Software Qual J (2014) 22:185–213
DOI 10.1007/s11219-012-9193-8

Using aspects for testing of embedded software: experiences from two industrial case studies

Jani Metsä · Shahar Maoz · Mika Katara · Tommi Mikkonen

Published online: 23 January 2013
© Springer Science+Business Media New York 2013

Abstract Aspect-oriented software testing is emerging as an important alternative to conventional procedural and object-oriented testing techniques. This paper reports experiences from two case studies where aspects were used for the testing of embedded software in the context of an industrial application. In the first study, we used code-level aspects for testing non-functional properties. The methodology we used for deriving test aspect code was based on translating high-level requirements into test objectives, which were then implemented using test aspects in AspectC++. In the second study, we used high-level visual scenario-based models for the test specification, test generation, and aspect-based test execution. To specify scenario-based tests, we used a UML2-compliant variant of live sequence charts. To automatically generate test code from the models, a modified version of the S2A Compiler, outputting AspectC++ code, was used. Finally, to examine the results of the tests, we used the Tracer, a prototype tool for model-based trace visualization and exploration. The results of the two case studies show that aspects offer benefits over conventional techniques in the context of testing embedded software; these benefits are discussed in detail. Finally, towards the end of the paper, we also discuss the lessons learned, including the technological and other barriers to the future successful use of aspects in the testing of embedded software in industry.

J. Metsä
Elektrobit Inc., Bothell, WA, USA
e-mail: Jani.Metsa@elektrobit.com

S. Maoz
School of Computer Science, Tel Aviv University, Tel Aviv, Israel
e-mail: maoz@cs.tau.ac.il

M. Katara (✉) · T. Mikkonen
Department of Pervasive Computing, Tampere University of Technology,
Tampere, Finland
e-mail: mika.katara@tut.fi

T. Mikkonen
e-mail: tommi.mikkonen@tut.fi

Keywords Software testing · Aspect-oriented programming · Embedded software · Case studies

1 Introduction

The *raison d'être* for software is the fulfillment of its requirements. Requirements and their associated tests are tightly coupled together; in addition to finding defects, the main purpose of testing is to check that the implementation satisfies the requirements.

In industrial practice, testing functional and non-functional requirements may involve heavy instrumentation of the system under test (SUT). Furthermore, conventional modularization techniques used for implementing the tests suffer from a scattering of the implementations of *testing concerns*[1] to various components, where they are tangled up with other concerns. In the extreme, this leads to the inability to separate the additional software that is used for testing purposes from the SUT code itself. Scattering and tangling may suggest the emergence of problems concerning traceability, comprehensibility, maintainability, low re-use, high impacts of changes, and reduced concurrency in development (Clarke 1999). For the testing process, scattering and tangling problems imply added expenses and quality issues.

Toward solving these issues, *aspect-orientation* provides programming-level mechanisms that enable non-invasive changes to existing code (Filman et al. 2004). This provides separation of concerns by enabling the modularization of crosscutting concerns, in the context of this paper test code, the implementation of which would otherwise be scattered around different SUT components and tangled up with the SUT code. Such separation between test code and SUT code, through the use of aspect-orientation, carries several potential advantages. First, it facilitates maintenance tasks if (and when) the requirements (and the corresponding tests) need to be changed after analyzing the design trade-offs. Second, the correspondence between the requirements and test code becomes more explicit, resulting in better traceability, which is required for instance by safety standards. Third, it is trivial to remove the test aspects from the final product when necessary, as they are explicitly injected into the system only when needed (at compile time or run time). To take advantage of the above benefits, however, in a systematic, large-scale use of such techniques, we need to reconsider the relation between aspects and the requirements of the SUT.

In our previous work, we have studied the use of aspects for testing from the viewpoints of implementation (Pesonen et al. 2006), integration (Pesonen 2006), and improved testability (Metsä et al. 2008). This paper focuses on the viewpoints of non-functional testing and model-based testing and is an extended and revised version of two earlier conference papers, each introducing a case study (Metsä et al. 2007; Maoz et al. 2009).

The first case study described in this paper concentrates on non-functional testing using manually coded aspects. In the second case study, we used a model-based approach to automatically generate test aspects from visual models. Thus, although the two case studies share the same SUT and the general technique of using aspects for testing, they are also rather different. Toward the end of the paper, we discuss our conclusions from the two case studies, including the challenges we have identified for the industrial deployment of aspects in software testing.

[1] For the purposes of this paper, a concern is defined to be any conceptual matter of interest, that is, testing concerns are conceptual matters of interest related to testing, which are important to some stakeholder(s).

2 Background and related work

Software testing is tightly integrated into the software development process (Craig et al. 2002). In the traditional *V-model*-based development (Rook 1986), testing is planned during the waterfall phases from the requirements gathering to the implementation while the actual test execution takes place in the later process phases including the unit, integration, system, and acceptance testing. In *agile development*, on the other hand, testing is planned much less, but test design and execution take place much more frequently and earlier on in the software life cycle than in the V-model due to the iterations. Methods and techniques for agile testing include test-driven development at the unit level, continuous integration at the integration testing level as well as acceptance test-driven development and exploratory testing at the system level. Of these, the last one is a manual technique, while the former ones are based on tools automating the test execution [see, for instance (Kaner et al. 2002; Fewster et al. 1999)].

Regardless of the process used, important types of testing include so called *positive* and *negative* testing. The former aims to show that the implementation satisfies the requirements and use cases important to the end-users. These same artifacts also drive the design and coding, so positive testing is not sufficient for revealing defects. Hence, negative testing tries to test the robustness of the implementation by testing areas that are not specified or only vague addresses by the requirements and use cases in order to find defects. Techniques for negative testing include boundary value analysis [see, for instance (Pezzè et al. 2008)] used to derive test cases that check for “off by one” type of errors often made by developers, which can even contribute to memory leaks or security holes in the form of buffer overflows, for instance.

Aspect-oriented software development, on the other hand, provides means for capturing crosscutting concerns and modularizing them as manageable units (Filman et al. 2004). Approaches offering tool support at the code-level target in encapsulating implementation issues in non-tangling manner. In programming-level aspect-oriented techniques, such as AspectJ 2012, AspectC++ 2012, a *pointcut* is a collection of *joinpoints* and defines the condition for aspect to take effect. Joinpoints define when the aspect surfaces and are of type *call* or *execution*, for instance. The former is associated with the call to a certain code element, for example, a function call, and latter to the execution of a referred element, for example, the function, correspondingly. Consider for example the following simple aspect code that introduces a simple test case:

```
1: aspect SimpleTestCase {
2:   pointcut testedFunction = call("void MyClass::MyFunc(...)");
3:   advice testedFunction:around() {
4:     // trace function parameters
5:     tjp->Proceed(); // Run the function
6:     // trace results and test pass/fail
7:   }
8: }
```

Above, the pointcut is defined as a call joinpoint taking effect around the defined function of our example class. The test case aspect takes effect as the function is called, and the aspect code is executed both before and after actually executing the original function.

Using aspect-oriented techniques and tools to enhance requirements definition and testing has been addressed in prior work. For example, aspect-orientation has been studied in the context of system design by a number of researchers. Rashid et al. (2002) propose the Aspect-Oriented Requirements Engineering model (AORE), where stakeholders' requirements are used to identify crosscutting concerns at the early phases of development. The authors proposed aspects to be used to resolve conflicts in requirements and to identify the influence of crosscutting properties, thus improving the means to managing resulting tangled representations. The method aims to improve requirements of engineering and architecture analysis by advancing viewpoints with crosscutting issues, but does not address testing concerns, which is the main target in our approach. In addition, AORE is semi-formal and, to the best of our knowledge, has not been used to specify or generate concrete running tests.

Another branch of research has been to use aspects to enhance software testing, but almost only at the unit testing level. For example, studies by Stamey and Sounders (2005) and by Kulesza et al. (2005) use aspects to extend unit tests in the Java environment. Aspects have been used also as an assistant in monitoring unit testing (Coelho et al. 2006); in this work by Coelho et al., aspects were not used in the actual test cases but only to measure the test efficiency or coverage. In addition, only a limited set of possible uses of aspects in testing was proposed.

Rajan and Sullivan (2005) used aspects to analyze concern coverage. The goal of this work was to use aspects to express the adequacy criteria in a more declarative manner. Although this profiling approach resembles our idea of capturing concerns and using them in testing, their tool focuses on the code level to enhance white-box coverage analysis. In our approach, aspects are used to expose new concerns for testing, when creating test objectives together with other system requirements.

Feng et al. (2007) proposed generic non-functional unit test cases to be written using a product line approach where aspects are used for the actual implementation. In their approach, test cases are written using XML; joinpoint information is gathered manually from specifications or using a tool from component source codes. The test cases are formulated using generic core sections and variant sections defining component specialization. Furthermore, abstract test cases present a basis for all test cases, and final test cases are manually written according to the target system. Our approach is somewhat similar in the use of aspect inheritance and the specialization of the tests according to the SUT. However, their approach is limited to unit testing level and, compared with our approach, does not allow testing for system-wide concerns.

Xu et al. (2007) have proposed aspectual use cases for the generation of test requirements. In this approach, system test requirements are generated from the use cases by first transforming the use case diagrams into Petri nets. The resulting formalized system model is used to generate the related use case sequences. This technique relies on the formalization of the base system and aspectual use cases.

