Tekes Ubicom program

LIME project

LightweIght formal Methods for distributed component-based Embedded systems

Final Report

Date:	January 29, 2010
Sites of research:	Department of Information and Computer Science Helsinki University of Technology (TKK) Department of Information Technologies Åbo Akademi University (AAU)
Responsible leader:	Prof. Ilkka Niemelä Tel: +358-9-470 23290 Email: Ilkka.Niemela@tkk.fi
Project duration:	1.10.2007 - 30.09.2009

1 Introduction

This is the final report of the project LightweIght formal Methods for distributed component-based Embedded systems (LIME) carried out at the Department of Information and Computer Science of Helsinki University of Technology (TKK) and Department of Information Technologies of Åbo Akademi University (AAU) during the years 2007-2009. The project has been funded by the Finnish Funding Agency for Technology and Innovation (Tekes), Space Systems Finland Oy, Nokia Oyj, Conformiq Software Oy and Elektrobit Oyj. The goal of the project has been to develop new lightweight design and validation methods for distributed embedded software components which are easy to introduce in a conventional design flow in a stepwise manner.

Executive Summary

The results and achievements obtained in the project with respect to the project plan are summarized below.

- The project has reached its main goal of developing new techniques for designing and validating distributed embedded software components successfully, the key achievements being: (i) an interface specification language for software components, (ii) a method for generating these specifications from UML protocol state machines, (iii) a method for monitoring the specifications at runtime, (iv) a technique for generating test cases for software components in a way that the specifications are taken into account, and (v) evaluation of

state-of-the-art lightweight approaches for software validation and evaluation of the methods and tools developed in this project.

- The developed techniques have been implemented and evaluated in proof-of-concept style prototype tools that have been made publicly available under an open-source license.
- The project has produced six Master's Theses, one Bachelor's Thesis, one student project report, two technical reports, and one international scientific publication. Furthermore, the project has provided valuable post-graduate education for the people that have worked in it.
- The costs of the project have stayed well within the budged.

The rest of this report is organized as follows. The rest of this section summarizes (i) the people that have worked in the project, (ii) the produced Master's Theses and Student Project Reports, (iii) the scientific talks, research visits, and other scientific activities made in the project and (iv) the main deliverables produced in the project. A brief overview of the project and the architecture of the developed tools is given in Section 2. The results and developed techniques are explained in more detail in Sections 3-6, corresponding to the Tasks 1-4 in the Project Plan.

1.1 Personnel

The following people have been working in the project at Helsinki University of Technology.

- Ilkka Niemelä, responsible leader, October 2007 - September 2009.
- Keijo Heljanko, research leader; project manager, part-time October 2007 - August 2008.
- Kari Kähkönen, research assistant, full-time October 2007 - August 2008; project manager, full-time September 2008 - September 2009
- Lauri Harpf, research assistant, full-time June–August 2008
- Janne Kauttio, research assistant, full-time June–August 2008; part time September 2008 - May 2009; full-time June–September 2009
- Jani Lampinen, research assistant, full-time October 2007 - September 2008
- Sami Liedes, research assistant, full-time June–August 2008; part time September 2008 - May 2009; full-time June–August 2009; part-time September 2009
- Karoliina Oksanen, research assistant, part-time June 2009,
- Olli Saarikivi, research assistant, full-time June–August 2009, part-time September 2009

- Xi Chen, research assistant, full-time June–August 2009.
- Arto Vuori, research assistant, full-time June–September 11, 2009.

The following people have been working in the project at Åbo Akademi University.

- Johan Lilius, responsible leader, October 2007 - September 2009.
- Ivan Porres, research leader, October 2007 - September 2009.
- Emil Auer, research assistant, 15 January 2008 - December 2008.
- Sam Grönblom, research assistant, November 2007 - November 2009.
- Petter Holmström, research assistant, part-time September 2008 - May 2009, full-time June 2009 - September 2009.
- Sören Höglund, research assistant, October 2008 - September 2009.
- Sebastien Lafond, research assistant, 15 February 2009 - 15 July 2009.
- Ye Liu, research assistant, January 2009 - September 2009.
- Mats Lövdahl, research assistant, August 2008 - September 2009.
- Niclas Snellman, research assistant, full-time 15 May 2008 - August 2009, part-time September 2009.
- Johan Selänniemi, research assistant, part-time April 2009 - May 2009, full-time June 2009 - August 2009.
- Leif Sirén, research assistant, part-time 15 March 2009 - April 2009, full-time May 2009 - September 2009.
- Andreas Åkesson, research assistant, full-time 15 May 2008 - August 2009, part-time September 2009.

1.2 Theses and Student Project Reports

The following academic theses and student project reports have been produced in the project.

