Architecting a Testing Framework for Publish/Subscribe Applications*

Anton Michlmayr, Pascal Fenkam, Schahram Dustdar
Vienna University of Technology, Distributed Systems Group
Argentinierstrasse 8/184-1, A-1040 Vienna, Austria
{michlmayr,fenkam,dustdar}@infosys.tuwien.ac.at

Abstract

The publish/subscribe style is an emerging paradigm for the construction of loosely coupled systems. Yet, the verification of such systems remains difficult. We have constructed a framework called RAY *for testing publish/subscribe applications. This framework is implemented as an Eclipse plug-in. In this paper, we present the detailed architecture of* RAY*, as well as the interaction with its supporting components. In addition to the presented architecture for a testing framework for publish/subscribe applications, the contribution of this paper includes the experience gained during the development of this framework.*

1 Introduction

Verification and validation are key factors for the correctness of software systems. The most popular supporting technique is software testing, where the actual and expected behaviors of the application under test are compared based on a finite set of selected test cases. The concept of architecture-based software testing [20] proposes that the testing methodology leverages the architecture of the application under test. In this paper, we support this concept by presenting the architecture of a framework specifically designed for testing publish/subscribe applications.

In the publish/subscribe (pub/sub) paradigm [8], as illustrated in Figure 1, communication between components is achieved by sending and receiving *events* (also called *notifications*). Event receivers (called *subscribers*) express their interest in events announced by other components (called *publishers*) by means of *subscriptions*. The *pub/sub infrastructure* is responsible for persistency, management, and delivery of events to the interested subscribers.

The pub/sub paradigm possesses widely acknowledged benefits, which can be summarized as time decoupling (event publication and consumption may happen at different points in time), referential decoupling (publishers and subscribers have no reference to each other), and flow decoupling (publication and consumption are non-blocking). This decoupling makes the pub/sub architectural style interesting for many application domains. Yet, it significantly complicates the verification process. In fact, verification approaches for pub/sub applications are still not mature.

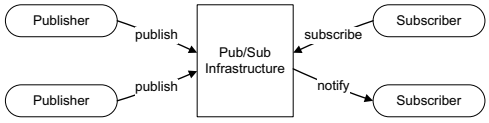

Figure 1. Pub/Sub Paradigm

The RAY framework aims at alleviating the task of testing pub/sub applications. The basic concept and the theoretical details of RAY are introduced in [19]. In the present paper, we focus on the architecture of RAY that is targeted at the Eclipse platform. Our framework leverages the pluggable architecture of Eclipse by using existing plug-ins where possible. In addition, we introduce plug-ins for specification transformation, and a mock pub/sub infrastructure. The contribution of this paper is the presented architecture of our pub/sub testing framework RAY, and the experience gained during the development of this framework.

The remainder of this paper is organized as follows. Section 2 introduces the Java Modeling Language, linear temporal logic, and the Eclipse platform. Section 3 gives a short overview of the RAY framework, while Section 4 describes its architecture in detail. Section 5 positions our work among related approaches, and Section 6 presents the experiences gained with RAY. Finally, Section 7 concludes the paper and points the way to future work.

2 Background

This section gives the background of our work: the Java Modeling Language, linear temporal logic, and Eclipse.

2.1 Java Modeling Language

The Java Modeling Language (JML) [3] is a formal behavioral interface specification language for Java programs.

*This work is supported by the Austrian Research Foundation (FWF) project RAY (Number P16970-No4).

Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)
0-7695-2655-1/06 $20.00 © 2006 IEEE

By adding annotations to the Java source code, developers are able to specify pre- and post-conditions of methods, as well as their normal behavior, exceptional behavior, class invariants, frame axioms, and assertion statements.

From a theoretical point of view, JML can be seen as a porting of the Design by Contract paradigm [18] to the Java programming language. In practice, JML is supported by a JML checker and a JML compiler. The first is responsible for parsing and type-checking JML-annotated source files, while the second generates Java bytecode. This bytecode includes assertions that are used for runtime verification purposes. Additional automated testing tools have emerged around JML, among which are JMLUnit [6], Tobias [15], Korat [2], and Jartege [21].

2.2 Linear Temporal Logic

Linear temporal logic (LTL) [11] is a well-established logic for reasoning about the behavior of computer systems. Typically, LTL formulas are used to express properties that program traces must satisfy. Such formulas are essentially constructed by extending the propositional logic with the temporal operators X (*"next"*), G (*"globally"*), F (*"eventually"*), U (*"until"*), and W (*"awaits"*) that apply to the states of the execution trace.

