Model Based Program Synthesis and Runtime Error Detection for Dependable Embedded Systems

Abstract of the PhD Thesis

Gergely Pintér

Advisor: Dr. István Majzik
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Budapest, 2007
Introduction

As computer-based systems pervasively extend to virtually all facets of life ranging from power plant control through everyday PCs to nearly invisible computers working as embedded intelligence in vehicles, health-care, or mobile devices, the dependence of the society on their correct behavior leads to growing impact of these systems’ dependability. My PhD thesis presents research achievements of over three years focusing on various aspects of software dependability. The first part of this booklet (Chp. I-4) discusses the motivation, summarizes the achievements, outlines the methods of research and enumerates the related publications according to the organization of the thesis. Chp. 5 highlights the applicability of the achievements, finally the last chapter presents the organized list of our publications.

The four parts of the thesis focus on the semantics of UML 2.0 statecharts, implementation of systems specified by statecharts, runtime error detection and automatic processing of data obtained from experimental evaluation of dependability:

- The first part introduces a formal operational semantics for UML 2.0 statecharts. Our approach provides support for the entire set of practically relevant modeling concepts, defines an unambiguous formalism for handling concepts related to state refinement, compound transition and activity structures and being an inherently pragmatic solution provides a solid foundation for various activities based on statecharts including model checking, automatic code generation and runtime error detection.

- The second part presents a method for implementation of systems specified by statecharts. The low memory consumption and processing power requirements of the approach enable its application even in resource constrained embedded systems.

- The third part introduces two methods for runtime error detection in systems specified by statecharts. The first approach defines a temporal logic language that enables the specification of various temporal correctness criteria and presents a method for the effective evaluation of these criteria during runtime – this solution is capable of detecting errors caused by design (model refinement) faults. The second solution is a runtime monitor that observes the execution of a statechart implementation and a checks whether the behavior corresponds to the statechart enabling this way the detection of implementation faults.

- The fourth part presents a data mining based method for automatic processing of data obtained from experimental evaluation of dependability. The approach provides support for identifying those infrastructural, configuration or faultload-related properties that have a primary impact on the behavior of a complex COTS system in the presence of faults.

The structure of the thesis is outlined Fig. 1: the four subjects addressed are represented by four rectangles respectively; the vertical alignment indicates conceptual interdependence, the arrows between blocks show the application of some achievements in other areas.
Figure 1: Overview of the Thesis
Chapter 1

Formal Operational Semantics for UML 2.0 Statecharts

1.1 Motivation

The first subject addressed in the thesis is related to the operational semantics of statecharts in the UML 2.0 visual modeling language. With respect to the dominating trends in modern software industry, the importance of visual modeling and the actuality of subject does not need much discussion: the straightforward graphical notation enables the construction of models of high documentation value supporting the communication with the users, helping the organization of work within the development team and providing powerful facilities for ruling the complexity. Apart from the documentation role the direct application of visual models in the development receives ever growing attention: a semantically well established model can be the basis of automatic implementation (code generation), various verifications can be performed on models to prove their correctness with respect to some dependability requirements (model checking), moreover, formal models play even the role of executable specification in runtime verification approaches.

UML is the most popular modeling language in the software industry, its most recent version, 2.0 offers sophisticated modeling facilities throughout the entire development chain (requirement analysis, design, implementation and deployment). Many professional UML modeling tools are available and most of them have already been integrated into full featured development environments supporting requirement management, development, debugging, version control, etc. The rigorously defined syntax of the language (metamodel) and the XML metadata interchange built upon it (XMI) enables interchanging models between various tools.

At present, UML is primarily applied for modeling static aspects of the software under development (use cases, data structures, package hierarchies, deployment etc.). Although the language provides various facilities for modeling dynamic aspects (activity diagrams, statecharts, communication diagrams etc.) the visual modeling of the behavior is usually less emphasized. As we will see below, the probable reason for this is the fact that the meaning of behavior models is less intuitive as the semantics of e.g., class diagrams. This unbalanced situation at the expense of behavior models should be undoubtedly corrected since the application of model-based software development approaches like automatic program synthesis, testing and runtime verification are expected to considerably reduce development costs, increase productivity and enhance the quality of the software delivered. In order to achieve this, we are obviously in the need of formally well established visual behavior models. The first three parts of the thesis focus on various aspects of modeling behavior, synthesizing programs and runtime verification; the visual notation used is the statechart formalism of the UML 2.0 modeling language.

In order to enable the application of visual models in engineering practice, naturally emerges the need for assigning unambiguous meaning to graphical models i.e., we have to define the formal semantics of these models. Similarly to all non-trivial languages, also in case of UML holds, that the pure definition of the grammar does not prevent the construction of semantically meaningless models thus various well-
formedness criteria are to be defined in order to prevent the uncontrolled usage of modeling artifacts. In contrast to the detailed definition of syntax and the sophisticated graphical notation, the formal specification of operational semantics is unfortunately a weak point of UML resulting in the emergence of many non-elaborated details and open questions:

- The standard uses OMG’s Object Constraint Language (OCL) [36] for ensuring the well-formedness of models. Unfortunately with respect to some important artifacts, the corresponding well-formedness rules are simply missing from the standard and some requirements are discussed only informally (see the corresponding appendix of the thesis).

- Focusing on statecharts it is to be highlighted that some concepts related to state refinement, hierarchical organization, compound transition structures and compound activities are defined by the standard only informally, their unambiguous formal definition is missing.

- Finally the most important deficiency of the standard is that it does not define a formal operational semantics for statecharts at all. The operation is meant to be introduced by textual discussion relying on human intuition but unfortunately in case of non-trivial models we can easily encounter such situations where the expected behavior is not clear i.e., which transitions are to be fired, what is the resulting state configuration etc.

There were many formalization efforts published in the literature aiming at the definition of a formal semantics for statechart models. Some of these approaches use some specification languages (e.g., PVS [10] or abstract state machines [2]), there are ones based graph transformation [22] and model transition systems [74] and the solution primarily inspiring our work is based on extended hierarchical automata [23, 24].

The large number of publications and the solution proposals originating from various research fields on the one hand justifies the need for defining the formal foundation of visual behavior models but on the other hand the issue can not be considered to be solved at all. Although behavior modeling is basically an engineering task whose direct and primary goal is to solve practical and realistic problems, this pragmatic point of view is frequently missing from approaches originating in the field of formal methods and applied mathematics.