- The Master's Thesis of Jani Lampinen [22] has been accepted at the Department of Information and Computer Science at Helsinki University of Technology (TKK) in May 2008. The topic of the thesis is the LIME interface specification method for software components.

- The Master's Thesis of Kari Kähkönen [17] has been accepted at the Department of Information and Computer Science at Helsinki University of Technology (TKK) in August 2008. The topic of the thesis is automated test case generator developed in the project.
- The Master's Thesis of Emil Auer [1] has been accepted at the Department of Information Technology at Åbo Akademi in December 2008. The topic of the thesis is the generation of LIME interface specifications from UML protocol state machines.
- The Master's Thesis of Sam Grönblom [8] has been accepted at the Department of Information Technology at Åbo Akademi in February 2009. The topic of the thesis is the experimental comparison of unit testing and inspections.
- The Master's thesis of Mats Lövdahl [26] has been accepted at the Department of Information Technology at Åbo Akademi in 2009. The topic of the thesis is the generation of LIME and JML interface specifications from UML protocol state machines with invariants.
- The Master's thesis of Sören Höglund [13] has been accepted at the Department of Information Technology at Åbo Akademi in 2009. The thesis describes an evaluation of different unit testing tools based on the LIME and JML interface specification languages.
- The Bachelor's Thesis of Janne Kauttio [20] has been accepted at the Department of Information and Computer Science at Helsinki University of Technology (TKK) in May 2009. The topic of the thesis is the C language version of the LIME interface specification language and its implementation.
- A report by Olli Saarikivi [29], describing the fitness based search heuristic that is used to guide the LIME Concolic Tester towards execution paths that violate given LIME interface specifications, has been accepted as the deliverable for the course "T-79.5001 Student Project in Theoretical Computer Science" at the Department of Information and Computer Science of Helsinki University of Technology (TKK).

1.3 Talks, Research Visits and Other Scientific Activities

The following lists the conference participations and research visits made in the project.

Helsinki University of Technology:

- On February-July 2008 Jori Dubrovin visited the Embedded Systems Research Unit (formerly IRST-SRA) of the Bruno Kessler Foundation in Trento, Italy. The research unit led by Dr. Alessandro Cimatti has substantial experience in designing methods and tools for the development and verification of embedded systems in industrial settings. The main research topic of the visit was

the efficient application of *abstraction* methods on model checking concurrent systems. Abstraction is a key technique in enhancing the scalability of model checking methods. This work contributes to Task 1.3 and Task 2.3, where such methods are planned to be used for guiding the generation of relevant test cases.

- On February 14-15, 2008 Keijo Heljanko participated in a Artemis Brokerage event in Düsseldorf, Germany

- On June 23-27, 2008 Jori Dubrovin participated in 8th International Conference on Application of Concurrency to System Design (ACSD 2008), Xi'an, China and presented the paper " Symbolic model checking of hierarchical UML state machines" by Jori Dubrovin and Tommi A. Junttila.

- On March 29 - April 6, 2008, Kari Kähkönen and Keijo Heljanko visited The European Joint Conferences on Theory and Practice of Software (ETAPS) and its satellite events Foundations of Interface Technologies (FIT), Fourth Workshop on Model-Based Testing (MBT) and 8th International Workshop on Runtime Verification (RV).

- On May 26 - 30, 2008, Kari Kähkönen visited the 15th International Symposium on Formal Methods (FM 2008) held in Turku, Finland.

- On June 4-6, 2008 Jori Dubrovin participated in the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2008), Oslo, Norway and presented the paper "Symbolic Step Encodings for Object Based Communicating State Machines" by Jori Dubrovin, Tommi A. Junttila and Keijo Heljanko.

- On October 21-23, 2008 Keijo Heljanko participated in the Artemis and ITEA2 co-summit in Rotterdam, the Netherlands.

- November 22-27, 2008 Jori Dubrovin participated in the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2008), Doha, Qatar and presented the paper "Encoding Queues in Satisfiability Modulo Theories Based Bounded Model Checking" by Tommi A. Junttila and Jori Dubrovin.

- On April-September, 2009 Matti Järvisalo visited the Institute for Formal Models and Verification at Johannes Kepler University Linz, Austria in April-September, 2009. The institute led by Prof. Armin Biere has a strong track record in developing automated verification techniques and the visit by Matti Järvisalo contributed to the task of developing automated test generation methods. In particular, Prof. Biere's group has recently developed a new powerful SMT solver approach supporting bit-vectors and bit-vector arrays and this approach has been used to enhance concolic testing techniques developed in the project.

