A Concept for Testing and Diagnosis of Embedded Systems based on Extended Finite States Machines

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

Abstract:
Due to the growing complexity of embedded systems and their requirements in reliability and real-time capability, support methods to help designers and testers at the different phases of the development cycle are needed. In areas such network protocol testing automatic test generation has been used successfully. Here, the method of extended finite state machines is applied to embedded systems. Additionally, fault models for testing and test result analysis are given to automate error detection.

1 Introduction

In the past, automatic diagnose systems like EMYCIN and MED2 were successfully used in medicine, and it was shown that some principles of these systems, such as finite state machines can be reused, e.g. for network protocol testing [6] [7]. In this paper, we present extended finite state machines for automatic testing and test result analysis. Applications are mechatronic systems in cars, for instance.

The software development cycle for embedded mechatronic systems is costly and error-prone. Costs and proper functioning play a central role in today's industrial competition. The development of products in due time is imposing constraints to the design. One of the means to reach the goals is formal methods to support all phases of embedded system development, i.e. specification, synthesis and testing. Additionally, in the area of safety-critical systems in cars, such as steer-by-wire or brake-by-wire the use of formal techniques is highly recommended [8].

There are several properties desired for formal methods. They should be abstract, understandable, analysable, scalable and unambiguous. The known finite state machine method has become popular for control flow specification of state/transition-based systems, and many related analysis methods have been developed. These comprise automatic test derivation

for function validation, e.g. However, finite state machines are limited to control flow specification. Extended finite state machines allow both, data and control flow specification. One of the most important topics for embedded systems is test result analysis, i.e. the error diagnostic of faulty implementations. Error diagnostic usually requires a lot of human knowledge and experience because one has to consider a huge number of symptoms and disturbances of the system under test. Automated diagnostic systems can help finding errors by means of semantic reasoning. Rapid and goal-directed deduction of error symptoms is provided by an automated testing system.

Much work is to be done for the practical use of formal description techniques at a high level of abstraction in the life-cycle development of embedded systems. We show that (E)FSMs ('extended' finite state machines) provide a good mean for unambiguous specification and automatic test data generation. EFSMs differ from conventional programming languages by providing not only a syntax but also formal semantics. In [6] and [7] examples of test generation based on (E)FSM specifications are presented. However, the (E)FSMs shown there cannot be easily applied because of the so-called state explosion problem.

This paper presents a diagnostic system, called EDiagnostiquor[1], we are currently developing to support the test result analysis. EDiagnostiquor aims to test and improve the interplay and communication between the components of an embedded system, i.e. between sensors, controller, actuators, and the external process to be controlled. Out of the view of the external process EDiagnostiquor is a black-box test method. It is based on the observable behaviour of the external process without considering its internal structure. A test suite, i.e. a set of test cases is carried out, that is automatically derived from the formal specification of an embedded system.

Section 2 shows the basic structure of embedded systems and their design process, and describes EDiagnostiquor. The corresponding fault model is also discussed. In Section 3, the diagnostic principles are demonstrated by means of an example. Finally, Section 4 gives summary and outlook to future work.

2 Description of EDiagnostiquor

2.1 Basic structure of embedded systems

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

[1] *E* stands for Embedded systems.

brake or a steering wheel. The embedded system is part of a larger unit comprising electrical and mechanical components and must meet real-time requirements. It has to react to stimuli from the controlled process within a time interval dictated by this process. An embedded system is connected to the physical environment through sensors and actuators. Its output depends on input and internal state for which a finite state machine is often an appropriate model. Software for embedded systems must be dependable and efficient. For the case of mechatronic systems, dependability means reliability, availability and safety. Efficiency comprises factors like power consumption, code-size, run-time, weight and cost. Additionally, an embedded system may have a user interface that remains constant during the product's life time. Knowledge about the process under control and the user interface can be used to minimise resources and to maximise robustness.