Most of solutions mentioned above suffer from the lack of being fully elaborated, e.g., the paper of Blech, Glesner and Leitner [1] aims at code generation and formal verification based on statecharts, but actually the approach is restricted to plain state diagrams without hierarchy, concurrency and any kinds of pseudostates; Börger, Cavarra and Riccobene proposed a definitely elegant solution [2] based on agents specified by abstract state machines (ASM), but it does not support such vital constructs as fork and join pseudostates, branch vertices (choice and junction) and history nodes; the title of Aredo’s paper [10] suggests the definition of a formal semantics for statecharts using the PVS specification language but it actually does not provide more than the definition of some well-formedness rules; the straightforward visual solution by Kuske [22] is a nice example for the elegance of approaches based on structured graph transformations, but it is still not practical enough: statecharts are considered as pure mathematical constructs not taking into consideration that statecharts are engineers’ models primarily aiming at defining the operation of a software artifact thus the lack of support for history and branch nodes and neglecting activities reduce its practical applicability.

According to my observation the difficulty of defining a semantics for statecharts is implied by the fact that the standard defines basic concepts only (states, pseudostates and individual transitions) that are hard or even impossible to be considered in isolation and does not introduce unambiguous concepts for handling compound constructs (e.g., compound transition structures built up of multiple basic transitions and pseudostates). This observation is the basis of Latella, Majzik and Massink’s approach [23, 24]: they decompose statecharts to extended hierarchical automata (EHA) where the concepts of state refinement and compound transitions appear in a more straightforward form, then they specify a Kripke structure by a deduction system as the direct semantics of EHAs. Being used as an alternative representation of statecharts, the operational semantics of extended hierarchical automata is a denotational semantics
for statecharts. Based on this approach Varró defined the semantics of EHAs by a visual model transition system \([74]\). According to my experiences, EHA-based solutions are promising approaches but the discussion of some important constructs is still missing (branch and history pseudostates) and the prototype implementation of the statechart to EHA transformation \([75]\) can not be considered to be a fully elaborated solution either since it was not designed for the 2.0 version of UML and does not support most of pseudostates (e.g., fork, join, junction, choice, histories etc.). From an implementation aspect the main difficulty related to the approach is mapping the deduction system to an imperative programming language. Furthermore this solution does not introduce a straightforward formalism for compound activity structures either.

To put together the main drawbacks of approaches published in the literature are the lack of full elaboration and the fact that the mathematical formalisms applied in them are far from the engineering practice. According to the discussion above, the goal of my research was to define a formal operational semantics for UML 2.0 statecharts supporting the entire practically relevant set of modeling elements. The solution should be a pragmatic approach close to engineers’ way of thinking. In order to achieve this, I identified the following sub-tasks:

(i) We need a formalism for the unambiguous definition of concepts related to state refinement and we have to formally define the corresponding well-formedness rules since the standard mostly provides textual discussion only, the OCL constraints defined by it are sometimes contradictory or incomplete.

(ii) Although the standard informally uses the concept of compound transitions built up of multiple basic transitions and pseudostates, the formal definition is missing and the related well-formedness constraints are incomplete. We need to explicitly define the notion of compound transition structures, clarify the concepts related to enabled transitions, discuss the tasks to be carried out when firing a transition and we have to unambiguously define the corresponding well-formedness rules.

(iii) We need a formalism for compound activity structures that is capable of unambiguously expressing the subsequence relations amongst basic activities to be carried out when firing transitions (state exit activities, effects, entering states) and indicates the possibilities for parallel execution.

(iv) Finally, based on the formalisms introduced above, we have to define the formal operational semantics of statecharts. The approach has to be both mathematically well-established and easily applicable in the engineering practice. Mathematical preciseness should be aimed at by building the formalism on a finite state-transition system (e.g., a Kripke transition system) and this specification should be finally mapped to algorithms that are easy to implement.

The issues of behavior modeling are addressed by the first part of the thesis.

1.2 New Scientific Achievements

**Thesis 1.** I elaborated a formal operational semantics for statecharts of the UML 2.0 modeling language. As compared to other approaches published previously in the literature, my solution supports the entire set of practically relevant statechart artifacts, it is based on a well-established formalism for refinement concepts, uses a high-level formalism for representing possibilities for parallel execution and its inherently pragmatic point of view provides a solid foundation for various engineering efforts e.g., automatic code generation, runtime error detection, etc. The thesis is built up of the following parts:

(i) I defined a new formalism for state refinement concepts and state hierarchies. As compared to the informal discussion of the UML standard, my approach is based on an unambiguous formalism and a comprehensive set of well-formedness criteria.

(ii) I defined a new formalism for compound transition structures built up of multiple transitions and pseudostate vertices and defined the corresponding well-formedness criteria.
(iii) I defined a new formalism for compound activity structures built up of multiple atomic activities with precedence relations based on PERT graphs and formally defined the corresponding well-formedness criteria and algorithms.

(iv) Based on the formalisms above I defined a formal operational semantics for UML 2.0 statecharts by a Kripke transition system.

(v) Finally I mapped the basically declarative notation of the Kripke transition system to imperative algorithms that can be implemented in popular programming languages on a straightforward way.

All the formalisms, data structures, well formedness rules and algorithms were implemented by me in a simulator application using the Microsoft AsmL executable specification language. The simulator can be used with any modern UML 2.0 modeling environments that support the XML metadata interchange format.

1.3 Methodology of Research

The first question naturally emerging at the beginning of the research was which method to choose for defining the semantics of statecharts: one could follow the denotational approach similarly to a previous work developed for UML 1.x standard versions based on the introduction of extended hierarchical automata (EHA) as intermediate syntax [23, 24] or I could define a semantics directly based on concepts of the UML 2.0 standard. In case of previous standard versions the introduction of a completely new intermediate syntax (EHA) was necessitated by the lack of many vital concepts from the standard (e.g., regions were not expressed as a metaclass but they were represented by a special application of composite states); obviously it seemed to be easier to introduce a new straightforward formalism instead of modifying the standard’s concepts. Fortunately most of these syntactic ambiguities were fixed in the 2.0 version of the standard; the new metamodel now really reflects the concepts of operation instead of structure of drawing symbols as previously (e.g., the notion of regions was introduced into the metamodel, representation of embedding sub-state machines was clarified etc.) thus it seemed to be easier to define our semantics directly based on concepts of the UML 2.0 standard instead of building it on an intermediate formalism. We were able to introduce the concepts that were still missing (compound transition and activity structures) as extensions of the standard without having to fundamentally re-structure the metamodel. Another benefit of not having introduced a new intermediate formalism was the elimination of the need for defining a transformation between the two formalisms (e.g., UML and EHA) and having to prove the correctness of this transformation.