Kartal and Schmidt (2007) evaluated aspect-oriented programming in the context of an industrial real-time embedded system. Their evaluation concentrated on comparing an object-oriented implementation with an aspect-oriented one implemented with AspectC++. The evaluation was twofold: it was based on software quality (Chidamber&Kemerer metrics suite) and embedded real-time performance (memory usage, CPU usage, and worst-case execution time) metrics. The authors concluded that the aspect-oriented approach was better than the object-oriented solution based on both types of metrics. However, the case study described in the paper seems quite limited in size and domain, and thus, the results may not be generalizable as such.

To summarize the relation of this paper and related work, we have extended earlier research to the direction of non-functional requirements and model-based testing in a setting where several constraints regarding the testing context, environment, and infrastructure were predefined, as usual to the embedded software domain. Furthermore, we discuss the observations made during the case studies and recommend future steps that make aspects more suitable for industrial use.

3 Case study I

The first case study focused on requirements that could be used as a basis for aspect-oriented, non-functional testing. The aim was to identify requirements that propose crosscutting concerns in design solutions, compositions, structures, or implementations; in other words, requirements that manifest system-wide behaviors or characteristics. These requirements together with the existing implementation were used to define *test objectives*.

The software system we studied (also referred to as SUT in the following) is used for the quality verification of mobile phones: it is deployed in mobile phone manufacturing to ensure the composed devices are flawlessly manufactured. The system is a relatively small-sized industrial system with 200 k source code lines and about 100 kB of target binary. It is a messaging-based diagnostics application that works inside the device and provides certain verification-related functionalities, composed of a central core, proxies representing procedure abstractions, and the actual procedure implementations. A schematic overview of the system is provided in Fig. 1. It important to note that while the target of our study is used for the quality verification of hardware, it will be referred to as SUT in this paper because we applied aspect-oriented testing to it.

The basic framework for the software design was set by the context: a mobile phone running Symbian OS (Nokia 2012). This dictated the basic design constraints: C++ as programming language, Symbian OS coding conventions and design patterns to be followed, strict memory footprint and limited system resources, and binary messaging to be used as the communication method between the client and the application. From these system characteristics, we formulated the initial set of requirements the system should fulfill. These included performance considerations, robustness, modularity, and fault tolerance. Examples of system characteristics and the derived requirements are listed in Table 1.

The test set up before the first case study was such that the software testing framework used to test the SUT was based on *test cases* that provide the required test input. Moreover, *test control* sets the preconditions for test cases, deploys test data, and observes the outputs from the SUT. A simplified overview of the software test framework is illustrated in Fig. 2, where

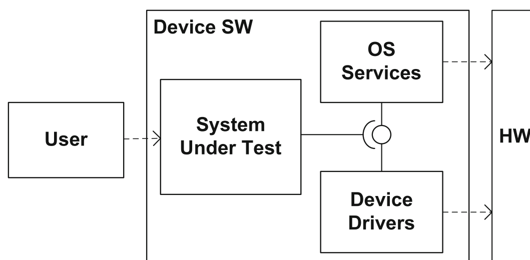

Fig. 1 A schematic overview of the software system under test

Table 1 Examples of system characteristics and derived requirements

System characteristic	Derived requirement
Mobile phones are treated as embedded systems and have limited amount of resources: Strict memory footprint.	REQ1: The software must occupy maximum of X bytes of ROM
	REQ2: The software must consume maximum of Y bytes of RAM
The manufacturing throughput must remain fast enough	REQ3: The system is able to respond to service requests after Z time units after power-up
	REQ4: All service requests for procedures are handled and completed in no more than the given time limit
The system must be fault tolerant and able to report on faulty hardware	REQ5: The system is able to recover from hung device drivers
	REQ6: Faulty hardware is identified and results as failed procedures
	REQ7: The system is able to identify correct and incorrect hardware behavior
	REQ8: Starvation and hang situations do not occur or there is a method to recover from these situations
Procedures cover all hardware	REQ9: The system is able to control all the hardware in the device
	REQ10: The system has interfaces to all peripherals and their device drivers or the lowest possible software layer toward hardware
Services can be requested outside the device using common methods	REQ11: The system provides a messaging interface for the client
The system can be utilized in all Symbian devices	REQ12: The system adheres to software product-line concept
	REQ13: Implementation is written in C++programming language
	REQ14: The system runs in Symbian OS
	REQ15: Symbian OS coding conventions and design patterns are followed
	REQ16: The system fulfills the Symbian OS platform security criteria and guidelines

the test software is marked in the gray color. In the Figure, the upper part illustrates test concepts at an abstract level and the lower part maps the concepts to the design of the SUT.

3.1 Analyzing requirements for testing

At a general level, the requirements present the expectations of stakeholders and are thus a starting point for the architecture design. Hence, in order to refine the test objectives, we analyzed the existing requirements so as to identify crosscutting issues and to find the tangled representations they proposed. We started by gathering implicit non-functional requirements set by the initial system requirements discussed above. These revealed that the system characteristics had crosscutting effects on the implementation that would be difficult or laborious to formulate and implement as test cases using conventional techniques.

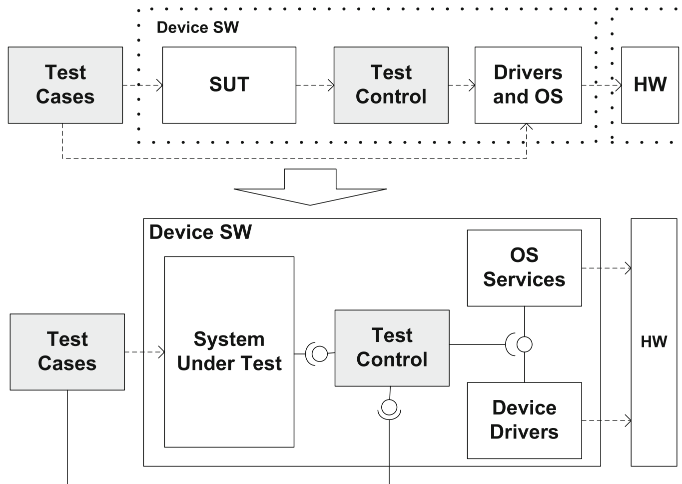

Fig. 2 Overview of the software test framework. The *gray color* denotes added test components

We limited our analysis to non-functional requirements since in the context of this particular SUT, existing conventional test methods provided sufficient tools for functional testing. Hence, we ignored the functional requirements, which where reconsidered in the second case study.

The non-tangling nature of aspects makes the technique an attractive choice for implementing most of the identified new test objectives. However, it seems that the ability to identify potential requirements and to derive the corresponding test aspects requires insight into the actual system structure and is dependent on the experience of the test designer. Although the requirements themselves are not explicitly tangled, we were able to identify tangled concerns when evaluating the requirements together. From the initial requirements, we were able to identify at least the following requirements tangled with each other:

- REQ3 sets constraints regarding performance on REQ4 and REQ11: Messaging is possible only after the system has been started up. No requests can be completed during the start-up due to the power-up delay set in REQ3.
- REQ4 in turn sets constraints on the performance of REQ5 and REQ8: In order to serve all service requests in the given period, all system lock states have to be resolved within this time limit.
- REQ6 and REQ7 set constraints regarding robustness or reliability on each other: Regardless of the correctness of the hardware assembly and the resulting system behavior, the system must handle results properly.
- REQ16 sets security constraints on REQ10 and REQ11: Security considerations affect the design and system composition by defining security rules that have to be followed in the final implementation.

Following the analysis, the resulting test objectives are listed in Table 2. From the initial requirements (Table 1), we identified seven test objectives (Table 2) that present non-functional characteristics that potentially benefit from aspect-oriented testing techniques. As the system design evolves and more details on the actual system design become

available, the number of requirements, and in consequence also the number of test objectives and related test cases, may increase significantly.

Our test objectives ignored most of the portability and variability issues represented by requirements REQ12 to REQ16. We decided that these would be out of the scope of the test code, since the testing was to be performed at the system testing phase by executing the test cases.

3.2 Derived test aspects

Based on the new test objectives, we identified testing concerns, which were then implemented as test aspects (Table 3). The derived test aspects present testing concerns that require long-term observation as well as the injection of complex tracking mechanisms. Performance profiling and coverage aspects require the collection of information on all functions, related execution times, and service requests, which could be laborious to implement. However, profiling and coverage both depend on crosscutting behaviors that are easy to capture as pointcuts and aspects.

Furthermore, since in C++, there is neither garbage collection nor built-in methods for taking care of memory usage, a run-time memory monitoring mechanism could be laborious to implement without aspects. Our memory aspect captures each call to memory allocation and deallocation, and thus tracks the ownership and lifetime of every memory element. Again, weaving the monitoring feature into the system is bound to memory allocation and deallocation calls, making it simple to formulate the aspect itself. Although defining the pointcut is non-trivial, it is reusable and portable.