2.3 Eclipse Platform

The open source platform Eclipse [14] provides a foundation for building tools and applications. The architecture of this platform consists of a small kernel known as the *Platform Runtime*, and additional *plug-ins* which represent the smallest units that can be developed and delivered separately. The Platform Runtime provides the core mechanism for activating other plug-ins. For instance, the Java Development Tools (JDT) build a plug-in that adds Java development capabilities to the Eclipse platform.

Eclipse plug-ins are described in XML manifest files that contain the information required to activate the plug-ins at runtime. Plug-ins may be related in two ways: a dependent plug-in *depends* on the functionality of a prerequisite plug-in, while an extender plug-in *extends* the functionality of a host plug-in (see Figure 2).

The host plug-in may provide different *extension points*, while an *extension* defined by the extender plug-in uses these extension points, and causes the host plug-in to modify or enhance its behavior. In addition, *callback objects* may be used for the communication between host and extender plug-in. The host plug-in provides one or more *callback interfaces* which are implemented by callback objects in the extender plug-in.

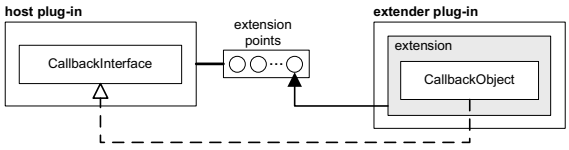

Figure 2. Eclipse Extension Model

3 RAY Overview

This section gives an overview of the RAY framework. Section 3.1 introduces the different components of our framework, while Section 3.2 describes how the pub/sub applications under test are specified.

3.1 Introduction

The testing activity is normally divided into test planning, test case generation, test execution, and test evaluation. The RAY framework aims at supporting the unit and integration testing of pub/sub systems from the planning phase up to the evaluation of test results. The current focus is on unit testing where a pub/sub application is tested without taking the pub/sub middleware into consideration.

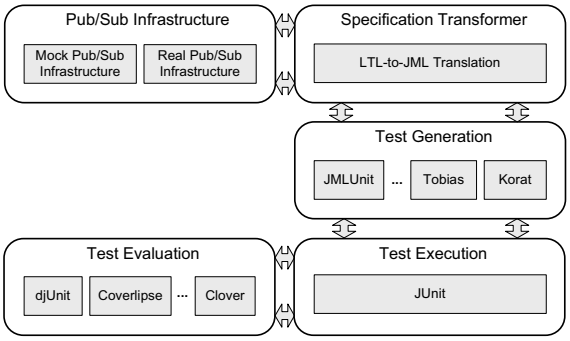

Figure 3. RAY Overview

Figure 3 illustrates the conceptual overview of the RAY framework. Basically, it is composed of a mock pub/sub infrastructure, a specification transformer, a test generator, a test instrumentation framework, and a test coverage tool.

The mock pub/sub infrastructure represents the core of the RAY framework. It is responsible for catching event publications and subscriptions, including the state in which they occur. This allows reasoning about the order and the states of event publications and subscriptions, respectively. The task of the specification transformer is to translate temporal logic formulas into JML specifications. The test generator is responsible for generating test drivers and test oracles from these JML specifications. The generated testing artifacts use the JUnit testing framework that provides facilities for executing the test suite, collecting the test results, and deciding about success or failure of the selected test cases. Finally, test coverage tools measure the success of the executed test suite by inspecting which parts of the

units under test are covered by the test cases. For the construction of RAY, we use existing open source frameworks where possible.

3.2 Pub/Sub Specification in RAY

The RAY framework follows the specification-based testing approach where programs are tested against formal specifications that define the expected behavior. This section outlines how these specifications are written for unit testing of pub/sub applications.

In unit testing, the focus of the testing process is at process and method levels. Traditionally, methods are specified using pre- and post-conditions that define the properties of the initial and final state of the method under test. These pre- and post-conditions typically do not support temporal properties. However, processes are better specified using temporal logic formulas which are not supported by JML.

The RAY framework constructs a temporal logic layer on top of traditional pre- and post-condition specifications that enables temporal reasoning. More precisely, we use LTL to specify the temporal behavior of methods. These LTL specifications consist of pre- and post-condition parts, and refer to the properties of the execution trace which is constructed by mock pub/sub infrastructure during the execution of the unit under test. In addition to state variables, these properties may refer to events published during method execution. For instance, the temporal formula

$$POST\ G[x > 0]\ \&\&\ F[e : type\ \text{EQ}\ \text{“}stockprice\text{”}];$$

represents the post-condition part of a method's LTL specification. It specifies that the state variable x is globally positive, and prescribes that finally an event of type *"stockprice"* is published. Section 6 gives another example of specifying pub/sub applications using LTL. More details on LTL can be found in [19].