The mathematical preciseness of the Kripke transition system is obviously a valuable benefit of the approach but due to its inherently declarative nature its implementation on an imperative programming language is definitely a nontrivial issue. Thus I found it important to describe the same behavior by easy to implement algorithms also; in order to achieve this I prepared a high-level state machine view of the Kripke transition system, identified the two run-to-completion scenarios (initialization and trigger processing) and finally described them by two straightforward algorithms.

1.4 Related Publications

Discussions about statechart semantics obviously appear in our papers about code generation and runtime error detection, e.g.: [49, 52] (conference and journal papers about code generation) and [13] (book chapter about runtime error detection). The most recent version of our semantics is outlined in the mathematical introduction part of [40] (book chapter). In addition to the relatively terse discussions mentioned above, the semantics was explained in-depth in detailed technical reports: [44] (about 50 pages), [59] (about 150 pages), [62] (about 50 pages), [46] (metamodel of precise statecharts for inter-tool communication, 15 pages).
Chapter 2

Implementation of Systems Specified by Statecharts

2.1 Motivation

The widespread application of visual modeling languages naturally implies the need for automatic implementation of these models. The substitution of the time-consuming and labor intensive manual programming with automatic tools promises considerable enhancement of productivity and software quality. Modern modeling environments usually provide some support for automatic implementation of static aspects of software models (e.g., generation of data types, function declarations, class headers, interfaces etc. according to the syntax of a programming language) i.e., these solutions are restricted to the interpretation of data models (package, class and deployment diagrams). It is important to see that however the resulting framework is beneficially usable during the implementation, it provides only a minor part of the finally built code since the largest part of application logic resides in dynamic models. Unfortunately, the automatic implementation of dynamic models (activity and collaboration diagrams, statecharts) is supported only by a very few number of high-end tools. The probable reason for this is the fact that while the implementation of a class model in a programming language can be easily achieved by some template-based approaches (since there is a direct association between data modeling concepts and object-oriented programming language artifacts), the implementation of behavior means actually writing programs. This is such an intellectual challenge for which the solution requires both the in-depth understanding of the complex formal semantics of behavior models, solid knowledge on the characteristics of the target platform and obviously a considerable expertise in the programming language used. These difficulties render the automatic implementation of statecharts a much more complicated task than generating data type declarations from a class model according to some simple templates. Even with respect to the issues mentioned above, the automatic implementation of behavior specified by statecharts is an important and up-to-date research area since because of the rich modeling toolkit and the complexity of the operational semantics the manual implementation of behavior specified by statecharts is a highly labor intensive and error-prone task and a minor modification of the model may require rewriting the complete implementation.

The reactive, state-based, event-driven behavior is typical for embedded systems that are in direct connection with their environment. In case of this application area not only the theoretical issues mentioned above are to be considered but we also have to take into account that the computing resources of the underlying platform (operative memory, processing power) are seriously constrained (small memory, weak CPU or microcontroller) thus the popular modern object-oriented languages and the corresponding virtual machines (Java, .Net, etc.) are not available in most cases.

None of the approaches published previously in the literature for the implementation of event-driven state-based systems are sophisticated enough for the implementation of UML 2.0 statecharts. The best known corresponding approach is the State design pattern [14], but unfortunately it is not much more than a set of suggestions for the representation of states by classes in a plain (non-hierarchical)
CHAPTER 2. IMPLEMENTATION OF SYSTEMS SPECIFIED BY STATECHARTS

state machine and does not propose any solutions for handling state hierarchies, concurrent operation etc. Some popular modeling environments also advertise providing support for the implementation of statecharts but taking a closer look we usually find that these solutions do not provide more than the automatic declaration of some function signatures or the approach is restricted to seriously simplified non-hierarchical state diagrams [1]. The most elaborated implementation method, the Quantum Hierarchical State Machine (QHsm) pattern published in the literature was proposed by Miro Samek [70, 71]. The QHsm pattern provides support for the implementation of those UML statecharts that do not contain concurrent state decomposition. According to my knowledge the most complete solution is provided by the Rhapsody and VisualSTATE software families from the I-Logix and IAR companies respectively; however since these are high-end commercial integrated modeling and development environments, the implementation algorithm of statecharts was not published.

With respect to the discussion above, the goal of my research was to propose a method for automatic programming language level implementation of UML 2.0 statecharts according to the operational semantics presented in the first part of the thesis. I decided to put explicit emphasis on the applicability of my approach in resource constrained embedded systems. In order to achieve this I identified the following sub-tasks:

(i) We have to identify the typical resources of the target platform (resource constrained embedded systems) and based on this analysis the platform-specific syntax and semantics of statecharts is to be specified according to OMG’s Model Driven Architecture initiative. The platform-specific representation maps the general UML artifacts and behavioral concepts to the resources available on the target platform (e.g., the CPU available on an embedded platform is not capable of the parallel execution of multiple threads thus concurrent operation should be substituted by a valid sequential scheduling of activities; with respect to the low amount of available memory we have to find compact solutions for storing data, furthermore in case of real-time systems we may have to substitute recursive algorithms by iterative implementations whose execution time analysis is easier, etc.). Having defined the platform-specific representation we have to prove that the resulting semantics corresponds to the platform-independent original one with respect to the characteristics of the target platform.

(ii) The algorithms and data structures of the platform-specific semantics are to be implemented in a programming language that seamlessly fits to the target platform. Similarly to the previous step we have to prove that the resulting source code correctly implements the platform-specific semantics.

(iii) The resource consumption of the applications built according to our approach is to be analyzed (memory consumption and execution time) and we also have to experimentally evaluate these properties for some benchmark models. With respect to the practical applicability of the approach I also find important to actually compare my solution with the most elaborated method published previously in the literature (QHsm) instead of barely presenting measurement results.

The issues related to the automatic implementation of UML 2.0 statecharts are addressed by the second part of the thesis.

2.2 New Scientific Achievements

Thesis 2. I elaborated an effective source code-level implementation of behavior specified by UML 2.0 statecharts. The low memory and processing power requirements of my solution enable its application even in resource constrained embedded systems. The thesis is built up of the following parts:

(i) According to OMG’s Model Driven Architecture initiative, I identified the typical resource conditions of embedded platforms (low performance CPU, lack of support for parallel execution, heavily constrained memory) and based on these observations I elaborated a platform-specific mapping of data structures and algorithms specified in the first part of the thesis.
(ii) I implemented the intermediate platform-specific representation in the ANSI-C programming language that seamlessly fits to the target platform. This solution is the most complete published approach for source code level implementation of systems specified by statecharts. The correctness of the solution was shown by systematic comparison of algorithms and data structures at corresponding transformation steps.