- On June 26 - July 2, 2009, Kähkönen presented the paper “The LIME Interface Specification Language and Runtime Monitoring Tool” by Kähkönen, Lampinen, Heljanko and Niemelä in the Runtime Verification workshop (RV 2009) held in conjunction with the Computer Aided Verification (CAV) conference in Grenoble, France.
- On June 8-9, 2009, Keijo Heljanko participated in the Artemis Summer Camp 2009, Brussels, Belgium.
- On June 30-July 3, 2009 Matti Järvisalo participated in the Twelfth International Conference on Theory and Applications of Satisfiability Testing, Swansea, Wales, United Kingdom
- On September 25, 2009 Keijo Heljanko gave a presentation of the LIME Interface Test Bench and the associated methodology developed in LIME with a talk ”Automated Testing of Interfaces in Component Based Embedded Systems” at the Tekes Ubicom programme result seminar held in association with the Elkom 2009 fair ECT Forum Embedded Software development session. The seminar was open to the general public.

Åbo Akademi:

- On February 13-14 2008, Ivan Porres visited Prof. Lionel Briand at the Simula Research Center, Oslo. Prof. Briand has a large experience on the evaluation of test methods and provided guidance for the Task 3.
- During June - November 2008, M.Sc. Torbjörn Lundkvist visited Technische Universität München in Germany to collaborate with Dr. Bernhard Schätz in his research group. Dr. Schätz is a merited researcher and model-based development of embedded systems for automotive applications is one of his main research interests.

1.4 Deliverables

The following briefly lists the main deliverables produced in the project. For more details, see Sections 3-6. The developed software and the research reports are available at `http://lime.abo.fi`.

- The LIME Interface Specification Language and the supporting monitoring tool has been described in [23]. The research report also describes the support for C programming language and partially implemented systems. This report is a deliverable produced as part of Task 1.
- The automated test case generation method has been described in [18]. The report is a deliverable for Task 1.

- A method to guide automated test generation based on specifications written with LIME Interface Specification Language is described in [30]. This report has been produced as part of Task 1.
- An approach to generate LIME interface specifications from UML protocol state machines containing state invariants and a supporting tool [15]. The tool also contains an interface to a UML modeling tool and can execute the LIME testing tools directly and report code coverage. The report and tool have been produced as part of Task 1.
- A tool to monitor LIME Interface specifications has been implemented for both Java and C programs [24]. The tool is a deliverable for Task 2.
- A tool to automatically generate test cases for sequential Java programs has been implemented [19]. The tool is a deliverable for Task 2.
- A tool to generate JUnit tests based on the test cases generated by LCT has been implemented [19]. The tool has been produced as a part of Task 2.
- A graphical user interface has been implemented to provide access to the tools developed in the LIME project via a common front-end [5]. The graphical user interface was implemented as part of Task 2.
- A report on the approach to evaluate the LIME interface specification language and its associated tools [11]. The report is a deliverable for Task 3.
- A report describing an evaluation of the LIME Specification Language [10]. The report is a deliverable for Task 3.
- A report describing a controlled Experiment Comparing Inspection and Testing of Embedded Software [9]. The report is a deliverable for Task 3.
- A report evaluating the LIME testing tools and similar tools, specially JML tools [14]. The report is a deliverable for Task 3.

2 An Overview of Tasks and Objectives of the Project

The project was divided into four tasks described in more detail below. Task 1: Design Methods and Languages, Task 2: Automated Validation Methods, Task 3: Research Drivers and Research Evaluation; and Task 4: Preparation of ARTEMIS proposal.

In Task 1 the idea was to develop a lightweight interface specification method which is easy to integrate into a normal design flow where the system under validation is already (partially or fully) implemented in source code level. In Task 2 the goal was to develop validation techniques, in particular, automated testing methods for such specifications.

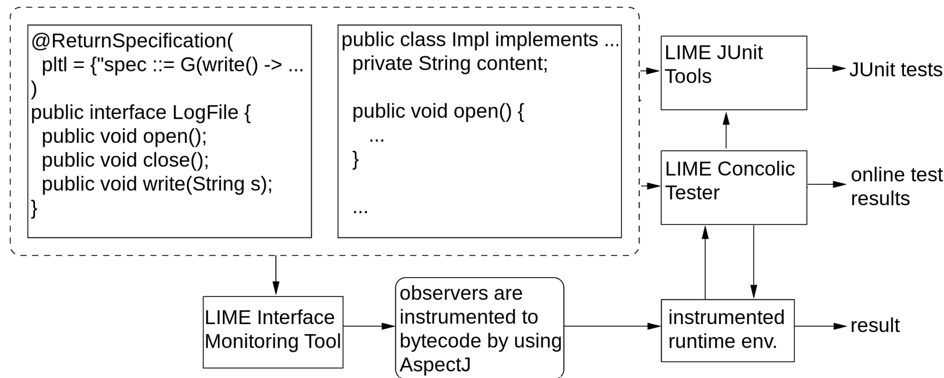

Figure 1: Architecture of the LIME Interface Test Bench

Task 3 comprised an evaluation of the state of the art in lightweight approaches for software validation. Finally, the objective of Task 4 was to create a proposal for the ARTEMIS call for an European project with similar goals as LIME.