4 RAY Architecture

This section describes the detailed architecture of the RAY framework that consists of a collection of Eclipse plug-ins. These plug-ins, and their interaction among each other, are presented below.

The RAY Eclipse plug-in provides a mock pub/sub infrastructure and facilities for specification transformation. Section 4.1 gives a detailed description of this plug-in which is specifically designed for testing pub/sub applications. Section 4.2 introduces the JMLEclipse plug-in that brings JML to the Eclipse platform. Section 4.3 describes the plug-in used for the generation of the testing artifacts, while Section 4.4 outlines the JUnit plug-in which is responsible for test execution. Finally, Section 4.5 addresses the test coverage plug-ins used in RAY.

4.1 RAY Eclipse plug-ins

The RAY framework is targeted at the Eclipse platform. In Eclipse, a *feature* represents a group of related plug-ins that get installed and updated together. This concept enables to split the functionality of larger plug-ins into smaller pieces. For instance, one plug-in may provide the user interface while another plug-in contains the application logic.

In our case, the RAY feature consists of two plug-ins. On the one hand, the mock pub/sub infrastructure provides the basic mechanism for unit testing pub/sub applications, while on the other hand, the specification transformer plug-in is responsible for checking and translating formal specifications of these applications.

Mock Pub/Sub Infrastructure: In general, unit testing is done in isolation in a simple context. To reach this goal, stubs and mock objects are commonly used to replace components that do not need to be tested yet. They should be significantly simpler than the object they replace, yet powerful enough to still allow the formulation of key properties of the replaced object.

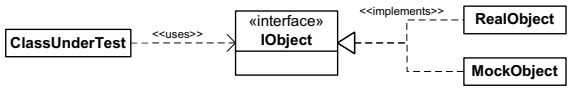

Figure 4. Mock Object Concept

This concept is illustrated in Figure 4. Typically, the real object and the mock object implement the same interface. The class under test uses this interface and therefore is not aware of the implementation mismatch between the real and the mock object. For testing purposes, the real object is replaced by the mock object.

When testing pub/sub applications, mocking the pub/sub infrastructure is a key factor for the quality and simplicity of the testing process. In fact, during unit testing it is not necessary to deliver published events to the interested subscribers. Therefore, we introduce a mock pub/sub infrastructure that captures all event publications and subscriptions, including the states in which they occur, but avoids the event delivery.

As in Figure 4, we define an interface *IPubSub* that declares the typical methods of a pub/sub infrastructure. This includes methods for event publication, subscription, and unsubscription. In addition to our mock pub/sub infrastructure, we provide an implementation of this interface for a real pub/sub infrastructure which uses the open source event notification service Siena [4]. Additional pub/sub middleware may be integrated in RAY by providing an implementations of this interface.

Figure 5 shows a UML component diagram of our mock pub/sub infrastructure. The mock pub/sub regularly captures information about the publishing object by using the Java reflection mechanism. The so extracted information

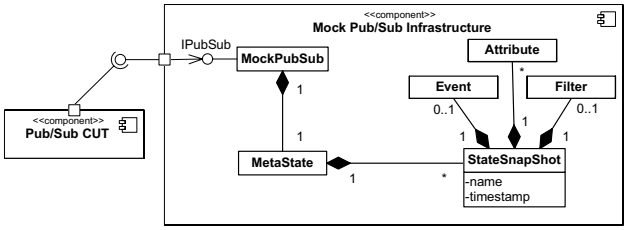

Figure 5. Mock Pub/Sub Infrastructure

is stored in a meta-state that represents a sequence of state snapshots. A state snapshot consists of the published event, the attributes of the publishing object including its values, the name of the publisher, and the timestamp of the event publication. In the same manner, the mock pub/sub captures all subscriptions and builds a state snapshot of the subscribing object, including the subscribed filter instead of the published event. The details of the mock pub/sub implementation are described in [19].

Letier et al. [16] propose two ways of building the states of the publishing objects: either regularly after a given delay δ or right after each event publication. Our current mock pub/sub supports the second technique. In addition, it provides the possibility to add state snapshots at user-defined breakpoints.

The integration of this plug-in neither adds user interface elements to Eclipse, nor does it extend other plug-ins. Instead it offers its pub/sub API to clients. In contrast to this, the plug-in depends on the Siena plug-in which is used for implementing the real pub/sub infrastructure.