Based on the analysis. we realized that a jamming mechanism was needed as a test aspect to check that the system fulfills requirement REQ5. Furthermore, requirements REQ3, REQ4 and REQ8 could be compromised by the services that get stuck at a hang situation. Hence, the robustness aspect developed for test objective TO5 was also utilized in the case of test objectives TO2, TO3, and TO4 in order to give proper insight into the system performance in these situations too. As the subsystems providing services to this particular system are identifiable already at the system design phase, we would be able to formulate the test aspect already at the requirements analysis phase by identifying the related interfaces. Thus, the test aspects can be defined already at the design phase without

Table 2 Test objectives for non-functional requirements

Requirement	Test objective	Requirement category
REQ2	TO1: Supervise memory consumption	Robustness
REQ3	TO2: Measure time from initial power on to the system being in responsive state	Performance
REQ3, REQ4	TO3: Measure time consumed on serving requests	Performance
REQ4	TO4: Track procedure executions and create execution time profiles	Profiling
REQ5	TO5: Generate hang situation on device driver	Robustness
REQ5, REQ6, REQ7	TO6: Analyze system reliability	Reliability robustness
REQ9, REQ10	TO7: Calculate test coverage based on components and interfaces during test execution	Coverage
REQ16	TO8: Test for security vulnerabilities	Security

Table 3 Formulated test aspects

Test aspect	Description	Test objective(s)
Memory aspect	Supervises system memory consumption by tracking all memory allocations and deallocations	TO1
Performance Profiler	Captures function executions and creates an execution profile over the SUT during the testing process	TO2, TO3, TO4
Robustness aspect	Generates service request jams to test whether SUT is able to recover from hang situations	TO5, TO6
Reliability aspect	Collects information on SUT states (resets, panic reasons, etc.), and failed cases and causes of failures (if known)	TO6
Coverage aspect	Monitors SUT execution by gathering data from all other subsystems, components, and interfaces accessed during the test execution	TO7
Security aspect	Performs security-related testing on the SUT and related components, subsystems, and system services	TO8

the need for understanding the final implementation because the pointcuts defining the possible hang points can be formulated.

A common characteristic of all these test aspects is the scattered nature of all the concerns they represent. While the hang situations may manifest themselves in a wide variety of system components, the monitoring, profiling, reliability, and coverage measurement activities are all system-wide concerns. Considering the performance profiler aspect as an example, it collects information on each function call and the execution time of related services and records the call time and the original client information, the caller and the target. Hence, it creates a system profile that can be used in evaluating the system design, for example, optimization needs, bottlenecks, and critical points. As call and execution joinpoints can be used to easily formulate the relevant criteria for the profiler aspect to surface, the design of such an aspect is rather trivial (although the result may not be efficient).

Coverage analysis typically requires the tracking of the system execution by means of instrumented marking mechanisms and collecting the resulting data for further processing. Commercial tools for dynamic test coverage analysis exist and provide the means for measuring the test coverage on the SUT. The coverage aspect we have developed is woven into the SUT similar to how traditional non-aspectual existing coverage analysis tools instrument the SUT for its analysis. However, instead of measuring the SUT, the coverage aspect collects data on the execution of other subsystem code and on the calls to other components. Compared with commercial coverage measurement tools, we found that aspects benefit from being integral to the SUT and allowing the use of custom coverage criteria.

At the code level, we used AspectC++ (AspectC++ 2012; Spinczyk et al. 2002) to write the aspect code. The advice code implementing the test objectives is composed of three parts: pre-test activities, test execution, and post-test activities. Pre-test activities set the environment as necessary for the test execution, including system state, parameters, and similar. The test execution part implements the actual test according to the test plan. Finally, post-test activities set the system state as specified after test execution. Created aspects were relatively small in size and simple in terms of functionality. For example, the robustness aspect of a major subsystem (around 20–25 % of the complete SUT in terms of lines of code) was implemented with only 174 lines of code and increased the final binary

size by 2kB (8 %). As expected, this increase is directly proportional to the SUT function calls to device drivers (in this case 6 calls). It should be noted that the control logic is not included in these numbers (we only count the aspect code that is weaved in the injection points). The code snippet of Fig. 3 gives an example of a part of a simple robustness aspect used to create stuck situations by manipulating function arguments and return values.

A description of the example aspect implementation is as follows.

Line 3 First we define a pointcut for the test objective that specifies the rule for the aspect to take effect. In this case for simulating a stuck situation, we define a call joinpoint to take effect on each function call to class `RDriver` functions.

Lines 5–8 Pointcut definition is followed by a class function for creating the aspect instance. This is only needed if one wants to override the default aspect of function.

Line 10 Test cases are defined by an advice that specifies the actions on joinpoint execution. In this case, the around advice specifies that our test code is to be executed around the original function.

Lines 12–16 The pre-test activities are used to set the system state corresponding to the test plan. In this case, we decided to skip all calls to method `Open` in the driver, since we want to allow this method to be called.

Lines 17–23 The actual test case implementation for a jamming situation replaces the original function call with an infinite loop. If test control is set to 1, the call will get stuck and start waiting forever in line 20. Hence, the code in line 21 should never be executed.

Lines 24–25 The post-test activities would normalize the system state after test execution, but are here left empty since no actions are needed in this case.

Line 28 Inside one test aspect, we have implemented control logic for more than one test objective since the robustness test aspect covers more than one objective. Specifically, in this case, another test case examines the system behavior with various parameter values and their impact on the SUT.

Line 30 Another advice implements test cases for exercising the set of functions with certain parameter values.

Lines 31–37 The pre-test activities initialize the test harness for test execution.

Lines 38–48 The actual test code changes the parameter value corresponding to the test plan and proceeds with the original function call. The original function is available for us and since we have an around advice, we execute it here. The resulting return value is passed to the test harness for evaluation and the test is executed a number of times according to the test plan.

Lines 49–50 After the original function has returned, the aspect code is executed and we have the return values available for manipulation. The SUT receives the return value only after this advice code.

This example robustness aspect is not reusable and it is also very much implementation dependent. To address these issues, we used aspect inheritance and the ability to combine pointcut expressions. The pointcut expression `testObjective6` (line 28) of this sample robustness aspect was extended to include all synchronous and asynchronous service requests outside the SUT. The test case designer had two possibilities: to investigate the SUT design to find out service request joinpoints, or allow the aspect weaver to inject the test code and examine the resulting project repository file for information about joinpoints. The former approach can be used in the early development phases, and the latter approach can be used when specializing the tests for the SUT. For example, a generic pointcut matching asynchronous service requests can be formulated to capture all such segments in the SUT code for robustness testing thus checking that the SUT is able to recover from

```
 1: aspect RobustnessAspect public testBaseAspect {
 2:
 3:   pointcut testObjective5() = call("% RDriver::%(...)");
 4:
 5:   static RobustnessAspect * aspectof () {
 6:     RobustnessAspect* z = ::new RobustnessAspect();
 7:     return z;
 8:   }
 9:
10:  advice testObjective5() : around() {
11:     // [Pre-test activities]
12:     TestCases.Logger(this);
13:     bool skip = false;
14:     char str[] = "int RDriver::Open(int) ";
15:     if (strcmp(str,sig)) skip = true;
16:
17:     // [Test execution]
18:     if ( (_testcontrol==KTestStuck)&&(!skip) ) {
19:       RDebug::Print(_L("[TEST] Stuck begins... %s"), sig);
20:       while(1) { } // foreverloop
21:       RDebug::Print(_L("[TEST] STUCK ENDS"));
22:     } else if(skip) { tjp->proceed(); }
23:
24:     // [Post-test activities]
25:
26:   };
27:
28:  pointcut testObjective6() = testMethods();
29:
30:  advice testObjective6() : around() {
31:     // [Pre-test activities]
32:     TestCases.Logger(this);
33:     bool skip = false;
34:     char str[] = "int RDriver::Open(int) ";
35:     if (strcmp(str,sig)) skip = true;
36:     TestCases.Initialize(this);
37:
38:     // [Test execution]
39:     if ( (_testcontrol==KTestParameters)&&(!skip) ) {
40:       for (int idx = 0; idx<TestCases.Count(this); idx++) {
41:         TestCases.Logger(idx, this);
42:         int* arg1 = reinterpret_cast<int*>(tjp->arg(0));
43:         *arg1 = TestCases.GetValue(idx,this);
44:         tjp->proceed(); // Run original code
45:         TestCases.ReturnValue(idx, this, (int*)tjp->result());
46:       }
47:     }
48:
49:     // [Post-test activities]
50:     *((int*)tjp->result()) = TestCases.SetReturnValue(this);
51:     };
52:   };
```

Fig. 3 Example of robustness aspect implementation

situations when such services get stuck. This can be done without reading the SUT code or its design documents.

Hence, to better support reuse in the aspectual test code, a simple and generic base aspect is formulated for each non-functional concern to act as a foundation for a number of specialized test aspects. Furthermore, base aspects' advices use generic pointcut expressions that are refined when specialized in order to fit the final implementation. The code snippets of Fig. 4 illustrate an aspect for a security concern, which checks before the execution of any of the guarded methods whether the method is called from a legal context and with legal parameters. The test aspect inherits the abstract base test aspect `testBaseAspect`, which is responsible for setting up the test environment.