(iii) I carried out the analysis of algorithmic complexity and theoretic memory consumption of my approach. In order to underpin the formal analyses I also performed the experimental evaluation of memory and processing power requirements comparing applications built according to my solution and the QHsm pattern. As QHsm supports less modeling concepts than my method, I used models whose implementation was possible by both approaches. The experiments have shown that for sufficiently complex models my solution does not only support much more modeling concepts but the resource consumption of the resulting applications is also lower.

2.3 Methodology of Research

In case of a code generation approach the following questions naturally emerge: what is the principle of operation in the generated applications, how to evaluate the resource consumption of the applications, how to prove the equivalence of the abstract semantics and the source code and how to integrate our solution to modern modeling environments; below we will discuss these issues.

The source code resulting from the code generation operates as an interpreter: the code generator constructs an initialized static constant data structure (as ANSI-C source code) that describes the statechart; this data structure is to be linked to a generic interpreter routine that directly implements the algorithms specified in the platform-specific semantics. Activities of the statechart (state entry and exit activities, transition effects) and guard predicates are represented by functions whose signatures are declared by the code generator and whose body is to be implemented by the programmer. The resulting programs operate in a callback structure: at the reception of an event the programmer has to call the interpreter and provide the event; then the interpreter updates the configuration and calls the appropriate functions (implementations of activities) with respect to the original configuration and the evaluation of guards. The interpreter-like operation is beneficial because the UML syntax (metamodel) and the semantics specified by us (algorithms) are directly mapped to programming language constructs, there is no wide semantic gap between the representations thus it is quite easy to show the equivalence of the abstract semantics and the implementation. With respect to the interpreter like operation it is important to highlight that obviously we do not process models (e.g., XMI files) during runtime but compact binary structures and we put emphasis on the pre-calculation of all those calculations that can be evaluated during code generation and store the results in the binary structure; this solution unifies the clarity of generic interpreter-like approaches and the high performance of dedicated solutions. According to my knowledge this is the most elaborated approach published in the literature as our proposal supports the most of key statechart modeling concepts (state hierarchies, concurrent operation etc.) while approaches published previously in the literature were mostly restricted to non-hierarchical state machines and not even the most elaborated solution known from the literature (QHsm) supports concurrent state decomposition. (The methods applied in the few number of commercial tools supporting code generation from statecharts are not public thus they can not be considered to be published solutions.)

It is interesting to assess the resource consumption of the generated applications. I found it beneficial to compare my solution to the QHsm pattern known from the literature. This comparison is particularly interesting since my solution is based on more complex algorithms than QHsm (primarily due to the support for concurrent state decomposition) thus we have to make sure that the benefits of supporting more modeling concepts are not lost because of increase in processing power and memory consumption. Obviously I had to use such models that can be implemented by both patterns thus I might not use those constructs that were not supported by the QHsm pattern. As the code generation based on statechart models is a relatively new research area there are no such standard benchmarks such as the synthetic
load patterns available for web servers, databases etc. thus we put together some scalable model schemes (e.g., a loop consisting of \( n \) states, a full mesh of \( n \) states, a pair of \( n \) level deep state hierarchies etc.). The benchmark applications were generated from these models and I provided a sufficiently long trigger sequence to measure the execution time; the memory consumption was assessed analytically.

Since the interpreter-like implementation directly maps the data structures (metamodel) and algorithms of the semantics to programming language constructs (through the intermediate step of the platform-specific representation), **proving the equivalence** of the specification and the implementation is quite easy: first I analyzed line-by-line the differences between the platform-independent and platform-specific algorithms, then the differences between the platform-specific algorithms and the source code level implementation and discussed the reasons for having taken these modifications and how these modifications affect the resulting behavior. Using this method I have shown that the implementation method targeting resource constrained embedded systems as outlined in the thesis corresponds to the platform-independent semantics taking into consideration the characteristics of the target platform (e.g., lack of support for parallel execution etc.).

The practical applicability of my solutions necessitates their **integration to modern modeling environments**; thus all solutions discussed here (statechart simulator, code generator and runtime error detection solutions) process files in the standard XML Metadata Interchange (XMI) [35] format supported by most of modern modeling environments.

### 2.4 Related Publications

While the formal operational semantics introduced in the first part of the thesis was not yet available, our early research focusing on automatic implementation of statecharts was based on a denotational semantics built on extended hierarchical automata published previously in the literature [23, 24]; at this stage of the research we published a conference paper [49], a journal article [52], a detailed technical report [50] (about 50 pages) and a short paper at a local PhD symposium [51]. In another conference paper [56] we analyzed the impact of statechart implementation techniques on the effectiveness of error detection mechanisms. Having prepared the new operational semantics, we presented our method for automatic implementation of statecharts in its present form in another detailed technical report [61] (about 100 pages).

Furthermore I took part in the preparation of two conference papers [30, 31] that discuss how to represent the structure of tasks in a hard real-time system using the toolkit of UML and how to generate input files for target-specific configuration tools from the visual models. Our other two code generation-related papers published on PhD symposiums [53, 54] are less directly bound to statecharts: we proposed a model-based code synthesis method for storing structured memory contents in files, to be used in fault tolerant systems for the implementation of rollback and recovery operation.
Chapter 3

Runtime Error Detection in Statechart Implementations

3.1 Motivation

The issues related to the semantics and automatic implementation of visual behavior models are addressed in the first two main parts of the thesis; the third part focuses on another emerging research area by discussing the application of behavior models as reference information for runtime detection of errors caused by design, implementation or physical faults.

Aiming at the enhancement of software dependability is another goal that does not require much justification: the use of both general-purpose and specialized computers pervasively extends to virtually all facets of life, the dependence of the society on their correct behavior leads to growing impact of these systems’ dependability. In the past, the term “safety critical” was primarily used for referring to such evident cases as flight control systems or nuclear power plant operation etc. At present, the wide variety of applications ranging from on-board embedded microcontrollers in cars to the use of high performance servers driving the Internet infrastructure all constitute “critical” systems whether for their clear impact on human health or financial conditions [8]. The typical origins of failures have also changed over the years: while in early computer-based systems the primary causes of service unavailability was related to hardware faults, modern systems are usually built on highly dependable hardware thus the main cause of failures is related to design or implementation faults in the software [8, 15, 79]. Due to its importance, runtime error detection is explicitly required even by standards of software development for safety critical applications (e.g., IEC 61508, EN 50128).

Using the approaches presented here complements the exploitation of behavior models’ expressive power: the same statechart can play the role of design documentation, provides the input for automatic code generation and can be used as reference information for runtime error detection.

