Automatic verification of embedded system based on EFSM

Jinjiang Liu*, Jingjing Liang

School of Computer and Information Technology, Nanyang Normal University, Nanyang 473061, Henan Province, China

Received 6 July 2014, www.cmnt.lv

Abstract

To ensure the correctness of embedded system, automation of test case generation is necessary in industrial. This paper present a technique for specifying coverage criteria and a method for generating test suites for embedded systems whose behaviours is depend on its interactive environment. The embedded system under test can be described as extended finite state machines (EFSM) and the coverage criteria can be specified as monitor automata with parameters, which monitor and accept traces that cover a given test criterion of an EFSM. The flexibility of the technique is demonstrated by specifying a number of well-known coverage criteria based on control- and data-flow information using observer automata with parameters. We also develop a method for generating test cases from coverage criteria specified as observers. It is based on transforming a given observer automata into a bitvector analysis problem that can be efficiently implemented as an extension to an existing state-space exploration such as, e.g. SPIN or Uppaal.

Keywords: EFSM (Extended Finite State Machine), embedded system, test case generation, model-based testing

1 Introduction

A Model based test case generation has in recent years been developed as a prominent technique in testing of reactive software systems. A model serves both the purpose of specifying how the system should respond to inputs from its environment, and of guiding the selection of test cases, e.g., using suitable coverage criteria. Typical notations for such models are state machines in some form, often extended with data variables. Test cases can be selected as individual "executions" of the model, checking that the outputs from the system under test (SUT) conform to those specified by the mode.

In this paper, we present a technique for specifying coverage criteria in a simple and flexible manner, and a method for generating test cases according to such coverage criteria. The technique fits well as an extension of a state space exploration tool, such as, e.g., SPIN [2] or Uppaal [4], which performs enumerative or symbolic state-space exploration. It can also be used to generate monitors that measure the coverage of a specific test suite by monitoring the test execution.

Most related work on test case generation from models of reactive systems employs some rather specific selection of coverage criteria. Explicitly given test purposes have been considered, both enumerative [5-7] and symbolic [9]. Test purposes in these works can in some sense be regarded as coverage observers, but are not used to specify more generic coverage criteria and do not make us of parameterization, as in our work.

Some approaches present more flexible techniques for specifying a variety of coverage criteria. Hong et al [11-14] describe how flow-based coverage criteria can be expressed in temporal logic. A particular coverage item is expressed in CTL, and a model checker generates a trace, which covers the coverage item. In our approach, we use monitors instead of temporal logic, which avoids some of the limitations of temporal logic [15]. Our technique using monitors whit parameters can let one pass of a state-space exploration tool generate a test suite that covers a large set of coverage items, whereas the above approaches invoke a run of a model checker for each coverage item.

The remainder of the paper is structured as follows. We present EFSM in the next section, and monitors in Section 3. In Section 4-5, we show how our definitions of coverage can be used for test case generation, and report on a partial implementation of the technique. Section 6 concludes the paper

2 Extended finite state machine

In this section, it is assumed that a System Under Test (SUT) interacts with its environment through events. Whenever the SUT receives an input event, it responds by performing some local computation and emitting an output event. To a given SUT, we associate a set A of event types, each with a fixed arity. An event is a term of form *a(d1, . . . , dk)* where a is an event type of arity *k* and *d1, . . . , dk* are the parameters of the event. The set *A* of event types is partitioned into *input event types* and *output event types*. A trace is a finite sequence

$$a_1(\bar{d}_1)/b_1(\bar{d}_1') \quad a_2(\bar{d}_2)/b_2(\bar{d}_2') \ \ldots \ a_n(\bar{d}_n)/b_n(\bar{d}_n')$$

of *input/output event* pairs. Intuitively, the trace represents a behaviour where the SUT, starting from its initial state, receives the input event $a_1(\bar{d}_1)$ and responds with the

*Corresponding author's e-mail: nytcc@sina.com

output event $b_1(\bar{d}'_1)$. Thereafter, it receives the input event $a_2(\bar{d}_2)$ and so on. An input sequence is a finite sequence of input events.