The abstract base aspect is realized by defining a specialized test aspect. This specialized test aspect is tailored to the SUT using proper pointcut expressions, as illustrated in the code snippets from the relevant aspect header files shown in Fig. 5.

```
-------- File testBaseAspect.ah:
...
10: aspect testBaseAspect {
11:   pointcut virtual initialization() = 0;
12:   advice execution(initialization()) : before() {
13:     // here code for initializing the
14:     // testing environment (stubs, mocks, etc.)
15:     ...
22:   };
23: };
...
-------- File testSecurityAspect.ah:
...
 2: #include "testBaseAspect.ah"     // Base aspect
 3: #include "methodsignatures.ah"   // SUT specific pointcuts
 4: #include "contextlimitation.ah"  // SUT context details
    ...
10: aspect testSecurityAspect : public testBaseAspect {
11:  // realize base aspect
12:   pointcut virtual initialization() =
13:       execution("int E32Main::(....)");
14:
15:   advice execution(methodsignatures()) &&
16:     contextlimitations() : before() {
17:       // [Pre-test activities]
18:
19:       // [Test execution]
20:       // here generic test execution code for the security concern.
    ...
30:      // [Post-test activities]
31:    };
32: };
...
```

Fig. 4 Snippets from an aspect for a security concern and its super class

```
-------- File methodsignatures.ah:
...
17: pointcut methodsignatures() = "% ...::HandleMessageL(...)"
      || "% ...::ServiceL(...)";
...
-------- File contextlimitation.ah:
...
12: pointcut contextlimitations() = within("CMyClass")&&
      !within("RDriver");
...
```

Fig. 5 Pointcuts from the aspect header files

3.3 Summary of case study I

Due to the imprecise nature of the most real-life software requirements, generally, test aspects cannot be directly derived from them. Instead, more detailed goals need to be targeted. Furthermore, especially non-functional requirements are often implicit and intimately associated with one or more functional requirements, implying that some refinement is needed before the actual implementation of the corresponding test aspects. We therefore refined the requirements based on certain kinds of use case descriptions as well as system characteristics that could help in gaining an improved insight into the non-functional requirements. Based on these and the original requirements, non-functional concerns had to be identified, which in turn could be used to derive *test objectives*. Test objectives bind individual non-functional concerns to testing goals associated with use case descriptions and system characteristics. In other words, a test objective defines qualities that should be observable in order to fulfill the designated testing goals. The next step was to write down *test aspects* that implemented the test objectives using a suitable aspect-oriented language. Finally, a *test case* defined how the test procedure was to be carried out, introducing the technical test steps, etc. With the above definitions, we essentially used the test objectives as informal specifications for test aspects. We wrote the aspect definitions, guided by the semantics of the aspect language and based on the descriptions of the objectives.

Another important finding is the following process we suggest for creating aspects from requirements:

1. Identify non-functional concerns by employing use-case descriptions and system characteristics to help in breaking down the requirements (cf. Table 1).
2. Based on the findings, define the testing concerns and formulate the test objectives (cf. Table 2).
3. Consider how the test objectives could be achieved at the design level and derive specifications for the test aspects. This includes defining related test cases supporting the objective by rigorously elaborating the behavior and mining for incorrect behaviors to test (cf. Table 3).
4. Fine-tune the aspect definitions in order to be able to formulate the test aspect implementations (cf. Fig. 3).

By giving descriptive names for the test aspects, we can define the connections between the system characteristics, requirements, test objectives, and test aspects. It should be

noticed, however, that the test aspects represent more general *testing concerns*, and thus their names do not always match the corresponding test objectives one-to-one. On the other hand, for a more complex test objective, there may be several test aspects implementing it—unless the objective is split into several more manageable objectives.

4 Case study II

The second case study was performed on the same SUT as the first one but concentrated on a model-based solution for automatically generating test aspects. The motivation was to lessen the need for expertise in writing the test aspect implementations. We used scenarios for specifying tests at a high level of abstraction. For the visual specification of testing scenarios, we used a UML2-compliant variant of *live sequence charts* (LSCs) (Damm and Harel 2001; Harel and Maoz 2008), a formalism that extends the classical sequence diagram's partial-order semantics mainly by adding universal and existential hot/cold modalities, allowing a visual and intuitive specification of scenario-based liveness and safety properties. LSCs extend the classical partial-order semantics of sequence diagrams mainly by adding universal and existential hot/cold modalities. It thus supports the visual specification of scenario-based liveness and safety properties, allowing specifying scenarios that may happen, must happen, or should never happen. Vertical lines called lifelines represent specific system objects. Time goes from top to bottom. Must (hot) events and conditions are colored in red and use solid horizontal lines; may (cold) events and conditions are colored in blue and use dashed horizontal lines. A specification typically consists of many charts, which may be interdependent.

An important concept in LSC semantics is the *cut*, which is a mapping from each lifeline to one of its locations—denoting event occurrences along the lifeline. The cut represents the state of an active scenario during execution and induces a set of enabled events—those immediately after it in the partial order defined by the diagram. A cut is hot if any of its enabled events is hot (and is cold otherwise). When a chart's minimal event occurs—that is, an event that is minimal in the partial order defined by the chart occurs—a new instance of the chart is activated.

The semantics of LSC depends on the temperatures of events and cuts. Roughly, a hot enabled event must eventually occur, while a cold enabled event may or may not occur eventually. Thus, an occurrence of an enabled event or a `true` evaluation of an enabled condition just causes the cut to progress. However, the result of an occurrence of a non-enabled event from the chart or a `false` evaluation of an enabled condition depends on the current cut's temperature: if it happens when the cut is cold, the scenario exits gracefully with a *completion*; if the occurrence of the non-enabled event happens when the cut is hot, this is considered a *violation*. Finally, note that a chart does not restrict events not explicitly mentioned in it to occur or not to occur during a run (including in-between events mentioned in the chart); a chart only restricts the order of events explicitly appearing in it.

We use the UML2-compliant variant of LSCs, which was defined using the *modal* profile in (Harel and Maoz 2008). Moreover, the LSC variant used in our work follows the play-out execution mechanism of (Harel and Marelly 2003) and includes an *execution mode* for each method. The execution mode can be either *execute* or *monitor*. When a method designated with *execute* is enabled in one chart and is not violating in any other chart, the play-out mechanism executes it. Finally, the LSC variant used in our work supports a polymorphic interpretation of lifelines: lifelines represent classes, not objects,

and so each lifeline may be applied to all objects directly or indirectly inheriting from the class it represents.

To execute the tests, we automatically translated the diagrams into test *scenario aspects* using a modified version of the *S2A compiler* (Maoz et al. 2011). After weaving with the SUT code, the generated aspects followed the execution of the tests that were specified in the diagrams and reported on their run-time progress and results using *scenario-based traces* (Maoz 2009a). The traces were visualized and explored in a prototype tool called the *Tracer* (Maoz and Harel 2011). An overview of the tool chain used in the second case study is illustrated in Fig. 6. Next, we elaborate on each of the components in the tool chain and on their use in the second case study.

4.1 Defining the scenarios

A specification was made of a number of LSCs, divided between several use cases. The profile extension allows setting the temperature mode and the execution mode of each method and condition, as required by LSCs: the temperature mode value can be "hot" or "cold", the execution mode value can be "execute" or "monitor".

Figure 7 shows an example of a simple test case, taken from our case study. This scenario specifies that whenever any instance of class `class_D1` calls method `method_E1m1` of an instance of class `class_E1`, the test code should execute a loop. The loop is executed five times, calling method `method_G1m2` of `class_G1`, where the caller is the instance of `class_E1` bound earlier to the middle lifeline of the LSC.

Note that the modeled scenarios combine monitoring with execution. That is, not only they listen for relevant events to monitor the progress of the tests, but also some methods are designated with the *execution mode*. When such a method is enabled in one chart and not violating in any other chart, the generated code, described next, executes this method using generated inter-type declarations. Unfortunately, the distinction between monitoring and execution is not visible in the diagram because of a limitation of sequence diagrams editor of RSA (IBM Rational 2012).

4.2 Generating and executing testing scenarios

S2A is a compiler that translates LSCs into AspectJ code. S2A supports scenario-based execution following the play-out operational semantics of LSC (Harel and Marelly 2003). It provides full code generation of reactive behavior from visual inter-object scenario-based specifications. Aspect code generation in S2A follows the compilation scheme first

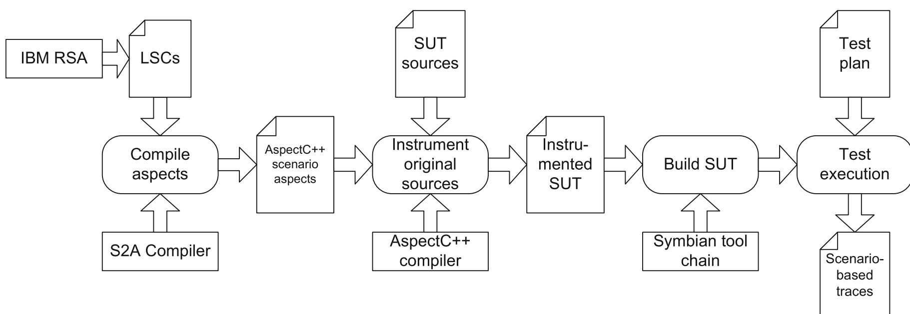

Fig. 6 Overview of the tool chain used in the second case study

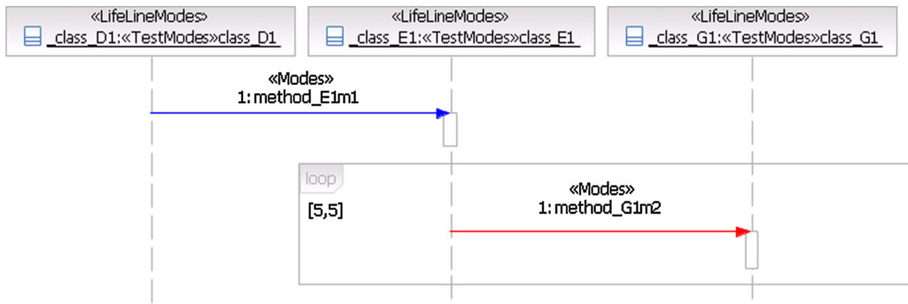

Fig. 7 An example of a simple testing scenario: after method `method_E1m1` of an instance of class `class_E1` is called by an instance of `class_D1`, the testware executes the method `method_G1m2` from `class_D1` to `class_G1` five times (in a loop)

presented in Maoz and Harel (2006). A detailed description of S2A and the compilation scheme appears in Maoz et al. (2011).