To get an overview of the tools developed in the project as part of Task 1 and 2, Figure 1 shows the outline of the tools. This toolset is called LIME Interface Test Bench (LimeTB).

The starting point when using LimeTB is the system under validation, e.g. a software system composed of Java classes. Each class is augmented with a set of specifications (in form of code annotations) that describe both (i) how the class interface should be used by its clients and (ii) how the class itself should behave. The specification language used by LimeTB was developed in Task 1.

The specifications do not need to completely cover the behavior of the class. In addition, (the methods in) the classes are not necessarily implemented at all, or are only partially implemented. Each specification in the classes is then translated into an *observer*, i.e., Java code that observes whether the specification is violated during the execution of the system. Together with a test case input (e.g. a sequence of user actions), the system code augmented with the observers can then be executed in a run-time environment resulting in an assertion violation and error trace if a specification was violated on that input. The test case inputs needed can be provided by the developers/testers of the system or an automatic test case generator can be applied. In Task 2 such a test case generator (LIME Concolic Tester) was developed and it integrates with the observers generated from the specifications.

3 Task 1: Design Methods and Languages

The goal of this task was to develop lightweight methods to support interface specifications in an embedded system. The initial design was based on ideas from the hardware community such as the use of IEEE 1850 Property Specification Language (PSL), as well as Java/ESC and the Java Modeling Language (JML) from the soft-

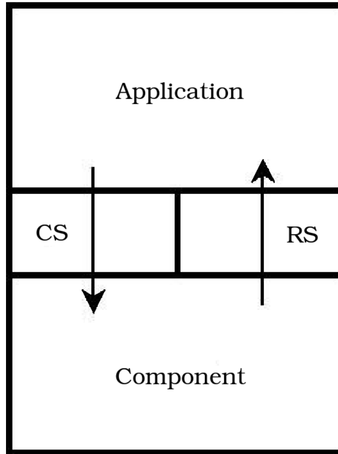

Figure 2: The interaction model

ware specification side. The aim was to ease the introduction of formal methods into a conventional design flow in a stepwise manner.

3.1 Task 1.1: Lightweight Interface Specification Method for Full Source Code

The aim in this task was to design a lightweight behavioral interface specification method including use specifications and pre/post conditions (assumptions/guarantees) with tentative deployment of a subset of the PSL language suitably modified for software use.

The LIME interface specification language (LIME ISL) developed in the project is a lightweight formal method for defining behavioral interfaces. The approach is supported by implementations for Java and C developed in Task 2 to monitor whether the specified interface specifications are violated. The core idea of the LIME interface specification language is to provide a declarative mechanism for defining how different software components can interact through interfaces in a manner that can be monitored at runtime. These interactions can be specified in two ways: by *call specifications* (CS) which define how components should be used and by *return specifications* (RS) which define how the components should respond. If a call specification is violated, the calling component can be determined to be incorrect and, respectively, if the called component does not satisfy its return specifications, it is functioning incorrectly. This interaction model between components is illustrated in Fig. 2.

The aim of the LIME ISL design was to enable a convenient way for the specification of behavioral aspects of interfaces in a manner that can be efficiently supported by tools. We have extended the *design by contract* [27] approach to software development supported by approaches such as the Java Modeling Language (JML) [4] to behavioral aspects of interfaces. The idea is to divide the component interface to

two parts in an assume/guarantee fashion: (i) call specifications (component environment assumptions) that specify requirements for the allowed call sequences to a software component and (ii) return specifications (component behavior guarantees) that specify the allowed behaviors of the component instance.

Another source of inspiration for the design of the LIME ISL has been the rise of standardized specification languages in the hardware design community such as IEEE 1850 - Property Specification Language (PSL) [16]. One of the key features of PSL is the inclusion of both temporal logic LTL as well as regular expressions in the specification language provided for the user.

Both the call and return specifications can be expressed in several different ways: as past time linear temporal logic (LTL) formulas, as (safety) future LTL formulas, as regular expressions, and as nondeterministic finite automata. On the syntactical level the interface specifications are annotations in Java and comments in a special syntax in the C code variant. The interface specifications consists of a set of atomic propositions definitions and the actual call and return specifications. Atomic propositions are used to make claims about the program execution and the state of the program. These atomic propositions are subdivided into three classes: *value propositions*, *call propositions* and *exception propositions*. There is also support for primitive data handling through the keyword `#pre` which makes it possible to reference an *entry* value in return specifications after the actual method has been executed. This allows specifications that describe how some value must change during execution of the observed method.