Having decided to aim at runtime checking of dynamic behavior, the first obvious task is to select those classes of faults that our approaches should address and identify those phases of the software development when these faults may be introduced. As a beneficial property of model-based development, we are not restricted to checking informal and hard to capture textual requirements during the execution of the implementation but we can provide integrated and formally well-established solutions for addressing faults causing behavioral errors introduced at any key phases of the development.

In correspondence to the V-model of safety-critical software development (Fig. 3.1) I considered the following development phases during my research (still focusing on the specification of control flow, i.e., evolution and implementation of statecharts): (i) during requirement analysis those fundamental temporal dependability requirements are collected that the implementation should meet in any cases (e.g., avoidance of dangerous operation modes etc.); (ii) the early versions of statecharts that capture only the key aspects of behavior (states, notion of safe and dangerous situations, operation modes, etc.) are constructed in the specification phase; although these early models require further elaboration, they enable the formal specification of the previously collected temporal dependability requirements in
Figure 3.1: The V-Model of Software Development

the context of statecharts; (iii) the fully elaborated statecharts that unambiguously specify the behavior of the system are prepared in architecture and module design phases; these statecharts correspond to the draft ones prepared in the specification phase (i.e., they can be considered as refinements of early models) and they are ready for being implemented; (iv) the source code level representation of behavioral models is prepared in the implementation phase by manual programming or automatic code generation; (v) finally after verification, integration and validation phases the system is certified and deployed. According to the observation above the following points can be identified where faults can be introduced into the system (Fig. 3.1):

- If the set of requirements collected in the requirement analysis phase is inconsistent or faulty or their initial formalization carried out in the specification phase is incorrect, the entire development is likely to fail due to fundamental misunderstanding between the developers and future users of the system. Ensuring that the user requirements (functional and non-functional ones) are fully understood by the developers and these requirements are correctly formalized in the specification is usually aimed by in-depth communication with users involving the construction of visual models, screenshot concepts and fast prototypes and discussion of typical application scenarios. As the correctness of the initial specification primarily depends on informal human interaction, the earliest entry points of our solutions are the temporal requirements defined in the specification phase (thus faults introduced during requirement analysis and specification phases are out of the scope of our work). Note that most of the approaches published previously in the literature expect much more elaborated inputs (e.g., design models or the implementation) thus they can not address faults introduced during the development (i.e., during which these more elaborated models are constructed, e.g., architecture and module design, implementation phases etc.).

- During the architecture and module design phases such model refinement faults may be introduced that result in incorrect models violating some temporal requirements of the specification.

- The application built in the implementation phase may not correspond to the behavior model elaborated in design phases; the possible reasons for this are programming bugs or misunderstanding the semantics; the possibility of these faults can be reduced by automatic code generation but can not be totally eliminated while the entire source code is not automatically synthesized by proven correct techniques.

- Finally, transient and permanent faults may occur during operation in the underlying hardware-software infrastructure that affect the behavior.

The corresponding research presented in the thesis primarily focuses on two of these fault classes i.e., the ones introduced in design and implementation phases. Obviously for addressing a fault introduced at
3.1. MOTIVATION

In correspondence to the solutions published in the literature for addressing design faults (i.e., faults in the model refinement) it seems to be beneficial to define a temporal logic language \[66\] for the definition of temporal requirements and checking that these requirements hold during runtime. According to our expectations by defining requirements in the context of specification models we will be able to address faults introduced even during the development (model elaboration) in contrast to previous approaches that use (fully elaborated and possibly faulty) design models as reference. Having defined the language, the evaluation of its formulae over execution traces of observed applications is another issue. The thesis presents a detailed overview about previously published methods and tools; this survey can be summarized as follows: although there have been several approaches proposed in the literature from the field of model checking, these solutions are not usable for handling finite execution traces for semantic reasons or can not be directly applied in resource constrained embedded systems due to their high resource consumption. (Model checking typically aims at proving various properties of protocols and automata expecting them to be running infinitely long. Since runtime error detection has to observe real applications that may happen to be terminated, the language and the method of verification has to be able to semantically handle the notion of finite execution traces. Checking finite traces is an open research area with many valuable achievements \[12, 18, 67, 68\] but the high resource consumption of the previous solutions prevent their application in resource constrained environments or the approaches seriously reduce the expressive power the usable temporal logic language.)

The goal of the implementation phase is to map the fully elaborated models to a programming language thus it seems to be obvious to specify a runtime monitor that compares the application’s actually exposed behavior to the design models (i.e., fully elaborated statecharts) in order to detect errors caused by implementation and operational (physical) faults. The monitor discussed here closely resembles some watchdog solutions proposed previously in the literature. Watchdog processors (WDP) are relatively simple coprocessors that detect if the application running on the main CPU deviates from the behavior specified by the control flow graph. A control flow graph (CFG) is a directed graph whose nodes represent those blocks of machine instructions that do not contain branches while edges of the graph correspond to syntactically enabled control transfer (jump instructions, subroutine calls etc.). The correct (expected) behavior described be the control flow graph is called the reference information. The watchdog processor checks the behavior of the application by directly observing its instruction fetch operations on the system buses, or the watchdog can be implemented as an ordinary peripheral device to which the application explicitly transmits information about its current activities. The fragments of observed behavior (instructions fetched or information explicitly transmitted) are called signatures of operation. The control flow graph can be stored in the watchdog (by an adjacency matrix or list, etc.) \[26, 32\] or it can be embedded in the signatures sent by the application \[28, 37\]. The idea of my approach is to construct a watchdog-like structure whose reference “control flow graph” is actually a fully elaborated statechart. Although methods based on control flow graphs were successfully applied for monitoring relatively low-level constructs (functions, interrupt routines, etc.) our task is more complicated
CHAPTER 3. RUNTIME ERROR DETECTION IN STATECHART IMPLEMENTATIONS

than the comparison of a control flow graph and an execution trace since states of a statechart can be organized into a hierarchy even involving concurrent operation. The need for runtime error detection based on a statechart specification has already occurred in the literature [29] but the solution has still remained an open issue. From an implementation point of view we have to take into consideration that in our case the reference information is of much higher abstraction level than in case of previous approaches: while in case of a traditional watchdog processor an edge in the control flow graph corresponds to a single or a few number of machine instructions, in our case an edge corresponds to the entire response of the application to an event – this means that the steps to be checked are delivered to the monitor at a significantly lower bandwidth but actually checking these steps is a much more complicated task than searching for an edge in a graph. According to this observation, in contrast to the inherently hardware-based approach of traditional watchdog processors, I decided to develop a software solution (but obviously not excluding the hardware implementation).

