abstraction. If the fault model describes the faults accurately, then one needs only to derive tests to detect all the faults in the fault model. This approach has several possible advantages. A higher-level fault describes many physical and software faults, thus reducing the number of possibilities to be considered in the generation of tests.

For a state transition-based embedded system, an appropriate fault model can be defined. We consider a two-level fault model which corresponds to the control flow and the data flow, respectively. The first level of the fault model deals with systems specified as simple FSMs whereas the second level the extension of FSMs, i.e. with (p)-EFSM specifications.

In the first level of the fault model, the following fault classes are defined for faulty implementations. To explain the principal we consider a very simplified example of the controller behaviour model as FSM (only the control flow) consisting of three states: lp (low pressure), hp (high pressure) and mp (middle power) (s. Figure 4). These correspond to the reaction of the controller through commanding an actuator depending on the pressure measured by sensors.

- *Output errors class:* A transition has an output fault if, for the corresponding state and input received, the machine provides an output different from the one specified (Figure 4).

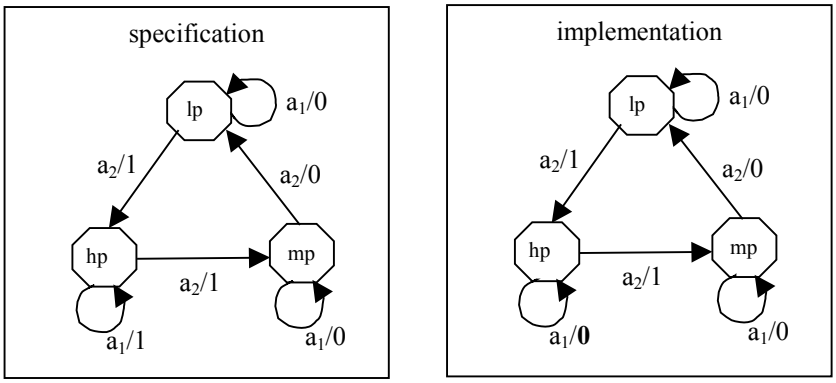


Figure 4 Output Errors class

- *Transfer errors class:* A transition has a transfer fault if, for the corresponding state and input received, the machine enters a different state than expected (Figure 5).

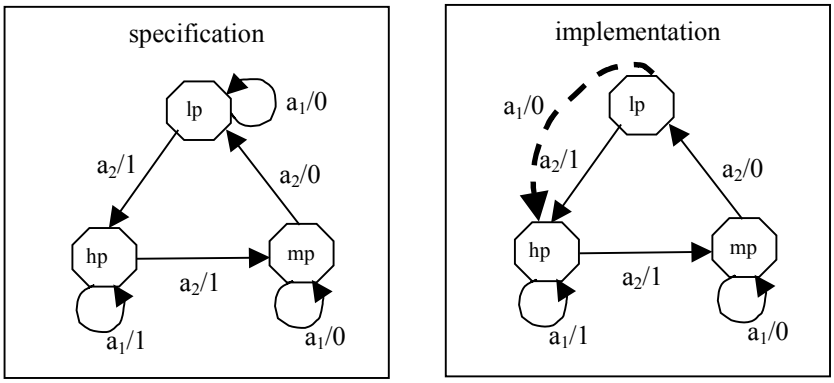


Figure 5 Transfer errors class

- *Transfer errors with additional states class:* here it is supposed there is more states as specified with possible transfer faults (Figure 6).

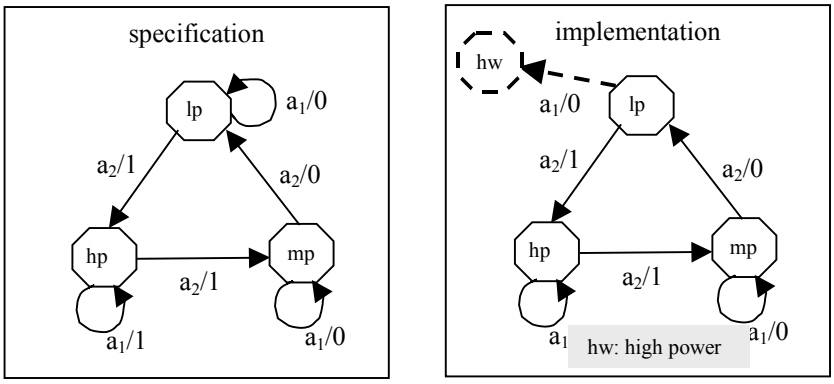


Figure 6 Transfer errors with additional states class

- *Missing states errors class:* In this case there are less states as specified. This is usually due to non-deterministic behaviours and/or incompleteness.