The final design of the LIME interface specification language can be found in [23] and a workshop paper describing the approach [21] has been presented at Runtime Verification (RV2009) workshop.

We have developed an approach to generate LIME specifications from UML protocol state machines and developed a tool support the approach. The supported input format is a UML 2.0 or 2.1 state machine represented in XMI 2.1. The tool produces LIME specifications in the form of a finite automaton. This allows a developer to use the UML as an interface specification language and benefit from the testing tools developed in Task 2. AAU has also evaluated the use of UML Sequence Diagrams in the context of the LIME project, but the results were not encouraging. This work is documented in the report [2]

3.2 Task 1.2: Extending the Method for Partially Implemented Systems

One aim in Task 1 was to extend the interface specification method to cope with systems which are only partially implemented or even with system designs with no actual implementation. Additional goal in this task was to enhance the interface specification method with new techniques that allow better handling of different aspects in interface specifications.

The support for partially implemented systems was developed by extending the interface specification method with automated stub code generation. The main idea

in this approach is to use the interface specifications to close a system from either top (call specifications) or bottom (return specifications) by generating stub code from the component interface. The stub code in the case of closing a system from top consists of a non-deterministic program whose call sequences to the underlying component that implements the interface is restricted to those allowed by the specification in a generate and test fashion. The approach for closing systems from below is similar, i.e., the stub code will return values nondeterministically that are restricted by the return specifications. The restriction to valid call sequences and argument/return values is achieved by making a test verdict to be inconclusive if the stub code causes a specification to be violated.

As randomly generated call sequences and return values will likely lead to many inconclusive test runs, the stub code approach is intended to be used mainly with the automated test case generator developed in Task 1.3 and Task 2.3. The use of the test case generator makes it possible to compute valid input arguments to the method calls and to make sure that the same call sequences are not tested repeatedly. The stub code generation is described in more detail in [23, 21].

To enhance the LIME interface specification language it was extended to support propositions on exception occurrence, as exceptions are commonly in a critical role in Java code and, thus, need to be supported in order to be able to properly specify the allowed behavior of the program. The extension allows specifying propositions to be used in monitoring which are true if and only if a certain kind of exception, as specified by the user, is thrown by the method being monitored. This proposition can then be used as part of conditions in any of the supported forms of specifications, i.e., regular expression, PLTL and NFA specifications. The full details on exception propositions are described in [23].

In order to provide better support to data aspects in the interface specification we have extended the UML to LIME approach described in Task 1.1 to support UML protocol state machines with state invariants.

We have implemented tool support to translate UML protocol statemachines with invariants into LIME specifications containing pre and postcondition specifications. The tool can also generate JML specifications and run the LIME and JML testing tools automatically. This was done to support Task 3.4. The applicability of the tool is demonstrated with examples ranging from simple to complex applications. We have also developed a plugin for a commercial UML modeling tool (MagicDraw) that integrates the specification generation tool and the testing tools. This plugin is an example on how the LIME project results can be integrated with existing UML modeling tools. This work is documented in the report [15]

3.3 Task 1.3: Enhancing the Method with Automated Test Case Generation

The goal in this task was to design techniques for automated test case generation that can be used together with the interface specification method developed

in Task 1.1 and Task 1.2. The aim was to combine testing with symbolic model checking methods.

The automated test case generation was designed to be based on concolic testing [7, 31]. In concolic testing a program is executed both concretely and symbolically at the same time. The main idea behind this approach is to first execute the program under test with random concrete input values and then during execution to collect symbolic constraints about input values that would force the program to follow a different execution path (i.e., to follow a different outgoing branch at some branching statement). All the execution paths of a given program can be expressed as a symbolic execution tree where the nodes represent branching points in execution that have symbolic constraints associated to them. Concolic testing can be seen as a method to generate a symbolic execution tree and use it to systematically test different execution paths of a program. The input values needed to explore the different execution paths are obtained by solving the collected constraints using off-the-shelf SMT-solvers. The details on the test case generation using concolic techniques can be found from [18].

The concolic testing approach was also extended to take LIME interface specifications into account when generating test cases. The approach here is based on computing a heuristic value of how close a specification is to being violated from the finite state automata of the runtime observers. This heuristic value is computed for each unexplored execution branch created by the LIME test generation tool and the execution branches that are the most likely to lead to a specification violation are explored first. The computation of the heuristic values is based on sampling random walks on the finite state automata. More detailed description of computing the heuristic values can be found from [30].

4 Task 2: Automated Validation Methods

The focus of this task was to implement prototype tools to support the methods developed in Task 1.