To put together the goals of my research, I will present two complementary approaches: faults possibly introduced during model refinement (architecture and module design) will be addressed by defining and checking temporal requirements while errors caused by implementation faults are to be detected by the runtime monitor. I defined the following sub-tasks:

(i) We need a temporal logic language fitted to statecharts that besides the usual Boolean and temporal constructs provides facilities for referring to statechart artifacts (states, transitions, activities etc.).

(ii) We have to develop an efficient method for runtime evaluation of temporal logic expressions on finite execution traces. The low resource consumption of the solution should enable its application as an error detection module in resource constrained embedded platforms.

(iii) Analogously to the watchdog processors known from the literature, we have to develop a runtime monitor that checks whether the behavior exposed by the application corresponds to the specification using fully elaborated statecharts as reference information.

(iv) Obviously emerges the need for assessing, which are those fault classes (design, implementation or physical faults) that can be most effectively addressed by our solutions and which are the ones that can be most beneficially addressed by the so called conventional error detection mechanisms of the execution platform (memory protection, consistency checking of machine instructions, sanity checking of arithmetic operations etc.); thus I decided to experimentally evaluate the error detection capabilities of my solutions by a fault injection campaign.

The issues related to runtime error detection in UML 2.0 statechart implementations are addressed by the third part of the thesis.

3.2 New Scientific Achievements

Thesis 3. I elaborated novel methods for detecting errors in implementations of UML 2.0 statecharts. The thesis is built up of the following parts:

(i) I defined a propositional linear temporal logic language fitted to the statechart syntax and semantics introduced in the first part of the thesis. The language enables the definition of temporal correctness criteria referring to various statecharts artifacts (states, transitions, etc.) and activities performed during execution. Due to the high abstraction level of the language, the essential temporal correctness criteria can be defined even in early phases of the development when only draft versions of statecharts are still available.

(ii) I defined a novel high performance method for evaluation of propositional linear temporal logic expressions over finite execution traces. According to my analysis and experimental evaluation the solution delivers significantly higher performance than previous approaches while its low CPU and memory requirements allow its application in resource constrained embedded environments.
(iii) I elaborated a monitor structure for UML 2.0 statechart implementations fitted to the semantics defined in the first part of the thesis. The solution is capable of detecting errors related to violation of the specification with respect to maintaining state configuration, firing transitions and execution of compound activity structures.

(iv) I carried out a fault injection campaign for experimental evaluation of error detection capabilities of my solutions. The experiments indicated that checking temporal requirements was effective for detecting errors caused by model refinement faults, the runtime monitor was successfully used for detecting errors caused by implementation faults while physical faults can be most beneficially addressed by traditional watchdog processors and low-level HW/OS mechanisms.

I implemented a code generator that for an input set of temporal logic expressions synthesizes a source code for evaluating them on finite execution traces according to (ii) and I prepared a prototype implementation of the runtime monitor mentioned in (iii).

3.3 Methodology of Research

With respect to the methodology of research it is worth outlining our proposal for the effective evaluation of temporal logic expressions and discussing the theory behind the implementation of the runtime monitor (watchdog).

We introduced our solution for the effective evaluation of temporal logic expressions on finite execution traces by a straightforward hardware analogy: (i) first we defined a normal form for temporal logic expressions where the sub-expressions corresponding to the present (actual state) and the future are clearly separated; (ii) we introduced the notion of “evaluation nodes” representing expression sets that are derived during the iterative decomposition of the input set of expressions; we illustrated the concept of an evaluation node by a logic circuit whose outputs are the values of the corresponding expressions, its inputs are labels of the actual state and the values of sub-expressions corresponding to the future while its internal structure describes the logical relation of inputs and outputs; (iii) evaluation nodes are connected one after another resulting in a chain of evaluation nodes that evaluates the original set of expressions over the execution trace. Finally we mapped the evaluation nodes of the hardware analogy to programming language classes and implemented a code generator that for an input set of expressions synthesizes the source code that evaluates the expressions on a finite trace according to the method outlined above. This solution is a valuable achievement even on its own since it is not bound to statecharts thus usable for the evaluation of expressions in practically any linear temporal logic languages and according to our experiments this solution features significantly higher performance and lower memory consumption than any other approaches published previously in the literature.

The runtime monitor aiming at the detection of errors caused by faults in the implementation of statechart-based behavior was inspired by the theory behind some language parsers: (i) let us consider the Kripke transition system that defines the operational semantics as an automaton whose transitions are annotated by output fragments; each of these output fragments describes a step of the application (e.g., opening a “run-to-completion” step, firing a set of transitions, dropping an event etc.); (ii) thus the output fragments printed by this automaton are words of a language, an execution of the automaton is a sentence of this language and all possible executions of the automaton correspond to all possible sentences of the language; it is easy to see that this language represents all semantically correct executions of the application; (iii) let us construct an automaton that accepts this language (this is quite easy since the language is specified by a finite state machine); this automaton will accept the semantically correct executions of the application and drop the ones that violate the semantics. The runtime monitor is an implementation of this automaton. Obviously we also have to check the sanity of the steps themselves (e.g., to check whether the transitions selected for firing in the step are really enabled, there is no conflict amongst them, etc.); these checks can be represented as guard predicates assigned to transitions of the accepting automaton and derived directly from the formal semantics.
3.4 Related Publications

An early approach for runtime error detection in statechart implementations also appeared in my TDK/OTDK (scientific students’ association) works [42, 43] and was discussed in the detailed technical report describing that research [64] (about 50 pages); these early solutions were still developed for plain state machines and could not support hierarchical state decomposition.

Similarly to our first code generation techniques, while the new semantics discussed in the first part of the thesis had not yet been defined, our runtime error detection approaches were built on a previously published semantics based on extended hierarchical automata [23, 24]; we presented these solutions in a conference paper [55] and a book chapter [13]. We also published a paper [56] discussing the impact of statechart implementation techniques on the effectiveness of error detection mechanisms. Having defined the new statechart semantics we fitted our previous achievements on runtime error detection to the new formalism and documented them in their present form in a detailed technical report [63] (about 100 pages).

Our temporal logic language developed for reasoning about temporal correctness of statechart implementations was first outlined in the book chapter mentioned above [13] while the high performance solution for the evaluation of temporal logic formulae was published in a conference paper [58].

Obviously we also presented some smaller publications in this subject: [60] (PhD symposium), [45] (conference paper in Hungarian). Moreover we published a conference paper [57] to show how to represent signals of error detection mechanisms in the UML statechart of the application itself as ordinary events. Based on this description we presented a scheme for organization of states in a statechart for modeling behavior similar to exception handling in traditional programming languages; the approach visually separates normal behavior and handling exceptional situations. This scheme enables the modeling of normal operation and the behavior in the presence of faults at the same high abstraction level. After the conference our paper was invited to the post-proceedings of the selected publications [16].