Specification Transformation: The second RAY plug-in provides facilities for syntax checking and translation of formal specifications. It adds user interface elements to Eclipse for starting these tools, and shows log messages in the appropriate views.

The pub/sub applications are specified using LTL. These temporal properties refer to the information captured in the meta-state of our mock pub/sub infrastructure. We use the parser generator ANTLR (ANother Tool for Language Recognition) [22] to translate LTL formulas into JML.

This process is illustrated in Figure 6. After writing a grammar that defines the syntax of the input language

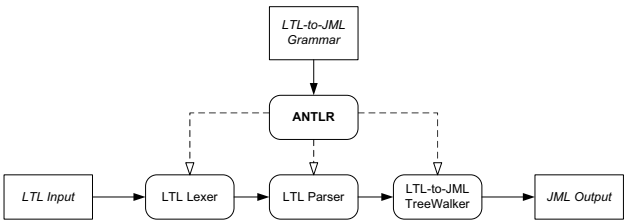

Figure 6. ANTLR Parser Generator

LTL, ANTLR automatically constructs lexer, parser, and tree walker. As usual in this context, the lexer is responsible for breaking up the input stream into tokens that are recognized by the parser. The task of the parser is to build an abstract syntax tree of these input tokens. Finally, this syntax tree is processed by the tree walker, that we use to generate the corresponding output in JML. Basically, the pre-condition part of the LTL specification is translated into JML *requires* clauses, while the post-condition part of the LTL specification is transformed into JML *ensures* clauses. Both of these clauses refer to the meta-state of the mock pub/sub infrastructure. In [19], we give the details of our LTL-to-JML translation including the translation rules.

The RAY framework uses the classes generated by ANTLR, to translate all LTL specifications which are annotated in a Java file. First, the input is processed by the *SpecChecker* that verifies the well-formedness of the LTL specifications using our LTL lexer and parser. If the syntax of the input is correct, the source file is forwarded to the *SpecErasure* whose task is to delete all JML specifications that were translated earlier. Finally, the modified source file is forwarded to the *SpecTranslator* that transforms the LTL specifications into JML by means of the LTL tree walker. If any errors occur during one of these steps, the modification of the source file is revoked, otherwise the changes are made durable. Eclipse's JDT therefore provides a *WorkingCopy* mechanism which represents a buffer whose changes are not performed immediately in the associated resource.

Figure 7 shows the components used for translating LTL into JML, and how they fit into the Eclipse platform. The RAY plug-in adds an object contribution for resources of type *org.eclipse.jdt.core.ICompilationUnit* which represent compilable units in Eclipse (i.e., Java source files), by using the extension point *org.eclipse.ui.popupMenus*. As a result thereof, the context menu of Java source files provides facilities for checking, translating and deleting specifications. In Eclipse, these actions are accessed via the callback interface *org.eclipse.ui.IObjectActionDelegate*. The Java source files are accessed by using the interface *org.eclipse.jdt.core.ICompilationUnit*.

Finally, problem markers (represented by the extension point *org.eclipse.core.resources.markers*) are used to highlight errors that occur during checking and translating of LTL specifications. These markers indicate the exact position and the reasons for the errors.

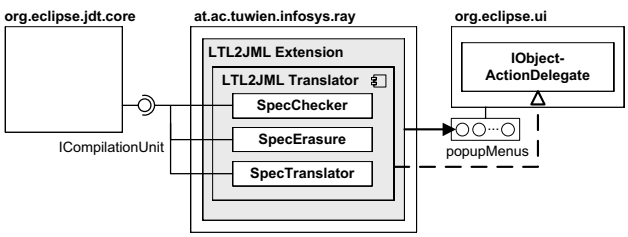

Figure 7. LTL-to-JML Translation

Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)
0-7695-2655-1/06 $20.00 © 2006 IEEE

4.2 JMLEclipse plug-in

The JMLEclipse plug-in integrates JML into the Java Development Tools (JDT) of the Eclipse platform. This plug-in, which is coordinated by the SAnToS laboratory at Kansas State University, consists of two components: the JMLEclipse core and the advanced user interface support. The latter mainly provides JML syntax highlighting, while the first is responsible for type checking and runtime assertion checking of JML statements. We use JMLEclipse for runtime assertion checking of the class under test.