Assume a set A_I of input event types, and a set A_O of output event types. An Extended Finite State Machine (EFSM) over (A_I, A_O) is a tuple $< L, L_0, \bar{v}, E >$ where:

1) L is a finite set of locations (control states).
2) $l_0 \in L$ is the initial location.
3) $\bar{v}$ is a finite set of state variables.
4) E is a finite set of edges, each of which is of form:

$$e : l \xrightarrow{a(\bar{w}), g \to \bar{u} := \overline{\exp r} / b(\overline{\exp r'})} l',$$

where e is the name of the edge, l is the source location, and l_0 is the target location. $a(\bar{w})$ is an input event type, and $\bar{w}$ is a tuple of formal parameters of event a, g is a guard, $u := \overline{\exp r}$ is an assignment of new values to a subset $\bar{u} \subseteq \bar{v}$ of the state variables, and $b(\overline{\exp r'})$ is an expression which evaluates to an output event.

Intuitively, an edge of the above form denotes that whenever the EFSM is in location l and receives an event of form $a(\bar{w})$, then, provided that the guard g is satisfied, it can perform a computation step in which it updates its state variables by $u := \overline{\exp r}$, emits the output event $b(\overline{\exp r'})$ and moves to location l'. We require the EFSM to be deterministic, i.e., that for any two edges with the same source location l and parameterized input event $a(\bar{w})$, the corresponding guards are inconsistent.

A system state is a tuple $< l, \sigma >$ where l is a location, and σ is a mapping from $\bar{v}$ to values. We can extend σ to a partial mapping from expressions over $\bar{v}$ in the standard way. The initial system state is the tuple $< l_0, \sigma_0 >$, where l_0 is the initial location, and σ_0 gives a default value to each state variable. A computation step is of the form $< l, \sigma > \xrightarrow{a(\bar{d}')/b(\bar{d}')} < l', \sigma' >$ consist of system state $< l, \sigma >$ and $< l', \sigma' >$, an input event $a(\bar{d}')$ and an output event $b(\bar{d}')$.Such that there is an edge of the (above) form $l \xrightarrow{a(\bar{w}), g \to \bar{u} := \overline{\exp r} / b(\overline{\exp r'})} l'$, for which $\sigma(g[\bar{d} / \bar{w}])$ is true, $\sigma' = \sigma[u \to \sigma(\overline{\exp r}[\bar{d} / \bar{w}])]$ and $\bar{d}' = \sigma(\overline{\exp r'}[\bar{d} / \bar{w}])$. A run of the EFSM over a trace $a_1(\bar{d}_1) / b_1(\bar{d}'_1) \ \ldots \ a_n(\bar{d}_n) / b_n(\bar{d}'_n)$ is a sequence of computation steps:

$$< l_0, \sigma_0 > \xrightarrow{a_1(\bar{d}_1)/b_1(\bar{d}'_1)} < l_1, \sigma_1 > \xrightarrow{a_2(\bar{d}_2)/b_2(\bar{d}'_2)} \cdots \xrightarrow{a_n(\bar{d}_n)/b_n(\bar{d}'_n)} < l_n, \sigma_n >,$$

labeled by the input-output event pairs of the trace.

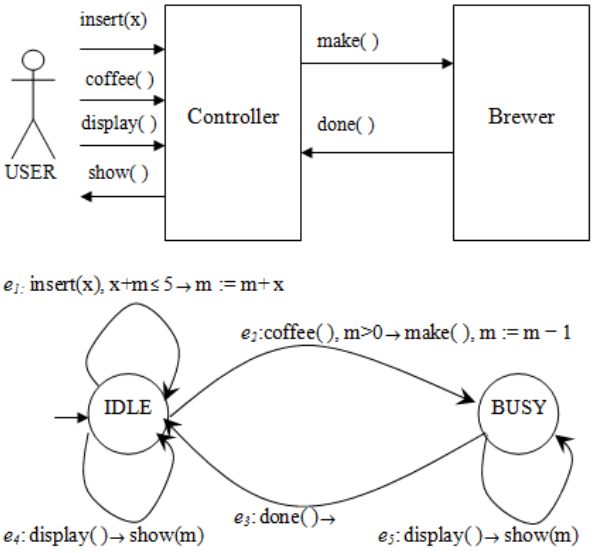

FIGURE 1 EFSM Model of the controller of a simple coffee machine

Example 1: an EFSM specifying for the simple coffee controller: Figure 1 demonstrates an EFSM specifying the behaviour of the controller of a simple coffee machine which interacts with a user and a brewer unit is shown. The controller has L = {*IDLE, BUSY*}, l_0 = *IDLE*, $\bar{v}$ = {m}, A_I = {*insert, coffee, display, done*}, A_O = {*show, make*}, and E = {e_1, e_2, e_3, e_4, e_5}. The parameter x and the variable m take values that are integers in the range [0 ... 5].