4.1 Task 2.1: Prototype Tool Framework for Interface Specifications

The LIME Interface Monitoring Tool (LIMT) is the Java version of a monitoring tool for the LIME ISL developed in Task 1. It allows monitoring the specifications at runtime to determine if some component violates the given specifications. An architectural overview of the toolset is given in Fig. 1.

The monitoring tool works by reading the specification annotations from the Java source files. The specifications are then translated into deterministic finite state automata that function as observers. These automata are translated into runnable Java code and AspectJ (`http://www.eclipse.org/aspectj/`) is used to weave the code into the original program that is being tested. This results in an instrumented

runtime environment where the observers are executed at the timepoints discussed in the previous section.

Spoon [28], the dk.brics.automaton (`http://www.brics.dk/automaton/`) package and SCheck [25] are adopted as third-party software. Spoon is used for analyzing the program and the dk.brics.automaton package is used for internal representation and manipulation of regular expression checkers. SCheck is used for converting future time LTL subformulas into finite state automata. The approach of [12] using synthesized code with history variables is used for past time subformulas, while for the future part the tool SCheck is used to encode informative bad prefixes [25] of future LTL formulas to minimal DFA. The implementation allows the use of past-time subformulas LTL in future-time LTL formulas but not vice versa. More technical details of the implementation approach taken for Java can be found from [23, 22].

4.2 Task 2.2: Adding Support for the C Programming Language

The C code variant of the LIME interface specification language has also been implemented sharing much of the source code with the Java variant. The main differences in the C language version to the Java version are: (i) comments in special syntax are used to define the specifications instead of Java annotations, (ii) instead of being part of an object creation in Java, in C monitor instances have to be explicitly associated to an interface, (iii) Doxygen tool (`www.doxygen.com`) is used (instead of Spoon for Java) to find the interface specifications from the C comments, (iv) AspeCt-oriented C (ACC, `http://research.msrg.utoronto.ca/ACC`) is used to weave the monitors into the C source code; (v) there is no support for Java specific features such as exceptions. The C variant, of course, generates the weaved monitoring code in C but is otherwise functionally identical to the Java version.

More technical details of the implementation approach taken for C can be found from [23, 20].

4.3 Task 2.3: Test Case Generator

The automated test case generation method developed in Task 1.3 was implemented in tool called LIME Concolic Tester (LCT) that supports Java as the target programming language. Concolic execution of programs requires that the program under test is instrumented with additional code that allows the program to be symbolically executed. For the instrumentation the Soot optimization framework [32] was adopted as an external tool. To solve the constraints resulting from symbolic execution, support for Yices [6] and Boolector [3] was implemented. Due to licensing issues, the solvers are not included in the tool and must be downloaded separately.

Typical Java Virtual Machines cause the following technical limitation in the instrumentation: core Java classes (e.g., java.lang.String and java.lang.Integer) cannot be modified freely and therefore they cannot always be fully instrumented by LCT.

The reason for this limitation is that most Java Virtual Machines are very sensitive to modifications in the core system classes (e.g., the load order of classes during bootstrapping may change due to instrumentation) and this can cause the JVM to crash. Furthermore, instrumenting the core classes means that the instrumented code that also uses core classes would use their rewritten versions which can cause complications.

To solve this limitation custom versions of Integer, Short, Boolean and Byte classes were implemented so that they can be instrumented. The system under test is modified to use these custom classes instead of the original Java core classes. The instrumentation of other core classes is not currently supported and if such classes are used in the system under test, the test generator may fail to explore all possible execution paths. The supported subset of core classes is, however, sufficient to make the test generation method work properly together with the monitoring tool. A more details description of the limitations regarding core classes can be found from [18].

The basic test case generation provided by LCT was extended by developing LIME JUnit Tools (LJUT) that provide the possibility to automatically generate test drivers to unit test methods and to generate JUnit test cases that allow replaying the unit tests even without LCT.

4.4 Enhancing the Usability of the Developed Tools

The tools included in the LIME Interface Test Bench were primarily designed to be used from the command-line. To make the tools more accessible for new users, a graphic user interface was developed for the tools.

The GUI was developed as a stand-alone application and therefore it does not force the developers to use any specific programming framework. The GUI allows the user to configure the LIME tools and to perform the basic tasks provided by the tools.

To help the user of LimeTB to understand and debug the cause of specification violations, the LIME Interface Monitoring Tool was extended with tracer functionality. The generated observers write extensive information about their state to a log file during the execution of the instrumented program. This logging functionality is enabled by default, but can be optionally disabled during the compilation of the program either on a per-specification basis or completely. The log file can then be read with the provided tracer-utility, which is a command-line tool that has also been integrated into the graphical user interface. The tracer parses the log file and shows the user the execution of the program from the point of view of the observers, which will make finding the reason of a possible specification violation easier.