The JMLUnit tool is part of the JML distribution. It is responsible for generating unit testing artifacts from JML specifications, as described in the following section. Yet, JMLUnit is not integrated into JMLEclipse. Therefore, the RAY plug-in provides a facility to invoke this tool directly from the Eclipse user interface. In the same manner as the extension shown in Figure 7, we use the extension point *org.eclipse.ui.popupMenus* again to add a context menu entry that launches the test generation process for the chosen Java file by using the interface *org.eclipse.debug.core.ILaunchManager*.

4.3 Test Generation plug-in

During test generation, test drivers (responsible for running the test suites) and test oracles (responsible for deciding about success or failure of an executed test case) are automatically generated from JML-annotated source files.

Currently, we use JMLUnit [6] as test generation tool. Figure 8 shows the generated testing artifacts in detail and illustrates how these artifacts are used for unit testing. JMLUnit takes a JML-annotated Java source file as input and generates two test classes. The test data for the class under test, say *TestClass*, can be found in a class called *TestClass_JML_TestData* that extends JUnit's class *TestCase*. The second class, called *TestClass_JML_Test*, extends the first class with the test oracle and the test driver. After invoking the constructor of the class under test, the test driver calls one method with a combination (e.g., the cross product) of the test data as argument. Furthermore, the tester may also add hand-written JUnit test cases to the test suite.

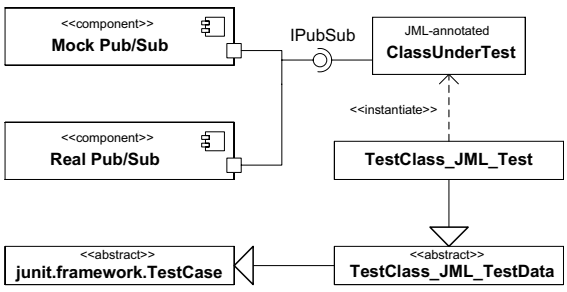

Figure 8. JMLUnit Test Generation

The actual test data is not generated automatically, but has to be filled by the tester manually. For each data type that appears as argument of the methods under test, a set of possible values is defined. This set is subsequently used by the test driver for combining the method arguments. For instance, if a method uses an argument of type *int*, the tester may define -1, 0, and 1 to be used for the tests. This mechanism is also leveraged to bring the mock pub/sub into the class under test by using the interface *IPubSub*. Instead of the real pub/sub infrastructure, the class under test gets a reference to the mock pub/sub in its constructor.

Automatic generation of test data represents an interesting extension to our framework. For instance, Cheon et al. [5] introduce an approach that generates test data using genetic algorithms and therefore aims at completely automating unit testing of Java programs using JML.

4.4 Test Instrumentation plug-in

The generated testing artifacts use the open source unit testing framework JUnit [17]. JUnit is well-established in Java unit testing and is therefore shipped as pre-installed plug-in with the Eclipse SDK. It provides a simple but powerful tool for writing and executing test suites that consist of repeatable and small unit tests.

The JUnit framework suggests to use assertions for validation purposes. These assertions, which have to be defined by the tester manually, specify the desired outcome of a specific test execution. The unit under test is finally verified by comparing the actual with the expected outcomes.

Used in combination with JML and JMLUnit, these assertions are extracted from the JML specification of the unit under test. More precisely, the pre- and post-conditions of the method under test are used by the test oracle to decide about the successful or unsuccessful test outcome. If the pre-condition of a method holds but the post-condition is violated, there is an error in the method implementation. If the pre-condition of a method does not hold, the test is meaningless, regardless of the method post-condition. If pre-conditions, post-conditions, or class invariants are not fulfilled during test execution, appropriate exceptions signal these errors and point to the violated statements.

4.5 Test Evaluation plug-in

The quality of the executed test suite is evaluated by measuring the test coverage. This task is accomplished by inspecting which parts of the unit under test are covered by the executed test suite. Therefore, different types of test coverage may be used, such as line coverage and branch coverage. In general, test coverage is an indicator for the quality of the chosen test cases, and shows which parts of the source code are not tested properly.

Our current prototype is based on the two open source test coverage tools djUnit and Coverlipse. We have also evaluated the commercial tool Clover. All these tools are

based on JUnit and provide plug-ins for the Eclipse platform. They can therefore be integrated in the RAY framework without further modification.

The test coverage tools take the generated unit tests as input, and present their output in the appropriate Eclipse views. Generally, this includes the percentage of coverage of the classes under test, as well as markers that highlight the lines or branches which are not covered by the test suite.

5 Related Work

Few attempts have been proposed for specifying event-based applications [7, 9, 10]. Some of them include formal computational models for the event-based paradigm, techniques for specification of systems, and approaches for reasoning about the correctness of programs. The merit of these approaches are more about researching the fundamental behavior of this paradigm than about practical use.