3 Monitors

In this section, we present how to use observers to specify coverage criteria for test generation or test monitoring.

As a very simple example, the coverage item "visit location l of the EFSM" can be represented by an monitor with one initial state, and one accepting location, named loc(l), which is entered when the EFSM enters location l. The coverage criterion "visit all locations of the EFSM" can be represented by a parameterized observer with one initial state, and one parameterized accepting location, named loc(L), where L is a parameter that ranges over locations in the EFSM. For each value l of L, the location loc(l) is entered when the EFSM enters location l.

Formally, an *monitor* is a tuple (Q, q_0,Q_f,B) where:

1) Q is a finite set of observer locations
2) q_0 is the initial observer location.
3) $O_f \in Q$ is a set of accepting locations, whose names are the corresponding coverage items.
4) B is a set of edges, each of form $q \xrightarrow{b} q'$, where b is a predicate that can depend on the input event received by the SUT, the mapping from state variables of EFSM to their values after performing the current computation step, and the edge in the EFSM that is executed in response to the current input event.

3.1 OBSERVER PREDICATE

In the following we introduce a more specific syntax for the predicates b occurring on observer edges. The predicates will use a set of predefined *match variables* that are given values at the occurrence of:

1) an event $a(\bar{d})$,

2) an edge $e: l \xrightarrow{a(\bar{w}), g \to \bar{u} := \overline{\exp r}} l'$ of the EFSM, traversed in response to $a(\bar{d})$,

3) the computation step $< l, \sigma > \xrightarrow{a(\bar{d})} < l', \sigma' >$ generated in response to $a(\bar{d})$.

For a traversed EFSM edge we use the following match variables (with associated meaning):

1) *event type* is the event type a of the occurring event;
2) *event-pars* is the list $\bar{d}$ of parameters of the event;
3) *edge* is the name *e*;
4) *target_loc* is the target location *l'*;
5) *guard* is the guard expression *g*;
6) *assignments* is the set *u* := $\overline{expr}$ of assignments;
7) *target_val* is the function from EFSM state variables to values.

To be able to express more interesting properties we also introduce a set of operations that can be used together with the match variables:

3.2 FUNCTION DEFINITION OF PREDICATE

With the match variables and operations above we define new functions that can be used as tests in the observer. In this paper, we shall make use of:

1) *def (v)* which is true *iff* the variable *v* is defined by the transition in the EFSM. This can be expressed as: $v \in map(lhs, assignments)$.

2) *use (v)* which is true *iff* the variable *v* is used (in a guard or assignment) by the transition in the EFSM. This can be expressed as: $v \in vars(map(rhs, assignments)) \vee v \in vars(guard)$.

3) *da* (v_1, v_2) which is true *iff* the variable v_1 is on the right hand side and variable v_2 is on the left hand side of the same assignment in the EFSM specification. The function can intuitively be understood to be true if v_1 directly affects v_2. This can be expressed as: $map\ (affect\ (v_1, v_2), assignments) \neq \varnothing$.

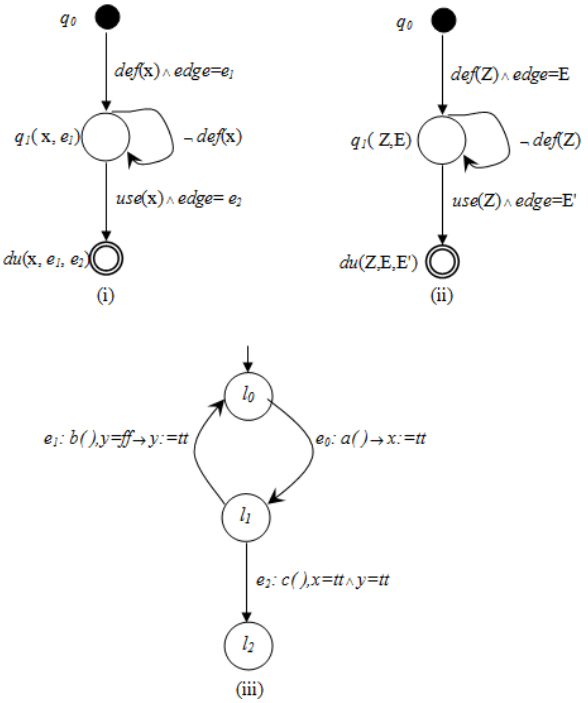

FIGURE 2 Examples of (i) a non-parameterized monitor (ii) a parameterized monitor, and (iii) a simple EFSM