Roughly, each sequence diagram is translated into a *scenario aspect*, which simulates an automaton whose states correspond to the scenario cuts; enabled events are represented by transitions that are triggered by pointcuts, and corresponding advice is responsible for advancing the automaton to the next cut state. The compiler comes with a runtime component (not generated), which includes code that is common to all scenario aspects such as utility methods and super classes.

To use S2A with the SUT (the same system as in the first case study), we have designed and implemented a version of S2A that outputs AspectC++ code rather than AspectJ code. Snippets from the generated AspectC++ test scenario aspect code appear in Figs. 8 (continued in Fig. 9), and 10), divided between .cc and .ah files, respectively. In the .cc file, note the `changeActiveMSDCutState` method, which is responsible for the automaton logic, moving the scenario from one cut state to another. The generated pointcuts and advice are declared in the corresponding .ah aspect header file. Note the pointcuts, listening out for method calls referred to in the model, and advice, calling the `changeCutState method` of the automaton logic.

4.3 Tracing and trace visualization and exploration

S2A's runtime component supports the generation of textual *model-based traces* (Maoz 2009a; Maoz 2009b). The traces can be generated from programs that are instrumented with S2A's generated aspects. In our context, the generated model-based traces provide information about the executed tests progress and completion states. These are viewed and explored using *Tracer* (Maoz et al. 2011), a prototype tool for the visualization and interactive exploration and analysis of scenario-based traces. The input to the Tracer is a scenario-based model of a system given as a set of UML2-compliant LSCs, and a scenario-based trace, generated from an execution of the system under investigation.

A screenshot from the Tracer's main view, taken from the second case study, is shown in Fig. 11. Roughly, the Tracer's main view is based on an extended hierarchical Gantt chart, where time goes from left to right and a two-level hierarchy is defined by the containment relation of use cases and sequence diagrams in the model. Each leaf in the hierarchy represents a sequence diagram, the horizontal rows represent specific active instances of a diagram, and the blue and red bars show the duration of being in specific

```
 1: #include "MSDAspectTest_case8.ah"
 2:
 3: //Constants for instances, locations and variables
 4: int MSDAspectTest_case8::class_H2_INST__class_H2 = 0;
 5: int MSDAspectTest_case8::class_I2_INST__class_I2 = 1;
    ...
 9: //Aspect constructor
10: MSDAspectTest_case8::MSDAspectTest_case8() {
11:   addMinimalEvent(MSDMethods::class_H2_class_I2_method_J2m1);
12:   addMinimalEvent(MSDMethods::class_J2_class_K2_method_K2m1);
13:   setHotCut(1,1,1,2,1); setHotCut(2,2,2,4,2);
    ...
18:   numberOfLifeLines = 5;
19:   numberOfInstances = 5;
20:   numberOfVariables = 0;
21:   setInstanceOf("MSDAspectTest_case8");
22: }
23:
24: MSDAspectTest_case8::~MSDAspectTest_case8() {}
```

Fig. 8 A (pretty-printed) snippet from a generated test scenario aspect code .cc file (continued in Fig. 9)

cold and hot relevant cuts. The horizontal axis of the view allows following the progress of specific scenario instances over time, identifying events that caused progress, and locating completions and violations. The vertical axis allows a clear view of the synchronic characteristic of the trace, showing exactly what goes on, at any given point in time, at the abstraction level of the scenarios used in the model. Additional screenshots from the second case study can be found on the Tracer's website (Maoz 2012).

4.4 Test setup

Figure 12 illustrates a simplified test setup used in the second case study. The basic test setup is composed of *test cases*, *test harness*, and *test results*. The test cases specify the tests to execute. Executing the test cases utilizes a test harness to control the SUT accordingly. The test harness exercises the SUT based on the test cases and reports the SUT outputs for further evaluation. In this setting, the generated test aspects implement most of the test harness and the test cases that exercise the SUT, so there was no need to create additional test cases or modify the test harness for this purpose.

We created test scenarios based on three goals: (1) generic monitoring, (2) monitoring for regression testing, and (3) simple new test cases. The generic monitoring scenarios check that the SUT actually behaves as designed, for example, that certain method calls are indeed followed by certain behavior. For regression testing, we created scenarios that had revealed certain problems in the older versions of the SUT but should be working with no problems in the version under test. Finally, we created new test scenarios that after a certain sequence of events exercise the target system with additional method calls. These scenarios were rather simple since we wanted to be able to notice other problems related to adopting the technique in the target platform. In total, we had 32 different test scenarios.

```
28: // MSD Logic:
29: void MSDAspectTest_case8::changeActiveMSDCutState
30:      (int MSDm, Object* sourceObject, Object* targetObject,
31:       ActiveMSDAspect* activeMSD, vector<void*>* args) {
32:   bool unification=false;
33:   switch (MSDm) {
34:   case MSDMethods::class_K2_class_L2_method_L2m2:
35:     if(activeMSD->instancesEquals
36:                       (class_K2_INST__class_K2,sourceObject)
37:     && activeMSD->instancesEquals
38:                       (class_L2_INST__class_L2,targetObject)) {
39:       unification=true;
40:       if(activeMSD->isInCut(2,2,1,2,1)) {
41:         activeMSD->setCut(2,2,1,3,2);
42:         return;
43:       }
44:     }
   ...
55:   if(!unification)//No unification...
56:     return;
57:   break;
58:
59:   case MSDMethods::class_J2_class_K2_method_K2m1:
60:     if(activeMSD->instancesEquals
61:                        (class_K2_INST__class_K2,NULL)
62:     && activeMSD->instancesEquals
63:                        (class_J2_INST__class_J2,NULL)) {
64:       unification=true;
65:       if(activeMSD->isInCut(2,2,0,0,0)) {
66:         activeMSD->setLineInstance
67:                        (class_K2_INST__class_K2,targetObject);
68:         activeMSD->setLineInstance
69:                        (class_J2_INST__class_J2,sourceObject);
70:         activeMSD->setCut(2,2,1,1,0);
71:         return;
72:       }
73:     }
   ...
82:   }
83:   if(activeMSD->checkViolation()) activeMSD->completion();
84: }
85:
86: bool MSDAspectTest_case8::evaluateCondition
87:              (int conditionNumber, ActiveMSDAspect* activeMSD){
88: ...
89:   switch (conditionNumber) {
90:   case 5:
91:     return variable_value_X == 5;
92:   }
93:   return false;
94: }
```

Fig. 9 Continued from Fig. 8, a (pretty-printed) snippet from a generated test scenario aspect .cc file. Cut states are represented using tuples of locations, one for each of the LSC's lifelines. Method `changeActiveMSDCutState` implements the automaton logic, moving from one cut state to another

```
 1: #ifndef MSDASPECTTEST_CASE8_AH
 2: #define MSDASPECTTEST_CASE8_AH
 3:
 4: // Include necessary runtime headers
 5: #include "MSDRuntimeClasses.h"
 6: #include "MSDAspect.ah"
 7:
 8: // Include needed project imports
 9: #include "inc"
10:
11: // Include needed system imports
12:
13: aspect MSDAspectTest_case8 : public MSDAspect {
14:
15:   //Pointcuts and advices:
16:   pointcut class_K2_class_L2_method_L2m2() =
17:     call("void class_L2::method_L2m2(...)");
18:   advice class_K2_class_L2_method_L2m2() : after() {
19:    changeCutState(MSDMethods::class_K2_class_L2_method_L2m2,
20:      tjp->that(),tjp->target(),NULL);
21:   };
22:   pointcut class_J2_class_K2_method_K2m1() =
23:     call("void class_K2::method_K2m1(...)");
24:   advice class_J2_class_K2_method_K2m1() : after() {
25:     changeCutState(MSDMethods::class_J2_class_K2_method_K2m1,
26:       tjp->that(),tjp->target(),NULL);
27:   };
28:   pointcut class_K2_class_L2_method_L2m1() =
29:     call("void class_L2::method_L2m1(...)");
30:   advice class_K2_class_L2_method_L2m1() : after() {
31:     changeCutState(MSDMethods::class_K2_class_L2_method_L2m1,
32:       tjp->that(),tjp->target(),NULL);
33:   };
34:   pointcut class_H2_class_I2_method_J2m1() =
35:     call("void class_I2::method_J2m1(...)");
36:   advice class_H2_class_I2_method_J2m1() : after() {
37:     changeCutState(MSDMethods::class_H2_class_I2_method_J2m1,
38:       tjp->that(),tjp->target(),NULL);
39:   };
...
```

Fig. 10 A snippet from a generated test scenario aspect code .ah file. Note the pointcuts, listening out for method calls referred to in the model, and advice, calling the `changeCutState` method of the automaton logic

We used existing test cases from the documentation of the system under test and created monitoring and execution scenarios based on them. In the monitoring case, a trace event is logged after the specified behavior realizes, whereas in the execution case, certain