In the second level of the fault model (for extended FSMs), the above error classes (of the first level) are extended by including other error classes which mostly are similar to those of software:

Specific behavioural error classes:

- *Interaction point errors:* A different interaction point should be used for a given transition.
- *Context errors:* Either the number and/or types of the parameters of a given transition is not correct or parameter value should be different.
- *Predicate errors:* In the case that a Boolean expression is associated with a transition, the Boolean expression is evaluated to false in the specification and/or the variables of the Boolean expression are erroneous.
- *Delay errors:* The corresponding interaction should occur before occurring the timeout.

General data flow-related error classes:

- Sequencing errors
- Arithmetic and manipulative errors
- Calling functions errors class (wrong calls)
- Data types specification errors
- Value errors (wrong values)
- Variables number errors
- Operator errors

3.2 Test Sequences Generation Methods

There exist many test generation methods that are based on FSMs. Some of them are able to detect only certain errors classes of the fault model given above, whereas other allow to cover all errors classes.

All test generation methods based on FSMs have a common basic idea. A test sequence is preferably short sequence of consecutive transitions that contains every transition of the FSM at least once and allows to check whether every transition is implemented as defined. To test a transition, one has to apply the input for the transition in the starting state of the transition, to check whether the correct output occurs, and to check whether the correct next state has been reached after the transition. Checking the next state might be omitted (transition tour method) or be carried out by means of distinguishing sequences (checking experiments method), characterizing sequences (W-method), or unique input/output sequences (UIO methods). Some of these methods were also extended to nondeterministic FSMs [7].

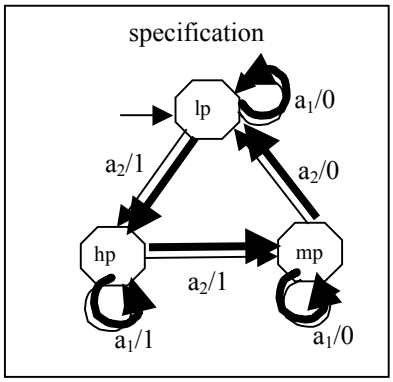
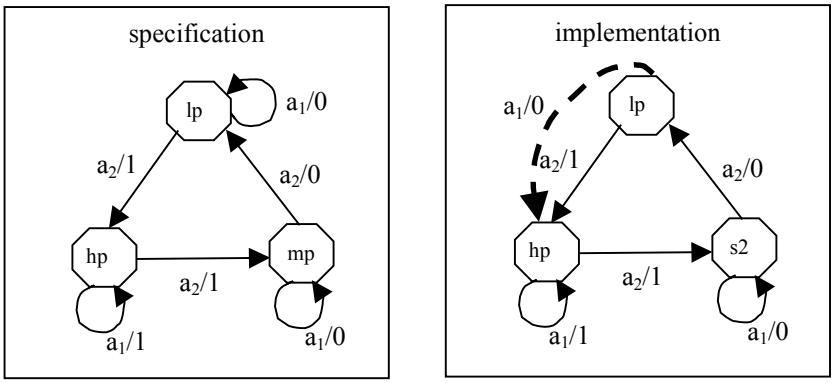


Figure 7 Test generation based on TT method

- *The transition tour method (TT):* This method is able to detect any set of output faults in the absence of transfer faults. A transition tour of a FSM is a path (test sequence) starting at the initial state, traverse every transition at least once, and returns to the initial state (Figure 7). From a transition tour, one can identify a test suite consisting of an input sequence and its expected output sequence based on the specification: the

input sequence $a_1a_2a_1a_2a_1a_2$ and its expected output sequence *011100*. The transition tour can find all output faults, e.g. the faulty observed output sequence *111100* is detected when executing the input sequence $a_1a_2a_1a_2a_1a_2$ on the implementation. However, the TT-method is not able to detect transfer errors (Figure 8).