5 Task 3: Research Drivers and Research Evaluation

The goal for task 3 was to evaluate and provide a feedback loop for the work performed in the other tasks. We have produced four reports [11, 10, 9, 14].

5.1 Task 3.1 Selection of Case Studies

The objective of this task was to organize the evaluation effort in the LIME project. We developed an evaluation criteria for the LIME specification language and LIME testing tools and introduced different software systems that are used in the evaluation.

The initial goal was to select two case studies in cooperation with the companies that are participating in the project. However, we only obtained one case study from the industry partners. Based on the recommendations from the Steering Group and our experiences using LIME, we decided to add two more systems equipped with formal interface specifications to be used in our evaluation: The Sun Java Card API and the JML contracts for this API created by Wojciech Mostowski and a collection of simple Java classes distributed with the JML tools.

The results of this task are reported in [11].

5.2 Task 3.2 Case Study Evaluation Using Standard Methods

The objective of this task was to evaluate the case studies selected in Task 3.1 using standard methods such as software inspections and unit tests created manually.

We carried out this task by conducting an experiment to compare which method finds more defects in a single session. In our study 22 graduate and undergraduate students performed each method on separate programs. We also tried to distinguish between merely finding a defect and also discovering the fault in the code that caused the defect. The results indicate that given a short amount of time for the task, inspectors find significantly more faults than testers.

The results of this task are reported in [10].

5.3 Task 3.3 and 3.4: Case Study Evaluation Using the Methods Developed in the LIME Project

This task had two phases: evaluation of the LIME specification language and the evaluation of the LIME tools. The evaluation was performed by using the case studies from Task 3.1 and the same set of metrics as in Task 3.2 and comparing LIME to other approaches and tools with the same objectives. This task produced two reports.

The first report [9] describes our observations on using the LIME specification language to describe software interfaces in three case studies. We analyzed which software interfaces from the case studies can be described in the LIME specification language and provide some recommendations for the future development of LIME. We found that the LIME specification language could be used to specify many properties of the interfaces present in the case studies. However, we also produced

some recommendations on future extensions to the LIME language that can help increasing its applicability.

The second report [14] contains a mutation testing experiment evaluating the performance (efficiency, effectiveness and statement coverage) of several specification-based tools. The tools studied include JML-JUnit, JET, ESC/Java2, UnitTesk, LIME Interface Monitoring Tool and LIME Concolic Tester. The evaluation revealed different issues in the LIME tools that were promptly fixed. Although it is too early to perform a complete evaluation of the LIME Concolic tester, the results are encouraging. The test suite is available to evaluate the future development of the LIME tools.

6 Task 4: Preparation of ARTEMIS proposal

The Reduced Certification Costs Using Trusted Multi-core Platforms (RECOMP) project proposal was submitted on time the 3rd for September.

The proposed RECOMP research project will establish methods, tools and platforms for enabling cost-efficient certification and re-certification of safety-critical systems and mixed-criticality systems, i.e. systems containing safety-critical and non-safety-critical components.

RECOMP will provide reference designs and platform architectures together with the required design methods and tools for achieving cost-effective certification and re-certification of mixed-criticality, component based, multi-core systems. The aim of RECOMP is to define a European standard reference technology for mixed-criticality multi-core systems supported by the European tool vendors participating in RECOMP.

The partners of the consortium cover several domains (automotive, industrial automation, lifts, avionics, health care and building automation), ensuring that the developed technology will be adoptable in several domain. The consortium also includes companies that will exploit the methodology for selling sub-contracting. The RECOMP project will bring clear benefits in terms of cross-domain implementations of mixed-criticality systems in all domains addressed by project participants: automotive systems, aerospace systems, industrial control systems, lifts and transportation systems.

RECOMP in Finland

RECOMP has 7 participants from Finland: Åbo Akademi, Metso, Space System Finland, the Helsinki University of Technology, Kone, Spinet and Vacon. Kone, Metso and Vacon are proposing the implementation of an application demonstrator for the RECOMP project.

The total eligible costs for Finnish partners are around 4 million euro and represents 16% of the total project costs. The total funding requested from Tekes is around 1,6 million euro, and the total funding requested from the ARTEMIS JU is around 0,67 million euro.

RECOMP in numbers

- 41 participants from 8 countries: CZ, DK, ES, FI, FR, DE, IE, UK
- 31 companies: 9 small, 4 medium and 18 non-SME
- 10 Universities and research organisations
- Total eligible costs of 25 879 930 euro
- Requested national funding of 9 704 226 euro
- Requested ARTEMIS JU contribution of 4 320 945 euro
- Total effort of 2 633 person months
- Proposal of 406 pages including 188 pages of annexes

References

[1] Emil Auer. Generation of LIME specifications from UML protocol state machines. Master's thesis, Åbo Akademi University, Department of Information Technologies, 2009.