Example 2: example for non-parameterized and parameterized monitor: apparently, the monitor in Figure 2(i) is non-parameterized which specifies definition-use pair coverage for a specific variable m, and specific edges e_1 and e_2. Figure 2(ii) shows a corresponding (parameterized) monitor that specifies definition-use pair coverage for any EFSM variable *Z*, and EFSM edges *E* and *E*'. This is done by parameterizing the location q1 with any variable and any edge, and the accepting location du with any variable and any two edges. The edges are parameterized in a similar way. For example, there is one observer edge from location $q_1(z, e)$ to location *du* (*z*, *e*, *e*') for each EFSM variable *z*, and each pair *e*, *e*' of EFSM edges.

4 Monitoring test generation by superposing a monitor on an EFSM

In test case generation or when monitoring test execution of a SUT, an observer observes the events of the SUT, and the computation steps of the EFSM. Reached accepting locations correspond to covered coverage items. We formally define the execution of an observer in terms of a composition between an EFSM and an observer, which has the form of a superposition of the observer onto the EFSM. Each state of this superposition consists of a state of the EFSM, together with a set of currently occupied observer locations.

Say that a predicate *b* on a monitor edge is satisfied by a computation step $< l, \sigma > \xrightarrow{a(\bar{d})} < l', \sigma' >$ of an EFSM, denoted $< l, \sigma > \xrightarrow{a(\bar{d})} < l', \sigma' > \models b$ if *b* holds for the

event $a(\bar{d})$, the computation step $<l,\sigma> \xrightarrow{a(\bar{d})} <l',\sigma'>$ and the edge $e: l \xrightarrow{a(\bar{w}),g \to \bar{u}:=\overline{\exp r}} l'$ from which the computation step is derived.

Formally, the superposition of an monitor (Q, q_0, Q_f,B) onto an EFSM < L,L_0, $\bar{v}$,E >is defined as follows.

1) *States* are of the form $<<l,\sigma>||Q>$, where $<l,\sigma>$ is a state of the EFSM and Q is a set of locations of the monitor.

2) The *initial state* is the tuple $<<l_0,\sigma_0>||\{q_0\}>$, where $<l_0,\sigma_0>$ is the initial state of the EFSM and q_0 is the initial location of the monitor.

3) A *computation step* is a triple $<<l,\sigma>||Q> \xrightarrow{a(\bar{d})} <<l',\sigma'>||Q'>$ such that $<l,\sigma> \xrightarrow{a(\bar{d})} <l',\sigma'>$ and

$$Q' = \{q' \mid q \xrightarrow{b} q' \text{and } q \in Q \text{ and } <l,\sigma> \xrightarrow{a(\bar{d})} <l',\sigma'> \models b\}.$$

4) A state $<<l,\sigma>||Q>$ of the superposition *covers the coverage item* represented by the location $q_f \in Q_f$ if $q_f \in Q$.

$$<<l_0,\{x=tt,y=tt\}>|| \{q_0,q_1(x,e_0),q_1(y,e_1)\}> \xrightarrow{a()} <<l_1,\{x=tt,y=tt\}>||\{q_0,q_1(x,e_0),q_1(y,e_1)\}>$$

5 Experimental results for three crucial embedded systems

5.1 ALGORITHM

At test case generation, we use the superposition of an observer onto an EFSM, and views the test case generation problem as a search exploration problem. To cover a coverage item q_f is then the problem of finding a trace $tr = <<l_0,\sigma_0>||\{q_0\}> \xrightarrow{a(\bar{d}) \cdots a'(\bar{d}')} <<l,\sigma>||Q>$, such that $q_f \in Q$.