5.1 Model-Checking Pub/Sub

In general, model checking aims at finding a representative finite state model of a system, and to explore all possible execution states for satisfaction of some properties. Many attempts exist that apply model checking to event-based applications in general, and pub/sub applications in particular [1, 12, 25].

Baresi et al. [1] introduce an approach to modeling and validating pub/sub applications. The modeling phase distinguishes between dispatchers that are defined by three characteristics (*Delivery*, *Notification*, *Subscription*), and other components described by UML statechart diagrams. The verification phase uses the model checker SPIN to prove properties that are defined by life-sequence charts (LSCs). These properties are transformed into automata, bundled with dispatcher and component descriptions, and transformed into Promela, the input language of SPIN.

5.2 Testing Pub/Sub

Model checking is usually intended for checking abstract models of software systems. Establishing a bridge between the correctness of the implementation of software systems and their model-checked abstract models is not mature.

Zhang et al. [25] introduce a source transformation-based framework for uniform testing and model checking of Implicit-Invocation (II) systems. The framework includes the Implicit-Invocation Language (IIL), a programming language designed for specifying II systems. Programs described in IIL are then transformed into both testing artifacts (Turing Plus programs) and verification artifacts (SMV programs). The test cases generated by this framework must be manually applied and evaluated. In contrast to this, our framework is based on a well-known programming language and leverages existing automated test generation and instrumentation frameworks.

5.3 Extending JML with temporal logic

To our knowledge, there is only one project that aims at extending JML with temporal behavior. While we translate LTL formulas into JML, Huisman et al. [24] extend JML with specific temporal operators, such as *after* and *before*, and liveness properties, such as *eventually* and *always*.

The tool JAG [13] provides a prototype implementation of this concept that translates formulas with said temporal operators into JML annotations. In contrast to this work, we use the well-known notion of LTL to specify temporal behavior, instead of introducing specific operators. This gives us the possibility to reuse and be compatible with all testing frameworks built around JML.

6 Evaluation

We evaluated our framework based on the "Set and Counter" example, which is widely used to illustrate the pub/sub paradigm [10, 25]. In this example, elements are added to and removed from a set component, while a counter component counts the number of elements in the set. The two components are loosely coupled since the communication is carried out by a pub/sub infrastructure. More precisely, the methods for incrementing/decrementing the counter are bound to the methods for adding/removing set elements. The set component is triggered by environment events *EnvAdd* and *EnvRemove*.

Figure 9 shows the event handler of the set component (Line 20 to 27), including its simplified LTL specification between `/*#` and `#*/` (Line 1 to 6), and the corresponding JML translation between `/*@` and `@*/` (Line 7 to 19).

The pre-condition (*PRE*) on Line 2 restricts the method argument's event type to start with *"Env"*. The post-condition (*POST*) from Line 3 to 5 consists of three conjunctions that correspond to the three *ensures* clauses from Line 9 to 18. The first conjunction prescribes to publish exactly one event, while the second defines that, on receiving an *EnvAdd* event, a *SetInsert* or *SetFull* event should be finally published. The third conjunction specifies to finally publish an event of type *SetDelete*, *SetEmpty*, or *SetElementDoesNotExist*, on receiving an *EnvRemove* event.

The LTL operator *F* is translated using the JML existential quantifier *\exists*, while the operator *G* would be translated using the JML universal quantifier *\forall*. The JML keyword *\old* provides a way of referring to the value of an expression prior to method invocation. After test generation, the tester defines different events (*IEvent*) which are used as arguments to validate that the method under test (*notifies*) conforms to the translated JML specification.