Having defined the new formal operational semantics for statecharts discussed in the first part of the thesis, we published another book chapter [40] that presents our error detection approaches based on the most recent mathematical foundations.
Chapter 4

Automatic Processing of Measurement Results

4.1 Motivation

It is widely accepted that the evaluation of dependability attributes of computer systems is a complex task. Traditional techniques based on analysis and simulation have to be supported with experimental evaluation of prototypes and observation of real systems. These experimental techniques, including fault injection, robustness testing, and field measurements, have been successfully used to evaluate specific fault tolerance mechanisms, validate robustness of software components, or to assess the general impact of faults in systems.

In spite of the big diversity of techniques and tools now available, all the experimental dependability evaluation approaches share a common problem: they tend to produce a large amount of raw data that has to be processed to obtain the desired dependability measures. Very often the analysis of the experimental data is quite complex, as it has to take into consideration many aspects of the experiment setup e.g., the target system architecture and configuration, the workload, the type of faults involved, the environmental aspects, etc. Surprisingly, the problem of coping with the large size of the experimental data sets and the high complexity of the data analysis has received less attention in the dependability research effort. Researchers have focused on the development of fault injection and robustness testing tools and on the mitigation of problems such as experiment representativeness, non-intrusiveness and portability of tools. In correspondence to this, many fault injection or robustness testing tools have been proposed for the experimental evaluation of dependability attributes (e.g., FERRARI [21], Mafalda [69], Xception [5], Ballista [19], NFTAPE [72]), but all these tools either provide rudimentary means to analyze data or, more frequently, just store the raw results in a spreadsheet format. Although this approach can be acceptable for very specific and simple analysis, it is clearly not enough when the analysis required is complex or when the amount of raw data is very large.

Another force rendering automatic measurement evaluation approaches an up-to-date research area is the occurrence of dependability benchmarking proposals. These new approaches represent an attempt to standardize experimental techniques with the goal of comparing dependability features of different systems or components. This research effort has already caught the attention of companies such as Sun Microsystems [81], IBM [25] and Intel [7], and lead to many dependability benchmark proposals, covering domains such as transactional systems [4, 33, 76], web servers [11], and operating systems [20]. Even topics such as human faults [3] or hardware maintenance [81] have already been subject of dependability benchmark proposals. Dependability benchmarks represent a new and important source of raw experimental data but the problem of analyzing that data has been even more neglected than in case of traditional fault injection and robustness testing. In fact, dependability benchmarks rely on a typically small set of measures and the data collected during the benchmark runs is just used to calculate the measures defined in the benchmark.

There are a relatively low number of proposals published in the literature focusing on automatic
processing of measurement data: Madeira, Costa and Vieira suggested first [27] to store the large amount of data obtained from dependability evaluation experiments in data warehouses and support the analysis and visualization by OLAP methods (On-Line Analytical Processing) enabling this way the effective sharing and comparison of data between research groups. The approach presented by them considered the data analysis as an interactive process where the analyst has some a priori assumptions and specifies queries for the database using the slice-and-dice approach of OLAP to prove or falsify the original assumption. In this method the specification of relevant queries and understanding the results, thus the success of the analysis, heavily depends on the experience and human intuitions of the data analyst.

The application of data mining, the most advanced method of intelligent data processing was first suggested by Pataricza and Tolvaj [39] for analysis of data obtained from fault injection experiments. Data mining is usually defined as an interdisciplinary field bringing together techniques from machine learning, pattern recognition, statistics, databases, and visualization to address the issue of extracting previously unknown, valid and actionable information from large databases to be used for making crucial business decisions [17]. Data mining surpasses the semi-automatic, human interaction-based approach of OLAP analysis by relying on fully automatic machine learning algorithms enabling this way the analysis of data sets from non-intuitive aspects that are not emerging from any a priori human intuitions.

Although the papers mentioned above propose valuable ideas, the automatic processing of data obtained from experimental evaluation of dependability attributes has still remained an open issue. It is important to highlight that we do not argue that the automatic processing of large data sets has not been discussed in the literature since obviously OLAP analysis and data mining are mathematically well-established and successful research areas – we would like to highlight here that despite of the success of data intelligence in the business field these techniques have not been used for processing data obtained from dependability evaluation experiments.

According to the observation above the goal of my corresponding research was to propose a method for automatic analysis of data obtained from experimental evaluation of dependability by the application of data mining techniques. I identified the following sub-tasks:

(i) We have to elaborate a data mining based method that enables the automatic identification of those (infrastructural) factors and faultload properties that determine the behavior of a complex system in the presence of faults by processing data obtained from fault injection experiments.

(ii) The viability of the approach should be demonstrated by the analysis of data obtained from a dependability benchmarking experiment.

(iii) Finally I would like to demonstrate the joint application of achievements from the four parts of the thesis by a comprehensive experiment campaign according to the following scenario: (i) let us define a behavior model by a UML 2.0 statechart (according to the semantics discussed in the first part of the thesis), (ii) generate the source code of applications by the approach discussed in the second part of the thesis, (iii) examine the behavior of applications in the presence of faults applying the error detection methods presented in the third part of the thesis (the faultload should contain both model refinement, implementation and low-level faults) and finally (iv) analyze the resulting large database by the method elaborated in the fourth part of the thesis. The experiments should aim at the identification of those parameters (configuration of the code generator, compiler options, faultload, workload etc.) that primarily determine the dependability of service delivered by the system and the efficiency of error detection mechanisms.

Our research focusing on automatic analysis of data obtained from experimental evaluation of dependability attributes is presented in the fourth part of the thesis.

4.2 New Scientific Achievements

Thesis 4. I presented a data mining based method for automatic identification of those key infrastructural factors (configuration, HW/SW components) and properties of the faultload that significantly affect the
behavior of complex HW/SW systems in the presence of faults:

(i) I elaborated a method for automatic processing of data obtained from dependability evaluation experiments by the application of the classification data mining technique. The method provides support for automatic identification of those infrastructural, workload and faultload factors that significantly affect the behavior of complex HW/SW systems built up of COTS elements.

(ii) I demonstrated the viability of the approach by the investigation of a data set obtained from fault injection experiments conducted on a complex data warehouse (DBench-OLTP experiments).

(iii) I assessed the error detection capabilities of our techniques presented in the third part of the thesis and identified their most beneficial application areas. (The experiment campaign applies the semantics discussed in the first part, the implementation method proposed in the second part and the data mining based analysis technique from the fourth part of the thesis.)