An abstract algorithm to compute test case is shown as below:

1) $Pass := \varnothing$, Max:=0, $w_{max} := w_0$
2) $Wait$:={ $<<s_0||\{q_0\}>,w_0>$ }
3) **while** Wait $\neq \varnothing$ **do**
4) select $<<s||Q>,w>$ from $Wait$
5) **if** $|q_f \cap Q| > Max$ **then**
6) $w_{max} := w$, Max:= $|q_f \cap Q|$
7) **if** for all $<s||Q'>$ in $Pass$: $Q \subsetneq Q'$ **then**
8) add $<s||Q>$ to $Pass$
9) for all $<s''||Q''>$
10) such that $<s||Q> \xrightarrow{a} <s''||Q''>$:
11) add $<<s''||Q''>,wa>$ to $Wait$
12) **return** w_{max} and Max

To improve the presentation, we use s to denote a system of the form $<l,\sigma>$ and s_0 to denote the initial system state $<l_0,\sigma_0>$ and a to denote an input action $a(\bar{d})$. The algorithm computes the maximum number of coverage items that can be visited (*Max*), and returns a trace with maximum coverage (w_{max}). The two main data structures *Wait* and *Pass* are used to keep track of the states waiting to be explored, and the states already explored, respectively.

Initially, the set of already explored states is empty and the only state waiting to be explored is the extended state $<<s_0||\{q_0\}>,w_0>$, where w_0 is the empty trace (in Line2). The algorithm then repeatedly examines extended states from Wait (in Line 3). If a state <s||Q> found in *Wait* is included in a state <s||Q> in *Pass*, then obviously <s||Q> does not need to be further examined(in Line 7-8). If not, all successor states reachable from <s||Q> in one computation step are put on Wait, with their traces extended with the input action of the computation step from which they are generated (in Line 9-11). The state <s||Q> is saved in *Pass*. The algorithm terminates when *Wait* is empty.

The variables w_{max} and *Max* are initially set to the empty trace and 0, respectively (in Line 1). They are updated whenever an extended state is found in *Wait* which covers a higher number of coverage items than the current value of *Max* (in Line 5-6). Throughout the execution of the algorithm, the value of *Max* is the maximum number of coverage items that have been covered by a single trace, and w_{max} is one such trace. When the algorithm terminates (in Line 12), the two values *Max* and w_{max} are returned.

5.2 BITVECTOR IMPLEMENTATION

In order to efficiently represent and manipulate the set Q of observer locations we shall use bitvector analysis [15]. Let the set Q be represented by a bitvector where each bit represents an observer location q'. Then each bit is updated by the following function:

$$f_{q'}(q') = \bigvee_{<b,q> \in in(q')} q \wedge b ,$$

where $in(q') = \{<b,q> \mid q \xrightarrow{b} q' \in B\}$ is the set of pairs of predicates b and source locations q of the edges ingoing to the location q'. That is, given a state of the superposition $<<l,q>||Q>$ and an EFSM transition $<l,\sigma> \xrightarrow{a(\bar{d})} <l',\sigma'>$ the bit representing q' is set to 1 if there is an monitor edge $q \xrightarrow{b} q' \in B$, such that $q \in Q$ and $<l,\sigma> \xrightarrow{a(\bar{d})} <l',\sigma'> \models b$. Otherwise the bit representing q' is set to 0. It should be obvious that this

corresponds precisely to the semantics of an monitor superposed onto an EFSM, described in Section 4.2

Example 4: Interpreting monitor states set Q into bitvector: when the monitor in Figure 2(ii) is superposed onto the EFSM in Figure 2(iii), we have: $E = E' = E = \{e_0, e_1, e_2\}$ and $Z = \bar{v} = \{x, y\}$. Thus, we have that:

$$Q = \{q_0\} \cup \{q_1(z, e_a) \mid z \in \bar{v} \wedge e_a \in E\} \cup \{du(z, e_a, e_b) \mid z \in \bar{v} \wedge e_a \in E \wedge e_b \in E\}.$$

Any enumeration of the set can be used as index in the bitvector. As the observer has three locations with parameters we get three types of bitvector functions:

$$f_{q_0}(q_0) = q_0 \wedge tt, \quad (1)$$

$$f_{q_1(v_i, e_j)}(q_1(v_i, e_j)) = (q_0 \wedge def(v_i) \wedge edge = e_j) \vee (q_1(v_i, e_j) \wedge \neg def(v_i)), \quad (2)$$

$$f_{du(v_i, e_j, e_k)}(du(v_i, e_j, e_k)) = (q_1(v_i, e_j) \wedge use(v_i) \wedge edge = e_k) \vee (du(v_i, e_j, e_k) \wedge tt). \quad (3)$$

There is one function of Equation (1), six of Equation (2), and 18 of Equation (3). Note that Equation (1) is always true and that Equation (3) will remain true once it becomes true, due to implicit self-loops in these locations.