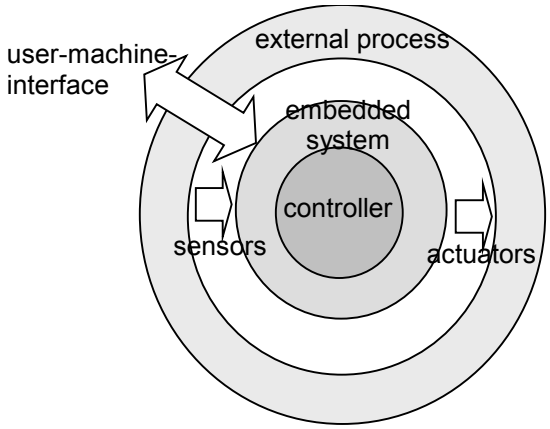

Figure 1 Basic structure of an embedded system

Figure 1 illustrates the basic structure of an embedded system comprising *external process*, *sensors*, *actuators*, and *controller*. *The external process* which is of physical, mechanical, or electrical nature. *Sensors* that provide information about the state of the *external process* by means of *monitoring events*. They are communicated to the *controller*. For the *controller*, they represent input events. *The controller* which must react to each received input event. Depending on that the states of the *external process* are deduced. *Actuators* which receive output results from the *controller.* They are communicated to the *external process* by means of *controlling events*.

2.2 Software development for embedded systems

The embedded system's software design cycle consists of specification, validation, implementation, testing, and test result analysis. Software

reliability is commonly reached by means of exhaustive testing and test result diagnosis. The latter consists of two items: test verdict (at least fail or pass) and error diagnostic in the fail case. Test verdict derivation checks the validity of the observable behaviour of an embedded system implementation, called implementation under test (IUT). This is done by assigning verdicts to the all test outcomes in form of *pass*, *fail* or *inconclusive* according to the TTCN notation [10]. In EDiagnostiquor, test verdict assignment is based on test cases and the conformance requirements described by formal specification. Figure 2 gives an overview of the main components of the diagnostic system EDiagnostiquor regarding the testing phase.

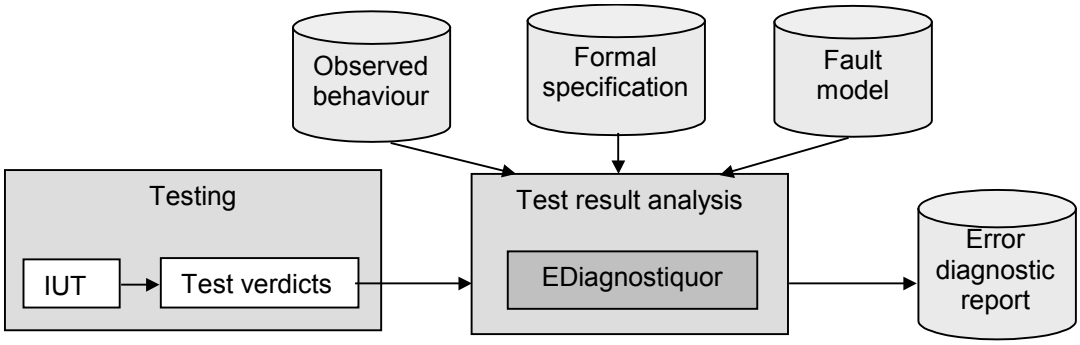

Figure 2 Overview of the diagnostic system EDiagnostiquor

Formal specification: We use extended finite state machines (EFSMs) to describe embedded systems. This model is common for communicating system, e.g. and their protocols. It permits to model both, control flow by the FSM and data flow by variables. EFSMs are the basis of languages like SDL and Estelle[2] that are used for formal description techniques (FDTs).

An *extended finite state machine* (EFSM) is 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) x … x *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, $T \subseteq S$ x C x I x O x S x 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.