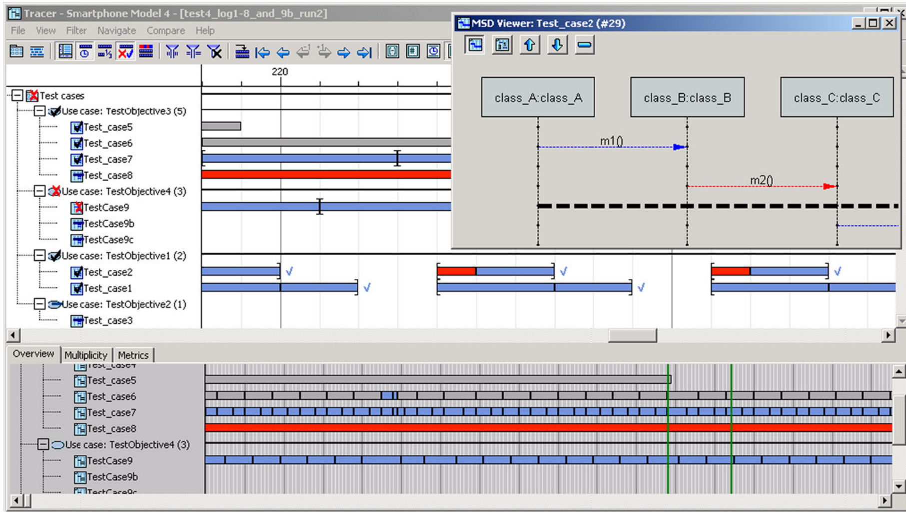

Fig. 11 A screenshot from the Tracer's main view, showing one of the test execution traces and (part of a) scenario instance, with its cut shown as a *black dashed line*. The scenario instance was opened after double-clicking one of the bars in the trace. Note the √ completion marks at the end of scenario instances' *bars*

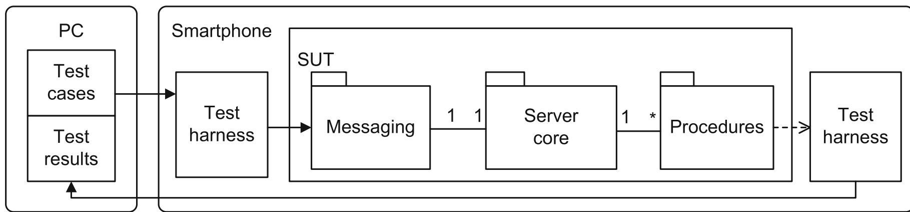

Fig. 12 Testing setup for exercising the testing LSCs in the second case study

functionality is executed after the specified behavior. This allowed us both to observe that the desired behavior actually happens and to advance and execute the related tests.

In the second case study, we wanted to utilize the existing test setup as much as possible, to avoid inventing new test cases or implementing extra functionality in the test harness. The available test cases described the testing objectives and we expected to be able to model these as testing scenarios using the sequence charts. Since the generated aspects do not generally execute the test cases, we needed explicit SUT control. Using the existing test setup to control test execution, we avoided writing extra control for test execution and the observation of the test results. The test setup is illustrated in Fig. 12, where test cases executed in PC request services from the SUT. The corresponding test aspects are executed by the automata according to the scenarios. Test results are collected from the SUT as traces and are received back by the test PC.

4.5 Generating test scenario aspects

To generate executable tests from the LSCs, we used the modified version of the S2A compiler described earlier. Rather than generating AspectJ code like the original version of

S2A, our modified version generates AspectC++ code, as required in order to apply to our target SUT. Some differences between the languages had to be addressed in our modified version of S2A:

- Instead of generating a single AspectJ file for each LSC (a .aj file), we had to separate the generated aspect code into two files: aspect header (.ah) and aspect source (.cc).
- The original implementation of S2A relies on AspectJ aspects' ability to have joinpoints or pointcuts matching other aspect code. This is not possible in AspectC++ and so we had to implement a workaround to bypass the use of this feature.
- The pointers need to be addressed when compiling models to AspectC++ because C++ heap allocated objects are accessed via pointers. This must be taken into consideration when drawing and compiling sequence diagrams: if there is only class name, it refers to the whole class. If it is a pointer, then it refers to an object. Instances are accessed via pointers, which should be taken into consideration when drawing the model. Furthermore, the private and protected visibilities of class methods and data must be followed.

In addition to the above, we made changes to the code because of the non-standard Symbian C++ used in the SUT. Thus, some of the features of the original S2A compiler are missing in the modified version, for example, the method parameters are ignored.

All test scenarios were modeled with LSCs using IBM RSA, and then compiled using the modified S2A compiler. After weaving, the instrumented SUT executed the tests. We repeated the experiments a number of times, each time with some additional or modified tests. The resulting test execution traces were copied back from the smartphone onto the PC where they can be viewed using the Tracer (Fig. 11 above).

5 Evalution

In this section, we first present the evaluation of the two case studies. Then, we discuss the implications and directions for future research on the topic.

5.1 Evaluation of case study I results

The approach used in the first case study was based on two fundamentals: identifying non-functional concerns from requirements and binding the related testing objectives into code to be formulated as aspects. The following evaluation first discusses the requirements mining and related activities, and then the resulting test aspects.

In the first case study, the target was to formulate new hand-coded test aspects to gain better test coverage related especially to the non-functional requirements. As a starting point, we had existing requirements for the system with brief use-case descriptions. We chose to analyze only the basic and initial requirements set for the whole product line, which contained close to 150 requirements from which we identified 16 basic requirements as the starting point. We excluded all product-specific requirements from the analysis to focus on testing common system-wide issues.

The corner-stone was successful identification of the non-functional testing concerns that can be formulated as test aspects. It is not only the requirements analysis that is difficult, but also defining the test aspects in general. Although the connection between the code segments to capture and the aspect to inject is often obvious, it is not straightforward to identify the code elements that are meaningful to the non-functional concerns. Table 4

summarizes the changes between the original testing approach and the aspect-oriented approach used in the first case study.

To generalize the test code, abstract base level aspects and aspect inheritance can be used to formulate generic aspects and reusable advice code for testing non-functional concerns. In the first case study, the requirements were mostly functional but they obviously implied implicit non-functional requirements. However, we propose defining generic non-functional requirements for the system and fine-tuning the related test objectives based on the SUT characteristics. A security concern, for instance, is used to set initial security requirements for the system—a security policy and related restrictions—and the related test objectives aim at verifying that these security requirements are met. The test objectives and resulting test code can be relatively generic, since the actual test aspects specialize test code based on the SUT implementation. Hence, the same security concern can be addressed in testing other SUT implementations as well, thus promoting reuse.

A generic problem related to aspect-oriented techniques is clearly related to code instrumentation. In order to have joinpoints harnessed for testing, the aspects have to be woven into the system, which often requires instrumenting the final code. This complicates the testing process by introducing extra steps to build processes and requires specific test versions to be built. However, managing the test aspects is relatively easy when compared with the corresponding conventional implementation-level techniques. The easy weaving and non-invasive nature of test aspects makes them an attractive solution for non-functional testing; based on the first case study and previous experiences (Pesonen 2006), the non-invasive nature of aspect-orientation is a clear benefit in software testing compared with the conventional techniques.

Another problem related to the industrial adoption of aspect-oriented testing is tool support. The lack of proper, complete testing tools or interfaces for existing tools to be extended with aspect-oriented extensions complicates the adoption of the proposed testing methods in existing processes and tool chains. Such a variety of different testing tools

Table 4 Comparing changes when moving to aspect-oriented testing

Activity	Original approach	Aspect-oriented approach
Functional testing	Using test cases and production version of SUT	Using test cases and test aspects instrumented in SUT
Non-functional tests	No non-functional testing	Additional test cases for non-functional tests. Increased the number of test cases
Profiling	External tool using trace data. Required instrumenting the whole SUT. Significant performance overhead	External tool using trace data produced by profiling aspect. Requires selecting the code segments to profile and weaving the profiling aspect to SUT. Small performance overhead when using for targeted profiling
Test coverage	Specific test coverage tool used for measuring test coverage. Requires instrumenting the SUT prior to testing	Test coverage aspect used to measure test coverage. Requires weaving the coverage aspect to the SUT. Provides customized coverage data
Test build	Re-compilation with testing option required	Tests instrumented using aspect weaver, no changes to the SUT
SUT source size	Test code embedded in SUT codes, bigger	Test code separated in aspect files, smaller

creates problems in the project, error, and test management and unnecessarily complicates them. However, from the software management perspective, the aspect code can be managed using the same development tools and environments as the original code and does not add overhead in this respect. Aspects add a more powerful means to develop test control over SUT, as the semantics provide additional tools for formulating the test case and implementing it.

5.2 Evaluation of case study II results

A clear strength of the model-based approach taken in the second case study is that test case definition is done at a rather high-level of abstraction; while it takes advantage of aspects' ability to access the SUT internals, it does so without requiring the engineer to know the syntax of the aspects language and its complex semantics. This seems a potentially positive adaption factor, since a good command of aspect-oriented programming is not common in testing organizations (and in the industry in general). Moreover, the fact that the aspect code is automatically generated from the models guarantees certain quality in the code that executes the tests. Another positive adaption factor is the end-to-end visual nature of our model-based testing tool chain. Visualization is known as a way to address complexity and to make tasks more accessible to engineers. Again, test developers need neither write nor even understand the generated (aspect) code.