6 Conclusion

This paper has presented a technique for testing the remote environment control systems. Our technique have shown to be a flexible tool in model checking and run-time monitoring, and by this paper we have shown that they are a versatile tool for specifying coverage criteria for test case generation and test monitoring.

In particular, the parameterization mechanism, as used in this paper, allows a succinct specification of several standard generic coverage criteria. In this way, test case generation can be transformed into a reachability problem, which can be general used in verification of environmental control systems.

References

[1] Hu C, Zhu L 2010 The analysis and the evaluation of complicated network software *LNCS* **13**(10) 1-5

[2] Yunfeng Wang, Hongde Xia, Raomei Yan 2008 The analysis of the social network and the study of the application cases of NetDraw *Modern education technology* **18**(4) 85-89

[3] Pothen A, Simon H, Liou K P 1990 Petitioning sparse matrices with eigenvectors of graphs S*IAM Journal on Matrix Analysis and Applications* **11** 430-6

[4] Grivan M, Newman M E J 2001 Community structure in social and biological networks *Proc Natl Acad Sci* **99**(12) 7821-6

[5] Newman M E J, Grivan M 2004 Finding and evaluating community structure in networks *Physical Review E* **39**(10) 69-84

[6] Toyoda M, Kitsuregawa M 2003 Extracting evolution of web communities from a series of web archives *Proceedings of the fourteenth ACM conference on Hypenext and hypermedi*a **101** 78-87

[7] Palla G, Derényi I, Vicsek T 2007 The Critical Point of *k-groups* Percolation in the Erdös-Rényi Graph *Journal of Statistical Physics* **128**(1) 219-27

[8] Palla G, Vicsek T, Barabási A-L 2007 Community dynamics in social networks *Noise and Stochastics in Complex Systems and Finance* **6601**(3) 273-87

[9] Xu C, Zhang Y, Yang D 2011 Ontology based Image Semantics Recognition using Description Logics *IJACT: International Journal of Advancements in Computing Technology* **3**(10) 1-8

[10] Ju C, Wei J 2012 Research on Multi-interest Profile Based on Resource Clustering *JCIT: Journal of Convergence Information Technology* **7**(21) 582-90

[11] Gargantini A, Heitmeyer C 1999 Using Model Checking to Generate Tests From Requirements Specifications *In Software Engineering – ESEC/FSE'99: 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering* **1687**(1) 146-62

[12] Kim J-M, Porter A 2009 A history-based test prioritization technique for regression testing in resource constrained environments *In ICSE '09: Proceedings of the 31th International Conference on Software Engineering* **139**(3) 119-29

[13] Rothermel G, Untch R H, Chu C, Harrold M J 2009 Test case prioritization: An empirical study *Proceedings of the IEEE International Conference on Software Maintenance* **168**(11) 179-83

[14] Srikanth H, Williams L 2005 On the economics of requirements-based test case prioritization *Proceedings of the 7th international workshop on Economics-driven software engineering research* **153**(1) 1-3

Authors

Jinjiang Liu, born in October, 1974, Nanyang County, Henan Province, P.R. China

Current position, grades: associate professor in School of Computer and Information Technology of Nanyang Normal University, China.
University studies: MSc in Computer Applications at Wuhan University of Science & Technology in China.
Scientific interests: computer modeling, data mining.
Publications: more than 6 papers.
Experience: teaching experience of 15 years, 6 scientific research projects.

Jingjing Liang, born in October, 1981, Nanyang County, Henan Province, P.R. China

Current position, grades: instructor at the School of Computer & Information Technology of Nanyang Normal University, China.
University studies: BSc in University of Electronic Science and Technology of China.
Scientific interests: software engineering, formal modelling.
Publications: more than 5 papers.
Experience: teaching experience of 10 years.