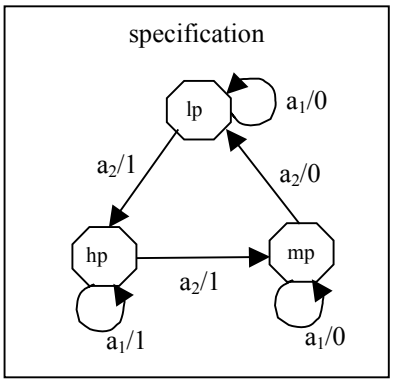
Input sequence: $a_1a_2a_1a_2a_1a_2$
Expected output sequence: 011100

Input sequence: $a_1a_2a_1a_2a_1a_2$
Expected output sequence: 010001

Figure 8 TT-method cannot detect transfer faults

- The DS-method (Distinguishing Sequence) and the UIO-method (Unique Input/Output Sequence) detect any set of output and transfer faults, assuming that the number of states is still the same as in the specification. An input sequence is a *distinguishing sequence* if after applying the input sequence, one can determine the source state by observing the output sequence (Figure 9). Let *s* be a state; an input sequence is a UIO sequence for *s* if after applying the input sequence, on can state whether the source state s is or not by observing the output sequence.

- The W-methods (Characterising Set) detect in addition transfer faults with additional and missing states. A set of input sequences is a characterising set if after applying all input sequences in the set, one van determine the source state by observing the output sequences.



‚a1' is not a distinguishing sequence

initial state	input seq.	output seq.	final state
lp	a_1	0	lp
hp	a_1	1	hp
mp	a_1	0	mp

‚a1a2' is a distinguishing sequence

initial state	input seq.	output seq.	final state
lp	a_1a_2	01	lp
hp	a_1a_2	11	hp
mp	a_1a_2	00	mp

Figure 9 DS-Method

DS-Method, UIO-Method and W-method have the following main idea: Generate a test suite that, for every transition (*s*, *i*, *o*, *s'*):

1. Puts the implementation into state *s* (setup)
2. Applies input *i* and check whether the actual output is *o* (output error)
3. Determines whether the target state of the implementation is *s'* (transfer fault)

3.2 Test Sequence Generation based on EFSMs

The main problem of the test sequences generation methods is that FSMs can only specify the control flow of an embedded system. To specify both the control and the data flow, EFSMs, which are based on FSMs extended by variables, should be applied. As seen in Section 2.3, EFSM models form the basis of the standardized formal description techniques. In order to apply FSM based methods, the EFSM should be transformed into an equivalent FSM.

Theoretically, provided that all variables of an EFSM have a finite, countable domain, an EFSM can be transformed into an equivalent FSM. The transformation leads to the removal of variables from the state machine and an increase in the number of states. In fact, an EFSM can be viewed as a compressed notation of an FSM. It is possible to unfold it into a pure FSM by expanding the values of variables. But, their domain have to be reduced in order to avoid the state and input explosion problem. In this case, the FSM-based methods cover not only the control flow but also the data flow.

An approach to bridge that gap between an EFSM and an FSM avoiding inordinate increase in the number of states has been discussed in [8]. The aim is to make FSM-based methods for test generation applicable to EFSM. The approach is discussed on the basis of one-module Estelle normal form specification representing an EFSM. An expanded EFSM that is equivalent to the original EFSM can be generated by applying a transformation algorithm without loss of important information. The expanded EFSM can be then interpreted in terms of FSM.

Such a transformation is based generally on the classification of state and input variables and taking into account only the variables that effectively influence the control flow. If their domains are originally small (which is usually the case for the most real-life systems) or reasonably reduced, then the state explosion problem can be alleviated. Furthermore, the minimization of the obtained FSM will lead to more reduce the number of states and makes the state explosion problem less severe. This takes place through eliminating all equivalent states of the FSM by applying a well-known minimization algorithm.

For the resulted FSM, classic test sequences generation methods, such the transition tour method, W-method, UIO method, etc. can be applied. This allows to generate test sequences that cover the control and data flows of the original (predicated) EFSM.

3.3 Related Models

Statecharts as a semi-formal model is actually the mostly used formalism to specify requirements for embedded systems [4]. Although Statecharts provide graphical facilities, they might lack formal and unambiguous semantics. Therefore, detecting bugs, incompleteness and inconsistencies becomes a difficult task. Furthermore, they are only used to describe behavioural requirements. To alleviate these lacks many authors try to combine formal notations like Z with state-transition models [5]. Z is based on set theory and first order predicate logic and used for data structuring and abstracting. Petri Nets have been also used as specification models for embedded systems to deal with their verification, e.g. [12]. However, approaches developed around this model do not clearly address formal test data generation methods, e.g. for testing purposes. In addition, they don't relate to standardised formal description techniques. Further, the readability and understandability of a Petri Net specification becomes difficult with the growing complexity of a system. On the other side, formal description techniques provide clear specifications because the combination of the single EFSMs to build the whole specification takes place just by linking (attaching) them (EFSMs). Thus, a formal syntax and a formal semantic are well supported by FDTs.