[2] Emil Auer and Ivan Porres. SM2LIME: A translation tool from UML state machines to LIME specifications, 2008. LIME project deliverable.

[3] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In Stefan Kowalewski and Anna Philippou, editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer, 2009.

[4] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[5] Xi Chen, Janne Kauttio, and Olli Saarikivi. A graphical user interface for LimeTB, 2009. Computer program in LimeTB 1.0.0 software release.

[6] Bruno Dutertre and Leonardo de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification conference*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.

[7] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[8] Sam Grnönblom. A controlled experiment comparing code reviews and testing of embedded software. Master's thesis, Åbo Akademi University, Department of Information Technologies, 2009.

[9] Sam Grönblom and Ivan Porres. A controlled experiment comparing inspection and testing of embedded software, 2008. LIME project deliverable.

[10] Sam Grönblom and Ivan Porres. Preliminary evaluation of the LIME specification language, 2008. LIME project deliverable.

[11] Sam Grönblom and Ivan Porres. Research drivers and research evaluation in the LIME project, 2008. LIME project deliverable.

[12] Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *Software Tools for Technology Transfer (STTT)*, 6(2):158–173, 2004.

[13] Sören Höglund. An empirical evaluation of specification-based unit testing tools. Master's thesis, Åbo Akademi University, Department of Information Technologies, 2009.

[14] Petter Holmström, Sören Höglund, Leif Sirén, and Ivan Porres. Evaluation of specification-based testing approaches, 2009. LIME project deliverable.

[15] Petter Holmström, Ye Liu, Mats Lövdahl, Irum Rauf, Johan Selänniemi, Leif Sirén, and Ivan Porres. Generation of behavioral interface specifications from UML protocol state machines with state invariants, 2009. LIME project deliverable.

[16] IEEE-Commission. IEEE standard for property specification language (PSL). Technical report, IEEE, 2005. IEEE Std 1850-2005.

[17] Kari Kähkönen. Automated dynamic test generation for sequential Java programs. Master's thesis, Helsinki University of Technology, Department of Information and Computer Science, 2008.

[18] Kari Kähkönen. Automated test generation for software components. Technical Report TKK-ICS-R26, Helsinki University of Technology, Department of Information and Computer Science, Espoo, Finland, December 2009. LIME project deliverable.

[19] Kari Kähkönen. LIME Concolic Tester, 2009. Computer program in LimeTB 1.0.0 software release.

[20] Janne Kauttio. *LIME-C – Rajapintamäärittelymenetelmä C-kielisille ohjelmille*. Bachelor's thesis, Helsinki University of Technology, Department of Information and Computer Science, 2009.

[21] Kari Kähkönen, Jani Lampinen, Keijo Heljanko, and Ilkka Niemelä. The LIME interface specification language and runtime monitoring tool. In *Proceedings of the 9th Internatinal Workshop on Runtime Verification*, volume 5779 of *Lecture Notes in Computer Science*, 2009. to appear.

[22] Jani Lampinen. Interface specification methods for software components. Master's thesis, Helsinki University of Technology, Department of Information and Computer Science, 2008.

[23] Jani Lampinen, Sami Liedes, Kari Kähkönen, Janne Kauttio, and Keijo Heljanko. Interface specification methods for software components. Technical Report TKK-ICS-R25, Helsinki University of Technology, Department of Information and Computer Science, Espoo, Finland, December 2009. LIME project deliverable.

[24] Jani Lampinen, Sami Liedes, Janne Kauttio, Lauri Harph, Olli Saarikivi, and Kari Kähkönen. LIME Interface Monitoring Tool, 2009. Computer program in LimeTB 1.0.0 software release.

[25] Timo Latvala. Efficient model checking of safety properties. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, volume 2648 of *Lecture Notes in Computer Science*, pages 74–88. Springer, 2003.

[26] Mats Lövdahl. Automatic generation of JML and LIME specifications from UML protocol state machines. Master's thesis, Åbo Akademi University, Department of Information Technologies, 2009.

[27] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[28] Renaud Pawlak, Carlos Noguera, and Nicolas Petitprez. Spoon: Program Analysis and Transformation in Java. Research Report RR-5901, INRIA, 2006.

[29] Olli Saarikivi. Design and implementation of a heuristic for directing dynamic symbolic execution, 2009. A student project deliverable made in the LIME project.

[30] Olli Saarikivi. Design and implementation of a heuristic for directing dynamic symbolic execution, 2009. LIME project deliverable.

[31] Koushik Sen and Gul Agha. CUTE and jCUTE : Concolic Unit Testing and Explicit Path Model-Checking Tools. In *18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006.

[32] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In Stephen A. MacKay and J. Howard Johnson, editors, *CASCON*, page 13. IBM, 1999.