Based on the second case study, it is easy to create monitoring scenarios using LSCs. In case the class names and methods are known, it is easy to draw a scenario describing a sequence of method calls that should happen. However, it is mandatory to have a proper model of the SUT available, for example, a class diagram, and to understand the model elements and their relationships to the scenario. Since test designers often tend to be unaware of the system internals, the true potential of the aspects may remain unused. Good knowledge of the SUT model, in terms of the classes and their relationships, is thus a necessary requirement for test developers. If such model is partly available, some tests could be developed simply by copying sequence diagrams from the model's documentation and extend them with hot/cold modes. If this is not available or is not up-to-date, it is difficult to draw useful LSCs.

Good knowledge of the modeling language itself is another necessary requirement for test developers. While sequence diagrams in general and LSCs in particular are quite intuitive to draw and to understand when combined with additional features such as symbolic instances, and when put against a real system with a complex structure, intuition alone does not suffice. When the test developer knows the SUT well but is not an expert in the modeling language, as in our case, some tests simply do not happen or result in unexpected behavior, as the generated aspects code does not match the developer's intention. One way to address this is to divide the work between a modeler and a test engineer; the modeler would develop *scenario templates*, while the test engineer would instantiate these with classes and methods specific to the SUT.

We used LSCs' semantics of symbolic lifelines, with a polymorphic interpretation. Thus, lifelines are labeled with class names and any instance of the class may advance the related automaton. Although this allows defining powerful scenarios, it does not allow capturing issues related to a certain specific instance of the class (when exploring the produced traces, it is possible to identify the instances; we can consider this issue as partly resolved). In addition, not all original test cases could be modeled using scenarios; some required more complex support for data and control. A different, considerable disadvantage is the inability to create scenarios that explicitly cover behavior across separate threads or

processes. The current semantics and implementation generate aspect code that can only be thread-specific. This is a true limiting factor in many settings.

We did not have a single integrated development environment (IDE) that could be used throughout the tool chain. Modeling was done in IBM RSA; S2A is written in Java but its AspectC++ output is weaved to and compiled with the SUT; resulting execution traces are viewed with the Tracer, outside the SUT. The lack of a single IDE resulted in technical problems and process overhead; for example, if the generated code does not compile with the SUT, it is difficult to know where to look for the problem. Thus, a solution needs to be developed to combine the different pieces into an IDE.

Moreover, tool support for AspectC++, for example, a static analysis tool or a debugger running on the target system, was not available. As a result, it was too easy to create models that S2A fails to compile, or models whose generated aspects fail to compile on the SUT or, when executed, result in SUT crashes (requiring reboot of the smartphone). These interoperability issues, bugs, and limitations of tool support (e.g., for debugging), resulted in a slow, sometime frustrating process, far from what one would hope to achieve when introducing a model-based testing solution to an industrial setting.

5.3 Discussion

If we compare the status of aspect-oriented software development in coding and testing, it seems that the relatively low success in the former can be attributed to the dominance of object-oriented techniques; while aspects provide added-value, they also introduce complexities in understanding what happens at run time. However, in testing, the situation is somewhat different as object-oriented test design is not as dominating and test aspects are trivial to remove from the production code, which is not always possible with other techniques.

From the industrial point of view, tooling is extremely important since commercial organizations need tools that they can rely on. Both commercial and open-source testing tools are used widely; such tools are backed up by either commercial support or support by an open-source community. Our experiments in the case studies were conducted on a real industrial system with commercial value. Unfortunately, the results of the experiments and derived test aspects were not included in the product line assets nor taken into everyday use. First, the organization had a number of powerful tools that are dedicated to certain specific testing tasks, and our limited study did not produce attractive enough results to compete with them using our prototype tools that lack commercial as well as community support. Nevertheless, as we did only very small efforts marketing our approach inside the company, there might be future chances provided the tools become more mature. Second, before the first case study, the non-functional testing was performed in a very ad hoc manner, with no specific tool support. Hence, our experiments aimed at defining the foundations for including non-functional testing in a generic testing approach using aspects. The first case study did not include creating a tailored test tool, but we were able to identify issues related to implementing testing using aspects, some of which were addressed in the second case study. Nevertheless, we believe that the experiences from our case studies can help in targeting future research efforts to enhance the industrial usage of aspect-oriented technologies. Below, some of the lessons we learned are discussed in more detail.

Testing of some types of non-functional requirements can be facilitated by improved control and monitoring of the SUT. The former is needed, for example, for covering error cases that might be difficult to encounter during normal operations, like failing to open a

file or hardware access operations, and the latter is needed for collecting data on the internal behavior of the system, like response times for certain performance-critical operations. Preferably, such facilities should be non-invasive in their nature in order to keep the original implementation intact. For implementing such testware, we propose using aspect-oriented technologies, provided the tool support is adequate.

However, we do not generally propose using aspects for generic profiling purposes, since a number of commercially available profiling tools are in most cases better in terms of setup effort and cost. However, if an organization is implementing in-house profiling tools, aspects provide additional means for writing specific profiling and monitoring code, which could be otherwise difficult to implement. For example, a profiling aspect can be very effective when there is no need for a complete execution profile but instead only a certain behavior is of interest. In the first case study, for example, profiling the whole software of the smartphone would have produced huge amounts of profiling data, most of which may be not of interest, whereas using aspects we were able to use a customized, lightweight profiling tool for measuring only the SUT characteristics we were interested in.

From the manually formulated test aspects, also, the coverage and robustness aspects faced strong competition from existing commercial test coverage and unit testing tools. Although writing a coverage analysis aspect implementing a test coverage tool is easy, there are a number of commercial test coverage tools that are more effective and richer in features thus providing better results.

Aspects provide a useful technique for controlling performance overhead in testing, because of being modularized outside the SUT code. For example, when compared with the profiling tools that instrument SUT with code to produce profiling data, for example, via tracing data, we found aspects to produce a more realistic overview of the system performance: the overhead caused by executing the aspect code could be simply excluded from the execution time profile of the original code. However, the performance overhead of executing aspect code could make the timing analysis difficult in case the aspect designer is not able to identify all related pointcuts affected by the added delay. In our experiments, this was not a problem, since we were able to identify the relevant timers and time-outs and increase them correspondingly using the aspects, thus mitigating the profiling impact on timing.

The use of test aspects can also lead to increased overall test coverage. By simply analyzing the basic requirements, we were able to identify new test objectives that increased the overall test coverage by some percentage points in the first case study, although the software was already at the production phase. While the system we studied was not a perfect textbook example in the sense of good test cases and test coverage, we believe it represents a typical industrial system and typical software testing effort, which increases the value of the case studies. We assume that the ability to define test objectives and to formulate related test cases based on requirements tends to enable testing that captures more generic and crosscutting concerns. Furthermore, the ability to identify crosscutting issues already at the requirements analysis phase predicts a better understanding of the system-level design issues that, in addition to being difficult to verify at the testing phase, would present reliability risks in the final product. Moreover, based on the results with our object-oriented system already in production, we believe the approach is applicable also to legacy systems.

The benefits of the model-based testing approach we used in the second case study include access to SUT internals without knowledge in aspect programming, and the end-to-end visual characteristics, which makes it accessible to test developers. Challenges include test developers' need to have deep knowledge of the modeling language and its semantics,

some limitations in the expressive power of the modeling language used, and technical issues related to interoperability, immature implementations, and partial tool support when considered in an industrial setting.

Based on our two case studies, we believe that the greatest power of aspect technology lies in its ability to modularize testing concerns that would otherwise be scattered around the system and difficult to expose otherwise. Whether this should be done using manual coding or visual specification tools that are combined with code generation mechanisms depends on the availability of high-quality tooling and the skills available in the organization in using the languages and the associated tools. Moreover, generic aspects and template models have the potential of improving the applicability of using aspects for testing.

In order to fully utilize the benefits of test aspects, however, it seems that systems should be designed in a way that supports separating crosscutting concerns and specifying them using aspect-oriented methodology. While this contradicts the basic principle of non-invasiveness, we see that an optimal design process would recognize aspects from the beginning. In practice, defining the crosscutting structures as modularized items can be a cumbersome task, since identifying the crosscutting properties is not trivial and requires domain knowledge and a profound understanding of the application context and disciplines. In order to utilize the wide range of conventional testing techniques, traditions, and tools with test aspects, a possibility to manage the aspects properly as architectural elements is required. Considering the testing concerns as architectural elements in a system would allow testability characteristics to be taken into consideration already at the architecture design phase. However, methods for mapping the requirements to the system architecture, further to the design, and finally to the implementation are needed in order to understand the effects of these concerns in a systematic way. Furthermore, this would require the ability to formulate test objectives directly based on the early test aspects derived from requirements and to map them as components in the test harnesses.

6 Conclusions

The development of associated test code has become an important part of many software development projects. However, numerous approaches are problematic in two respects. Firstly, they address only properties that are tightly coupled and tangled with the functionalities of the system, and secondly, once introduced to the system, the separation between features and code associated with tests can be difficult.

In this paper, we have assessed the applicability of aspect-orientation in supporting the testing of embedded software. In the first case study, we aimed at identifying the system characteristics that lend themselves to be tested with aspects and to consider how to derive test objectives from the non-functional requirements. Common to all these requirements were the nature of expressing scattered concerns either to be tested or as system characteristics. With our approach, it was possible to create tests addressing quality properties, as well as to maintain the separation between the tested system and test software at the level of code files. In the second case study, we studied the use of model-based techniques to automatically generate the test aspects from visual specification models. While this requires new skills from the test designers who use the modeling tools, it seems a realistic goal when a good enough tool chain becomes available and is integrated with the IDEs.