4.3 Methodology of Research

The method for automatic identification of factors primarily determining the behavior of the system in the presence of faults is based on the analysis of decision trees constructed by the classification data mining technique: (i) the input of the method is a set of data obtained from a fault injection experiment campaign; the experiments should investigate the system’s behavior with respect to various HW/SW setups (e.g., in case of an OLTP system multiple database managers running on various operating systems and hardware configurations) and being stressed by various faultloads; (ii) experimental data is organized into a database table where a record represents a single experiment, while attributes of the actual experiment (hardware setup, software components, faultload, etc.) and the properties of the observed behavior (performance delivered, failures experienced by the user etc.) correspond to columns of the table; (iii) next we configure the data miner to construct a model for predicting the output parameters (observed behavior) based on the infrastructural attributes (HW/SW properties, faultload). The data miner delivers the model as a decision tree whose internal nodes are basic decisions that divide the actual subset of data to two disjoint subsets by a Boolean predicate over the value of some infrastructural attribute (e.g., the operating system in case of the experiment represented by the record was X) while leaves of the tree are decision results representing the characteristic properties of records in the corresponding subset (e.g., during the experiment represented by the record we experienced low performance); in this structure a root-leaf path of the tree selects a characteristic subset of data (e.g., using the database manager X on the operating system Y and the fault occurring is Z, the performance will be low); it is easy to see that the decisions on the path identify those factors that we are looking for (in the example above these are the database manager, the operating system and fault type) and does not contain the ones of low importance (e.g., the hardware in our example).

4.4 Related Publications

Application of data mining techniques for automatic processing of data obtained from fault injection campaigns has already appeared in my TDK/OTDK (scientific students’ association) works [42, 43]; the achievements of this early research were published in two short conference papers [38, 65] and I also prepared a detailed technical report about the subject [64] (about 50 pages).

The research discussed in the thesis was mainly carried out during my stay in Coimbra, Portugal in 2004 at the research group lead by Henrique Madeira. The database analyzed in our corresponding conference paper [47] was based on their fault injection experiment carried out in the framework of the DBench project. Our research conducted in Coimbra was summarized in a detailed technical report [48] (about 120 pages).
Chapter 5

Practical Application of Achievements

A direct application of the formal operational semantics discussed in the first part of the thesis is the statechart simulator mentioned above but obviously its primary benefit is providing the semantic foundations for our automatic code generation and runtime error detection techniques presented in later parts of the thesis.

By the application of the implementation method discussed in the second part of the thesis we can develop code generation solutions that are capable of mapping complex behavioral models to source code: this approach promises a significant enhancement of productivity, reducing development costs and due to the elimination of error-prone manual programming increases software quality and maintainability. Automatic implementation of visual models for fast prototyping purposes can provide valuable support even at early phases of software development by helping the communication of developers and users.

Using the runtime error detection techniques introduced in the third part of the thesis such solutions can be developed that are capable of signaling if the behavior of a reactive state-based event-driven system deviates from its specification. In a dependable system these error detection signals can initiate fault-tolerance related activities involving error confinement, fault removal and continuation of the service or taking a system to a fail-safe state depending on the application domain. Our solutions can be beneficially applied even in case of non-critical systems by discovering faults related to state-based behavior during the testing phase.

The intelligent data processing method discussed in the fourth part of the thesis enables the automatic processing of large sets of data obtained from fault injection experiments aiming at the identification of those key infrastructural factors and faultload properties that affect the behavior of the system under investigation in the presence of faults. The method does not rely on any a priori intuitions being this way capable of discovering such phenomena that were not expected or even contradictory to the expectations. As the method does not require in-depth expertise in databases or statistics it can be integrated into the toolkit of computer dependability experts with a minor effort.

With respect to the practical application of my achievements it is worth noting again that the viability of all approaches proposed in the thesis were demonstrated by prototype implementations and their efficiency was assessed by comparing them to other solutions proposed previously in the literature.

The work presented in the thesis was carried out in correspondence to various Hungarian research and industrial projects and international cooperations. Research activities were carried out in the framework of the following projects:

- TEMPUS project – in 2001 I spent a month at the Friedrich-Alexander University, Erlangen, Germany (Friedrich-Alexander Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 3, Rechnerarchitektur), where I was working on the development of a distributed experiment controller system for supporting fault injection campaigns.

- “Self-Checking and Runtime Verification in Computer Programs” OTKA project (OTKA T-046527) – research on runtime error detection techniques was carried out in the framework of the project.
“Intelligent Measurement Data Processing for the Construction of Dependable IT Systems”, (Hungarian-Portuguese Bilateral Scientific and Technology Development Cooperation Agreement P-19/03) – in 2004 I spent a month in Coimbra, Portugal at the Centre for Informatics and Systems of the University of Coimbra with the Dependable Systems Research Group were we carried out research focusing on intelligent data processing.

The industrial application of our research achievements was carried out in the framework of the following programs:

- “EU-Conform, Constructive Safety Assessment of Railway Control Systems” GVOP project (GVOP-3.1.1-2004-05-0523/3.0, type: GVOP AKF KöZ, 2005-2006) – the elaboration of state-chart semantics and assessment of runtime error detection mechanisms were carried out in the framework of the project.

- In correspondence to the OTKA T-046527 project mentioned above we carried out fault injection experiments at the PROLAN Process Control Co.

- Embedded Information Technology Research Group of the Hungarian Academy of Sciences and Budapest University of Technology and Economics – my work at the research group focused on the development of automatic code generation techniques and the application of our achievements on the \textit{mitmőt} \cite{mitmott} modular embedded system developed by the group.

- “High-Availability Java Project” (cooperation with NOKIA Research Center) – we carried out research related to the development of checkpoint and recovery solutions supported by automatic code generation.

- Bosch Hungary (Robert Bosch Kft.) – our cooperation focuses on automatic test suite generation and test coverage analysis.
Chapter 6

Publications Related to the Thesis

This chapter enumerates our publications related to the thesis and indicates the known references to our works.

6.1 Book Chapters


6.2 Journal Papers Published in Hungary


Known references:


6.3 Publications in Foreign Language Conference Proceedings


Known references:


6.4 Publications in Hungarian Language Conference Proceedings

6.5 Works of Non-Publication Value

6.5.1 PhD Symposia


  Known references:


6.5.2 Research Reports


  Known references:


6.5.3 Scientific Students’ Conferences, Master’s Thesis


6.6 Publications Currently Under Review

• Gergely Pintér, Henrique Madeira, Marco Vieira, András Pataricza, and István Majzik. Integration of OLAP and Data Mining for Analysis of Results from Dependability Evaluation Experiments. Submitted to International Journal of Knowledge Management Studies, Inderscience, to be published in 2007 (journal article under review).


Bibliography