In addition to this simple example, we used RAY to test a real-world application. Palantír [23] extends existing Configuration Management Systems, such as CVS, by offering an awareness mechanism that provides insight into other workspaces. In this manner, it gives information about

```
/*#
PRE m:e:type PF "Env";
POST pub_counter == 1 &&
  m:e:type EQ "EnvAdd" -> F[e:type EQ "SetInsert" || e:type EQ "SetFull"] &&
  m:e:type EQ "EnvRemove" -> F[e:type EQ "SetDelete" || e:type EQ "SetEmpty" || e:type EQ "SetElementDoesNotExist"];
#*/
/*@ // --- generated by ltl2jml
requires e.getStringAttribute("type").startsWith("Env");
ensures pubsub.getPubCounter() == \old(pubsub.getPubCounter()) + 1;
ensures e.getStringAttribute("type").equals("EnvAdd") ==>
( \exists int i1; i1>=\old(pubsub.getMetaState().size()) && i1<pubsub.getMetaState().size();
  pubsub.getMetaState().getSnapShot(i1).getEventStringAttribute("type").equals("SetInsert") ||
  pubsub.getMetaState().getSnapShot(i1).getEventStringAttribute("type").equals("SetFull") );
ensures e.getStringAttribute("type").equals("EnvRemove") ==>
( \exists int i1; i1>=\old(pubsub.getMetaState().size()) && i1<pubsub.getMetaState().size();
  pubsub.getMetaState().getSnapShot(i1).getEventStringAttribute("type").equals("SetDelete") ||
  pubsub.getMetaState().getSnapShot(i1).getEventStringAttribute("type").equals("SetEmpty") ||
  pubsub.getMetaState().getSnapShot(i1).getEventStringAttribute("type").equals("SetElementDoesNotExists") );
@*/
public void notifies(IEvent e) {
  String type = e.getStringAttribute("type");
  String object = e.getStringAttribute("object");
  if (type.equals("EnvAdd"))
    add(object);
  else if (type.equals("EnvRemove"))
    remove(object);
}
```

Figure 9. Event handler of Set component

which developers currently change which artifacts. More details on testing Palantír with RAY are described in [19].

The experiences gained during the evaluation of RAY are promising. On the one hand, we found errors in our implementation of the "Set and Counter" example. On the other hand, we also detected errors in the specifications which were discovered by inspecting unexpected test failures. One of the main advantages of RAY is the use of temporal logic that enables to verify long execution traces without the need to calculate the expected outcome by hand. For instance, specifying that somewhere in the execution trace event *e* should be followed by event *f*, can be easily formulated as `F[e && X[f]]`. As shown in our examples, the usage of LTL for specifying the expected outcome is straightforward and does not impose a burden on the tester.

In contrast to this, the evaluation also revealed some drawbacks of our approach. On the one hand, the LTL-to-JML translation and particularly the JML compilation are time-consuming. In addition, the use of LTL and JML leads to increased file sizes (see Table 1), while the runtime assertion checking during test execution leads to slightly longer test execution times (see Table 2). On the other hand, the test generation tool currently used only detects errors that appear when calling one single method of the application under test. Therefore, we want to integrate more powerful test generation tools, such as Tobias and Jartege, to be able to automatically verify sequences of method calls.

The use of the Eclipse platform as foundation for the RAY framework turned out to be a good decision. The pluggable architecture of Eclipse enabled us to use existing software testing plug-ins for the construction of RAY. Furthermore, it eases to extend our framework with additional functionality, as we have experienced during the integration of JMLUnit.

7 Conclusion

Testing pub/sub applications remains fairly unexplored. The RAY framework presented in this paper provides a testing environment that alleviates the task of unit testing applications which use the pub/sub architectural style.

The RAY framework is targeted at the Eclipse platform and is built on top of several Eclipse plug-ins. We use existing plug-ins for test instrumentation and evaluation, as well as the JMLEclipse plug-in. The RAY plug-in itself provides a mock pub/sub infrastructure and a specification transformer. The first captures all published events, but avoids the delivery to the interested subscribers. The second is responsible for translating LTL specifications into JML. This enables the use of existing JML testing tools for the

	Type	*Plain*	*LTL*	*LTL & JML*	*%*
Set	source	2133	3075	6712	315%
	class	2760	2760	63717	2309%
Counter	source	1377	1807	3331	242%
	class	2140	2140	38603	1804%
Palantír EventProducer	source	15283	17252	22889	150%
	class	10804	10804	125605	1163%

Table 1. JML file size comparison (in bytes)

	Test Cases	*No RAC*	*RAC*	*%*
Set	85	187	266	142%
Counter	21	359	391	109%
Palantír EventProducer	1567	2864	3422	119%

Table 2. Test case execution time (in ms)

Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)
0-7695-2655-1/06 $20.00 © 2006 IEEE

construction of unit testing artifacts.

The architecture of the RAY framework leverages the pluggable architecture of the Eclipse platform. RAY therefore inherits the advantages of this development environment, as well as those of the testing frameworks on which it is built. The contribution of the RAY framework is a direct integration of pub/sub application testing into the Eclipse platform. Therefore, developers may use one single environment for implementing and testing said applications. In addition, the evaluation of our framework has already shown its practical use.