4 Conclusion

In this work we have presented the principle of a validation approach based on conformance testing that can be reused for embedded systems described as (extended) finite state machines. This implied appropriate test generation methods and related fault models. The fault model consists of a two-level fault model corresponding to the control flow and data flow, respectively. An appropriate communication model for embedded systems and its mapping on EFSMs and on formal description techniques as Estelle have been also discussed. We have identified the different analysis and testing issues and especially dealt with checking erroneous behaviour of embedded system implementations against the specification. The main goal of the work is the reuse of already standardized specifications languages and the testing methods that have been widely around them or for their semantic model. The FDT and (E)FSM formal specification proposed allows the automation of many tasks of

the embedded system development process, especially the test generation. The FDTs as specification languages are characterized not only by a formal syntax but also a formal semantic and have been successfully used in the formal design of many communication protocols and communicating systems.

We are refining the here proposed validation approach by particularly defining an appropriate testing architecture. We are also developing a knowledge-based diagnosis system to explain the reasons of errors in faulty implementations after test suites execution (sets of test cases), i.a. after analysing the test verdicts. In addition, we plan to investigate real-life embedded systems especially from the automotive area to study the extent of the application of the analysis and testing approach.

References

[1] Specification and Description Language SDL'92. ITU-T Recommendation Z.100, 1992.

[2] Information processing systems – Open Systems Interconnection – Estelle: A formal description technique based on an extended state transition model. International Standard ISO 9074, 1989.

[3] R. Buessow, R. Geisler, and M. Klar. Specifying safety-critical embedded systems with statecharts and Z: A case study. In Proceedings of Fundamental Approaches to Software Engineering (FASE'98), Lisbon, 1998.

[4] M. Mendler, G. Luettgen. Statecharts: From Visual Syntax to Model-Theoretic Semantics. In K. Bauknecht, W. Brauer, and Th. Mück (editors), Workshop on Integrating Diagrammatic and Formal Specification Techniques (IDFST 2001), pages 615-621, Vienna, 2001.

[5] B. Potter, J. Sinclair, and D. Till. Introduction to Formal Specification and Z (2nd Edition). Prentice Hall PTR; 1996.

[6] A. V. Aho et. al. An optimisation technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman Tours. In S. Aggarwal and K. Sabnani, editors, Protocol Specification, Testing, and Verification, New Jersey, 1988.

[7] S. Fujiwara, G.v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. IEEE transaction on Software Engineering 17(6): 591-603, 1991.

[8] H. Richter et al. A Concept For a Reliable, Cost-Effective, Real-Time Local-Area Network for Automobiles. In Proceedings of Joint conference Embedded in Munich and Embedded Systems, Munich, 2004.

[9] O. Henniger, A. Ulrich, and H. König. Transformation of Estelle modules aiming at test case derivation. Chapmann & Hall, 1995.

[10] H. Fouchal, et al. Generation of timed automata from Estelle specifications. In International Workshop on the Formal Technique ESTELLE, Evry, France, 1998.

[11] A. Avizienis, J-C. Laprie, and B. Randell. Fundamental Concepts of Computer System Dependability. IARP/IEEE-RAS Workshop on Robot Dependability: Technological Challenge of Dependable, Robots in Human Environments, 2001.

[12] Corts, L. A., P. Eles, and Z. Peng, Verification of embedded systems using a petri net based representation, In Proceedings of the 13th international symposium on System synthesis, Madrid, 2000.

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Wojciech Jamroga
Contact: wjamroga@in.tu-clausthal.de
URL: http://www.in.tu-clausthal.de/~wjamroga/techreports/
ISSN: 1860-8477

The IfI Review Board

Prof. Dr. Jürgen Dix (Theoretical Computer Science/Computational Intelligence)
Prof. Dr. Klaus Ecker (Applied Computer Science)
Prof. Dr. habil. Torsten Grust (Databases)
Prof. Dr. Barbara Hammer (Theoretical Foundations of Computer Science)
Prof. Dr. Kai Hormann (Computer Graphics)
Dr. Michaela Huhn (Economical Computer Science)
Prof. Dr. Gerhard R. Joubert (Practical Computer Science)
Prof. Dr. Ingbert Kupka (Theoretical Computer Science)
Prof. Dr. Wilfried Lex (Mathematical Foundations of Computer Science)
Prof. Dr. Jörg Müller (Agent Systems)
Dr. Frank Padberg (Software Engineering)
Prof. Dr.-Ing. Dr. rer. nat. habil. Harald Richter (Technical Computer Science)
Prof. Dr. Gabriel Zachmann (Computer Graphics)