Threats to the validity of our arguments are obvious: we have studied an industrial system that is proprietary and confidential, which makes it impossible for others to repeat our experiments. Despite these shortcomings, we think that experiences from studies such as ours

are needed in order the pave the way for the wider industrial deployment of aspect-oriented technologies in the embedded software domain. In a recent survey (Janicki et al. 2012), in the context of model-based smartphone application testing, it was concluded that pilot studies are the most important way to convince practitioners of the benefits of the proposed technology. Such pilot studies are often narrow and limited to specific domains and tools, but this is exactly what makes them relevant for the potential users the pilot is trying to address.

Acknowledgments The first and the third listed authors acknowledge partial funding from the Academy of Finland (grant number 121012). Part of the second listed author's work was done while he was with the Weizmann Institute of Science, Rehovot, Israel. In addition, the second listed author acknowledges partial funding from an Advanced Research Grant awarded to David Harel of the Weizmann Institute from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013).

References

AspectC++. (2012). AspectC++ website. At URL http://www.aspectc.org/. Cited March 2012.

AspectJ. (2012). AspectJ WWW site. At URL http://www.eclipse.org/aspectj/. Cited March 2012.

Clarke, S., Harrison, W., Ossher, H., & Tarr, P. (1999). Subject-oriented design: Towards improved alignment of requirements, design, and code. *ACM SIGPLAN Notices, 34*(10), 325–339.

Coelho, R., Kulesza, U., von Staa, A., & Lucena, C. (2006). Unit testing in multi-agent systems using mock agents and aspects. In *SELMAS '06: Proceedings of the 2006 international workshop on software engineering for large-scale multi-agent systems*, (pp. 83–90). ACM Press.

Craig, R. D., & Jaskiel, S. P. (2002). *Systematic software testing*. London: Artech House.

Damm, W., & Harel, D. (2001). LSCs: Breathing life into message sequence charts. *Journal on Formal Methods in System Design, 19*(1), 45–80.

Feng, Y., Liu, X., & Kerridge, J. (2007). A product line based aspect-oriented generative unit testing approach to building quality components. In *Computer Software and Applications Conference*, (Vol. 2, pp. 403–408). IEEE Computer Society.

Fewster, M., & Graham, D. (1999). *Software test automation: Effective use of test execution tools*. New York: Addison–Wesley.

Filman, R. E., Elrad, T., Clarke, S., & Akşit, M. (2004). *Aspect-oriented software development*. New York: Addison–Wesley.

Harel, D., & Maoz, S. (2008). Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling (SoSyM), 7*(2), 237–252.

Harel, D., & Marelly, R. (2003). *Come, let's play: Scenario-based programming using LSCs and the play-engine*. Berlin Springer.

IBM Rational. (2012). IBM Rational Software Architect homepage. Available at http://www-01.ibm.com/software/awdtools/swarchitect/websphere. Cited March 2012.

Janicki, M., Katara, M., & Pääkkönen, T. (2012). Obstacles and opportunities in deploying model-based GUI testing of mobile software: A survey. *Software Testing, Verification & Reliability, 22*(5), 313–341.

Kaner, C., Bach, J., & Pettichord, B. (2002). *Lessons learned in software testing: A context-driven approach*. New York: Wiley.

Kartal Y. B., & Schmidt E. G. (2007). An evaluation of aspect oriented programming for embedded real-time systems. In *Proceedings of the 22nd international symposium on computer and information sciences (ISCIS 2007)*, (pp. 1–6). IEEE, November 2007.

Kulesza, U., Sant'Anna, C., & Lucena, C. (2005). Refactoring the JUnit framework using aspect-oriented programming. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications*, (pp. 136–137). ACM Press.

Maoz, S. (2012). *Tracer website*. http://www.wisdom.weizmann.ac.il/maozs/tracer/. Cited March 2012.

Maoz, S. (2009a). *Model-based traces*. In M. R. V. Chaudron, (ed.), *Workshops and symposia at MODELS 2008, vol. 5421 of Lecture Notes in Computer Science*, (pp. 109–119). Berlin: Springer.

Maoz, S. (2009b). Using model-based traces as runtime models. *IEEE Computer, 42*(10), 28–36.

Maoz, S., & Harel, D. (2006). From multi-modal scenarios to code: Compiling LSCs into AspectJ. In Young M., Devanbu, P. T. (eds.), *Proceedings of the 14th international ACM/SIGSOFT symposium on foundations of software engineering (FSE'06)*, (pp. 219–230). ACM.

Maoz, S., & Harel, D. (2011). On tracing reactive systems. *Software and Systems Modeling (SoSyM), 10*(4), 447–468.
Maoz, S., Harel, D., & Kleinbort, A. (2011). A compiler for multi-modal scenarios: Transforming LSCs into AspectJ. *ACM Transactions on Software Engineering and Methodology (TOSEM), 20*(4), 18.
Maoz, S., Metsä, J., & Katara, M. (2009). Model-based testing using LSCs and S2A. In Schürr A., & Selic B. (eds.), *Proceedings of the 12th international conference on model driven engineering languages and systems (MoDELS'09), vol. 5795 of Lecture Notes in Computer Science*, (pp. 301–306) (short paper). Berlin: Springer.
Metsä, J., Katara, M., & Mikkonen, T. (2007). Testing non-functional requirements with aspects: An industrial case study. In *Proceedings of the 7th international conference on quality software (QSIC 2007)*, (pp. 5–14). Washington, DC: IEEE Computer Society.
Metsä, J., Katara, M., & Mikkonen, T. (2008). Comparing aspects with conventional techniques for increasing testability. In *Proceedings of the 1st international conference on software testing, verification, and validation (ICST 2008)*, (pp. 387–395). Washington, DC: IEEE Computer Society.
Nokia. (2012). *Symbian operating system* website. At URL http://symbian.nokia.com/. Cited March 2012.
Pesonen, J. (2006). Extending software integration testing using aspects in symbian OS. In McMinn, P. (ed.), *TAIC PART 2006*, (pp. 147–151). Washington, DC: IEEE Computer Society.
Pesonen, J., Katara, M., & Mikkonen, T. (2006). Production-testing of embedded systems with aspects. In Ur, S., Bin, E., & Wolfsthal, Y. (eds.), *Revised selected papers from the 1st international conference on hardware and software verification and testing, haifa verification conference (HVC'05), vol. 3875 of Lecture Notes in Computer Science*, (pp. 90–102). Berlin: Springer.
Pezzè, M., & Young, M. (2008). *Software testing and analysis: Process, principles, and techniques*. New York: Wiley.
Rajan, H., & Sullivan, K. (2005). Aspect language features for concern coverage profiling. In Mezini, M., & Tarr, P. L. (eds.), *AOSD '05: Proceedings of the 4th international conference on aspect-oriented software development*, (pp. 181–191). New York: ACM Press.
Rashid, A., Sawyer, P., Moreira, A., & Araujo, J. (2002). Early aspects: A model for aspect-oriented requirements engineering. In *RE '02: Proceedings of the 10th anniversary IEEE joint international conference on requirements engineering*, (pp. 199–202). Washington, DC: IEEE Computer Society.
Rook, P. (1986). Controlling software projects. *Software Engineering Journal, 1*(1), 7–16.
Spinczyk, O., Gal, A., & Schröder-Preikschat, W. (2002). AspectC++: An aspect-oriented extension to the C++ programming language. In J. Nobles & J. Potter, (eds.), *Proceedings of the 40th international conference on technology of object oriented languages and systems (TOOLS PACIFIC '02), vol. 10 of Conferences in Research and Practice in Information Technology*, (pp. 53–60). Sydney, NSW: Australian Computer Society.
Stamey, J., & Saunders, B. (2005). Unit testing and debugging with aspects. *Journal of Computing Sciences in Colleges, 20*(5), 47–55.
Xu, D., & He, X. (2007). Generation of test requirements from aspectual use cases. In *Proceedings of the 3rd workshop on testing aspect-oriented programs (WTAOP'07)*, (pp. 17–22). New York, NY: ACM.

Author Biographies

Jani Metsä is a program manager of wireless systems at Elektrobit Inc, USA. His research interests include aspect-oriented and non-functional testing, and mobile software. Metsä received a Doctor of Technology degree from Tampere University of Technology, Finland, in 2010. Contact him at jani.metsa@elektrobit.com.

Shahar Maoz is a faculty member at the School of Computer Science, Tel Aviv University, Israel. His research interests include the use of formal methods in software and systems modeling, static and dynamic analysis, and software visualization. Maoz received a PhD in Computer Science from the Weizmann Institute of Science, Israel, in 2009. Contact him at maoz@cs.tau.ac.il.

Mika Katara is currently a part-time associate professor at Tampere University of Technology, where he was in charge of software testing research and training before moving to Intel in April 2012. He earned his doctorate from the same institution in 2001. Contact him at mika.katara@tut.fi.

Tommi Mikkonen is a professor of distributed systems software at Tampere University of Technology. His current research interests include software architectures, distributed systems, aspect-oriented development methodologies, and web and mobile software. Mikkonen received a Doctor of Technology degree from Tampere University of Technology in 1999. Contact him at Tommi.Mikkonen@tut.fi.