Despite these satisfying results, some issues remain open. The main point is to support integration testing where multiple pub/sub components are tested in combination with the real pub/sub middleware. Considering the "Set and Counter" example, one could then verify that set and counter finally always contain the same values. Furthermore, we plan to integrate more powerful test generation tools, such as Tobias [15]. Finally, the support for automated test data generation should be evaluated.

References

[1] L. Baresi, C. Ghezzi, and L. Zanolin. Modeling and validation of publish/subscribe architectures. In S. Beydeda and V. Gruhn, editors, *Testing Commercial-off-the-shelf Components And Systems*, pages 273–292. Springer Verlag, 2005.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*, Rome, Italy, July 2002.

[3] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.

[5] Y. Cheon, M. Y. Kim, and A. Perumandla. A complete automation of unit testing for java programs. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'05), Las Vegas, Nevada, USA*, pages 290–295. CSREA Press, June 2005.

[6] Y. Cheon and G. T. Leavens. The jml and junit way of unit testing and its implementation. Technical Report TR #04-02a, Department of Computer Science, Iowa State University, 2004.

[7] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasoning about Implicit Invocation. In *Proceedings of the 6th International Symposium on the Foundations of Software Engineering (FSE-6), Lake Buena Vista, FL, USA*, pages 209–221. ACM Press, November 1998.

[8] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[9] P. Fenkam, H. Gall, and M. Jazayeri. A Systematic Approach to the Development of Event-Based Applications. In *Proceedings of the 22nd Symposium on Reliable Distributed Systems (SRDS 2003), Florence, Italy*. IEEE Computer Society, October 2003.

[10] J. L. Fiadeiro and A. Lopes. A formal approach to event-based architectures. In *Fundamental Approaches in Software Engineering (FASE 2006)*, Vienna, Austria, March 2006. Springer-Verlag.

[11] D. Gabbay, M. Finger, and M. Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects*, volume 2. Oxford University Press, 2000.

[12] D. Garlan, S. Khersonsky, and J. S. Kim. Model checking publish-subscribe systems. In *Proceedings of the 10th International SPIN Workshop on Model Checking of Software (SPIN 2003)*, Portland, OR, USA, May 2003.

[13] A. Giorgetti and J. Groslambert. Jag: Jml annotation generation for verifying temporal properties. In *Fundamental Approaches in Software Engineering (FASE 2006)*, Vienna, Austria, March 2006. Springer-Verlag.

[14] IBM Corporation and The Eclipse Foundation. *Eclipse Platform Technical Overview*, December 2005.

[15] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron. Filtering tobias combinatorial test suites. In *Fundamental Approaches to Software Engineering (FASE 2004)*, pages 281–294, Barcelona, Spain, March 2004. Springer Verlag.

[16] E. Letier, J. Kramer, J. Magee, and S. Uchitel. Fluent temporal logic for discrete-time event-based models. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, Lisbon, Portugal*, pages 70–79. ACM, 2005.

[17] P. Louridas. Junit: Unit testing and coding in tandem. *IEEE Software*, 22(4):12–15, July/August 2005.

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

[19] A. Michlmayr, P. Fenkam, and S. Dustdar. Specification-based unit testing of publish/subscribe applications. In *Proceedings of the 5th International Workshop on Distributed Event-based Systems (DEBS 2006)*, Lisbon, Portugal, July 2006. IEEE Computer Society.

[20] H. Muccini, A. Bertolino, and P. Inverardi. Using software architecture for code testing. *IEEE Transactions on Software Engineering*, 30(3):160–171, 2004.

[21] C. Oriat. Jartege: A tool for random generation of unit tests for java classes. In *Proceedings of the 2nd International Workshop on Software Quality (SOQUA 2005)*, Erfurt, Germany, Sept. 2005.

[22] T. J. Parr and R. W. Quong. Antlr: A predicated-*ll(k)* parser generator. *Software - Practice and Experience*, 25(7):789–810, July 1995.

[23] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: raising awareness among configuration management workspaces. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 444–454, Portland, OR, USA, 2003. IEEE Computer Society.

[24] K. Trentelman and M. Huisman. Extending jml specifications with temporal logic. In *Proceedings of the 9th International Conference on Algebraic Methodology And Software Technology (AMAST'2002)*, pages 334–348. Springer-Verlag, 2002.

[25] H. Zhang, J. S. Bradbury, J. R. Cordy, and J. Dingel. Implementation and verification of implicit-invocation systems using source transformation. In *Proceedings of the 5th International IEEE Workshop on Source Code Analysis and Manipulation (SCAM 2005)*, Budapest, Hungary, Sept. 2005.