[2] SDL uses the 'block' concept whereas Estelle 'module'

EFSMs are described as processes by means of appropriate syntactical and functional constructions of the formal description techniques SDL and Estelle. In FDTs a system is considered as a collection of concurrently-running blocks or modules which communicate with each other through signals conveyed via channels. And a block or module is a collection of concurrently-running processes, i.e. EFSMs.

For an embedded system, a specification consists of the behaviour description of its environment and its controller by means of EFSMs. This consists of a set of modules where each module describing a given function is modelled as a one or many EFSMs (Figure 3). These modules interact with each other via broadcasting events. However, a sequence of operations has to be obeyed in this communication. For instance, the direct communication of a module of an actuator with a sensor is not allowed.

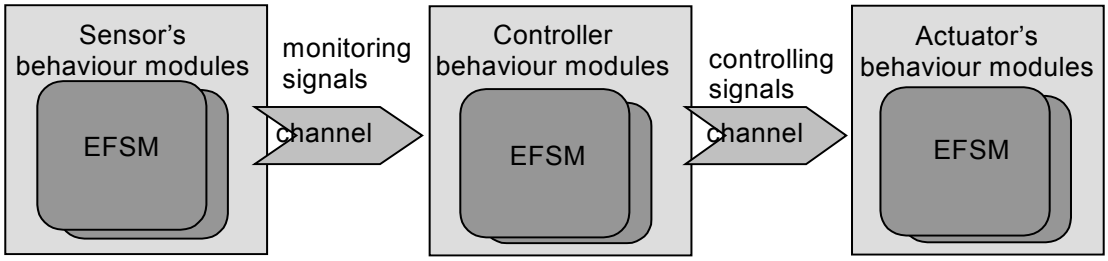

Figure 3 Embedded system based on EFSM modelling

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 recognised by all components, these events have to be declared as global variables for adjacent EFSMs. The events output from sensors represent 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/of state for an electrical unit, the speed of a mobile object such as a car, etc.) the corresponding EFSMs 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.

It is easy to map a EFSM specification on the behavioural part (transition part) of an SDL block or 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
```

The *when clause* corresponds to input events in EFSM, *from* to edge state, *provided* to predicate, *delay* is a timing special input event, *to* to the tail state and *output* to output event.

Observed behaviour: The observed behaviour of an implementation, IUT (implementation under test), constitutes a set of interaction sequences noted $\sigma_i \in (I \times O)^*$ that are obtained by running test cases on the implementation (IUT). *I* and *O* are the set of inputs and the set of outputs, respectively. During the test run an observation is made which consists of a log trace of occurring interactions. The run of a test case on an implementation is called test case execution which can be formalised by a function: *exec:* $\mathbf{T} \times \mathbf{I} \rightarrow \mathbf{O}$. **T** is a set of test cases, **I** is a set of implementations, and $\mathbf{O} \subseteq (I \times O)^*$ is a set of observations. This assumes a given test architecture which is not discussed here. When a test execution leads to an observation σ_i, the test result is defined by a verdict assignment that can be formalised as a function: *verdict*: $\mathbf{T} \times \mathbf{O} \rightarrow$ **{***pass***,** *fail,* *inconclusive***}**.

Test generation: For the test case generation, obtaining the set **T** above, different methods based on FSMs can be applied: transition tour method, distinguishing sequences' method, characterising sequences' method and unique input/output sequences' method [6] [7]. They have all a common basic idea. A test sequence is preferably a short sequence of consecutive transitions that contains every transition of the FSM at least and allows to check whether every transition is implemented as defined. For example, the principle of the transition tour method consists of the determination of a sequence, called transition tour, that must include each transition at least one time. An interaction sequence, i.e. test case, constitutes a subtour of a transition tour. 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) or be carried out by means of distinguishing sequences (Checking experiments method), characterising sequences (W-method), or

unique input/output sequences (UIO methods). We consider here a black-box testing (conformance testing), i.e. the states of the implementation under test are not visible for the environment. Thus, only input/output interactions are observable at the environment interface. This explains why we consider here that the set of observations **O** above consists only of input and output sequences and not explicitly including states.

Fault models: Due to the large number and complexity of embedded systems failures, a practical approach for testing should avoid working directly with those failures that may be of physical or software nature. Indeed, in most cases, one is mainly concerned by 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 for resolving this problem is by using a fault model 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 state transition-based system fault models may potentially include following errors classes:

- 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.
- 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.
- Transfer errors with additional states class: here it is supposed there is more states as specified with possible transfer faults.
- Missing states errors class: In this case there is less states as specified. This is usually due to non-deterministic behaviours and/or incompleteness.

In our case, the fault model is based on EFSMs. An EFSM is described by a module definition in an Estelle normal form specification. This normal form allows the interpretation of modules in terms of an EFSM. Specifications may be transformed into normal form specifications by means of syntax-directed transformation rules. Main characteristics of a normal form specification are given in [8]. A transition-declaration in an Estelle normal form specification has the following form:

```
trans from current to next when input provided predicate
begin assignment; output end;
```

where

- *current* and *next* are state-identifiers specifying the start state and the end state of a transition;
- *input* is an interaction (possibly associated with interaction parameters) specifying the input of a transition;
- *predicate(v_1, …, v_k)* is a Boolean expression of the variables v_1, …, v_k specifying the enabling condition for a transition;
- *assignment* is a (possibly empty) sequence of assignment-statements specifying that variables are set to new values during execution of a transition;
- *output* is a sequence of output-statements specifying the output caused by execution of a transition.

Note that to the *when*-clause above, a *delay*-clause could be associated to indicate a time-out event as input. With the help of *delay*-clause in Estelle, a requirement on the time aspect may be specified, i.e. the system can be found in a given state only during a given time period, after that a transition is automatically carried out. The interactions, i.e. an input or out put, of a transition are noted *ip.interaction*. *ip* indicates the interaction point, i.e. an abstract interface of a module, to which the given interaction *interaction* is associated assuming FIFO-queues [2]. For embedded systems, a module models a function of one of the embedded system components, for instance, *controller*. An appropriate notation is adopted using the interaction points to identify the component destination or origin of the interaction.

According to the back box testing (for conformance) as indicated above, we consider a fault model with two types of errors: *erroneous transitions*, and *additional and missing transitions*. These are considered as error classes and explained bellow.

Erroneous transitions: This error type includes the following error subclasses:

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

Missing and additional transitions: This type of errors refers to interactions that are not present according to the specification, or that an interaction foreseen by the specification is not present in the observed behaviour. The diagnostic proceeds in a similar manner as in the case of erroneous transitions as shown in section 3.

Knowledge representation: We consider that the diagnostic problem consists of three items: the problem characteristics (symptoms), the solutions (diagnostics), and the knowledge that relates symptoms to diagnostics. These correspond in our case to the observed behaviour, the fault model, and the diagnostic to be generated by taking into account the specification. All these components can be perfectly described with the help of the knowledge representation formalisms, i.e. the object concept, the inference rules and the related mechanisms (backward or forward chaining for inferencing). In particular, the diagnostic knowledge is based on rules of the form:

```
if a diagnostic D causes the symptom S
and S is observed
then D is a possible explanation of S
```

The set of these inference rules builds the rule base. The whole knowledge base of the diagnostic system EDiagnostiquor is composed from this rule base and an object base which consists of other facts and data needed for inferencing and/or results from as specification, fault models, observed behaviour, and the diagnosis report which explains the error reasons.

3 Example

In this section, we demonstrate the function of the diagnostic system EDiagnostiquor on an example (Figure 4). The example (pressure control) represents the controller behaviour of an embedded system modelled by EFSM which consists of 4 states: l_pres (low pressure), l_pow (low power), h_pres (high pressure), and h_pow (high power). These correspond to the reaction of the controller by commanding an appropriate actuator in function of the pressure measured by sensors.

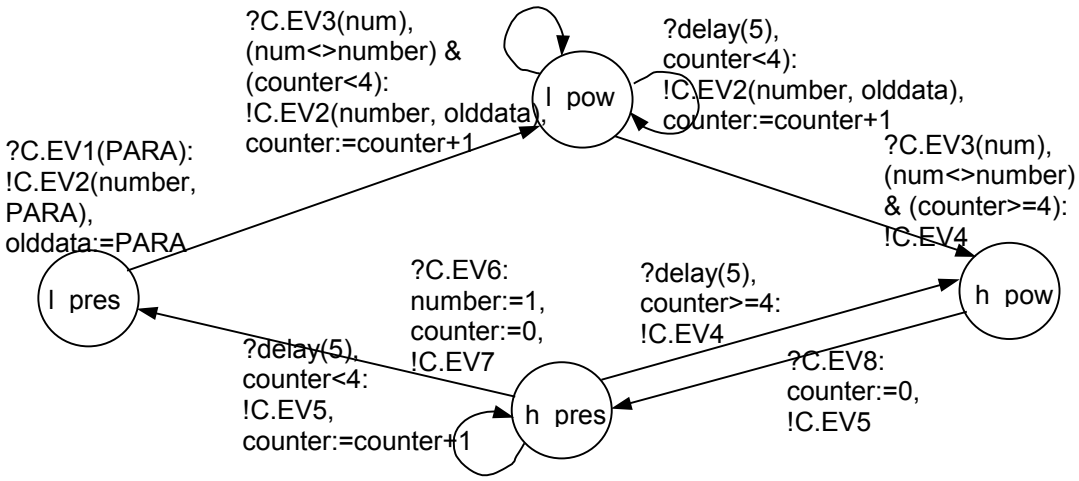

Figure 4 Specification example

Let consider an observed behaviour of the IUT that fails a given test case run. The diagnostic system performs a search of the possible interaction sequences of the specification. It finds then the possibilities of interaction errors that are sufficient to explain the observed interaction sequence. If the analysis reaches the end of the observed behaviour, a diagnostic results reflecting interaction sequences of the specification. Interaction sequences are algorithmically derived by means of the transition tour method. The number of errors to be diagnosed in an interaction sequence is theoretically optional. However, the experience shows that for more efficiency of the diagnostic this is usually limited to a small number of errors.

In the example specification of Figure 3, the interaction parameters *num* and *number* are integer and *dom*(*num*)= *dom*(*number*)= [0,1]. Let be given the observed behaviour σ=*exec*(*tc*, *I*) and suppose that *verdict*(*tc*, *I*) = "*fail*", where:

σ="!C.EV8; ?C.EV5; !C.EV6; ?C.EV8; !C.EV1(1); ?C.EV2(1,1)"

The symbols ! and ? denotes an input symbol and output symbol, respectively, and ';' the interaction sequence symbol. Note we have inverted the symbols ! and ? regarding the specification because the corresponding interactions are now considered from the point of view of the tester. For this example, EDiagnostiquor will explore the different interaction sequences (subtours) and encounters an inconsistency between the specification and the analysed behaviour. In this case, the anomaly in the implementation under test I is explained by one wrong interaction at the fourth place. Nor more inconsistency can be found and the diagnostic message is given as follows:

```
*** interaction 4 is erroneous ***
*** interaction 4 must be ?C.EV7 ***
```

This type of errors belongs to the subclass “output error” of the class “erroneous transitions”. For this example, it seems intuitively clear that the error is located at the fourth interaction. However, in more complicated situations in the case of a large number of errors, it is difficult to manually state what the errors are and to diagnose them. It is difficult to provide a number of possible diagnostic messages, each indicating how the observed interaction sequence could be obtained w.r.t. the specification by a given number of changes.

We use again the example of Figure 4 and consider the erroneous interaction sequences bellow. We assume that for each observation σ_i, with ($1 \leq i \leq 5$): $exec(tc_i, I) = \sigma_i$ and $verdict(tc_i, \sigma_i)$=”fail”. The first phase of the error diagnostic indicates that all these interaction sequences are invalid w.r.t. the specification as follows:

```
1. σ1=″!C.EV8; ?A.EV5; !C.EV6; ?C.EV7; !C.EV1(3); ?C.EV2(1,1)″
*** interaction 2: erroneous interaction point ***

2. σ2=″!C.EV8; ?C.EV5; !C.EV6; ?C.EV7; !C.EV1(4); ?C.EV2(4); !C.EV3(0);
    !C.EV2(-2,8)″
*** interaction 6: erroneous parameter ***
*** interaction 8: erroneous parameter ***

3. σ3=″!C.EV8; ?C.EV5; !delay(5); ?C.EV5; !delay(5); ?C.EV5; !C.EV6;
   ?C.EV7; !C.EV1(9); ?C.EV2(1,9); !delay(5); ?C.EV2(1,9); !delay(5);
   ?C.EV2(1,9); !delay(5); ?C.EV2(1,9); !delay(5); ?C.EV2(1,9);
   !C.EV3(1); ?C.EV2(1,9); !C.EV3(0); !C.EV4″
*** interaction 20: erroneous predicate ***

4. σ4=″!C.EV8; ?C.EV5; !delay(5); ?timeout(5); !C.EV6; ?C.EV7″
*** interaction 4: occurring of time out ***

5. σ5=″!C.EV8; ?C.EV5; !delay(5); ?C.EV5; !C.EV6; !C.EV1(2);?C.EV2(1,2);
   !delay(5); ?C.EV2(1,2); !delay(5); ?C.EV2(1,2); !delay(5);
   ?C.EV2(1,2); !delay(5); ?C.EV2(1,2); !C.(0); ?C.EV4; ?C.EV9″
*** missing interaction in position 6 ***
*** erroneous interaction in position 18 ***
```

The second phase of the diagnostic is then initiated to determine the reasons of non-conformance of the IUT. The system generates the following diagnostics:

```
1. *** interaction 2: erroneous interaction point --- interaction 2 must
   be ?C.EV5 ***

2. Diagnostic 1:
*** interaction 6: erroneous interaction parameter --- missing
interaction parameter ***
   Diagnostic 2:
*** interaction 8: erroneous interaction parameter --- first parameter
must be included between 0 and 1, second parameter must be 4 ***

3. *** interaction 8: predicate evaluated to false ***

4. *** interaction 5: expected interaction must occur within time limit 5
   ***
```

```
5. Diagnostic 1: *** missing interaction in position 6, it must be ?C.EV7
   ***
   Diagnostic 2: ***interaction in position 18 must be absent ***
```

4 Conclusion

Conventionally, test derivation and diagnostic are most time consuming in the design cycle of reliable embedded systems because of their complexity and the huge number of potential failures. Special knowledge is required to explain found errors in faulty implementations. In our case, error diagnostic is based on EFSM specifications which are equivalent to blocks of Estelle or SDL code and specify control and data flow. The diagnostic system EDiagnostiquor proposed here handles important error classes related to these two aspects.

In a future work, we plan to develop an appropriate test architecture for embedded systems and to apply the principle of EDiagnostiquor to a real-life embedded system of the automotive. For more flexibility of the diagnostic system, we will also try to involve other specifications formalisms.

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, Lisbon, 1998.

[4] M. Mendler, G. Luettgen. Statecharts: From Visual Syntax to Model-Theoretic Semantics. In K. Bauknecht et al. (eds.), WIDFST, 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 et al. (eds.), Protocol Specification, Testing, and Verification, New Jersey, 1988.

[7] S. Fujiwara et al. 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] ISO/IEC 9646-3, Information technology – Open systems interconnection – Conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation (TTCN), 1998.