Specification-Based Unit Testing of Publish/Subscribe Applications*

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

Abstract

Testing remains the most applied verification method for software systems. Typically, the behavior of such systems is evaluated against their informal or formal specifications. In this paper, we consider an architecture-driven approach to software testing. We support the argument that, in many cases, the difficulties in testing can be alleviated by optimizing the test methodology to leverage the architecture of the application under test. To support this claim, we present a test framework for publish/subscribe applications. This paper evaluates our initial steps in this regard. We focus on the design of our framework, and illustrate how to accomplish unit testing of publish/subscribe applications against LTL specifications, considering a real-world application as example.

1 Introduction

Software quality is a widely desired software property. This desire led to a multitude of supporting techniques including formal verification, model checking, and testing. In the latter case, 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 testing [2, 15] proposes that the test methodology leverages the architecture of the application under test. The present paper supports this approach by showing that such test methodologies can deliver encouraging results even where formal methods and traditional testing techniques are still not mature. More precisely, we propose a framework for testing publish/subscribe applications.

In publish/subscribe (pub/sub) applications [8], communication between components is achieved by sending and receiving events. An event receiver (called *subscriber*) expresses its interest in events announced by other components (*publishers*) by means of *subscriptions*. The *pub/sub infrastructure* is responsible for persistency, management, and delivery of events to the interested subscribers.

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

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 the preferred integration basis for many application domains. Yet, it significantly complicates the verification process. In fact, to our knowledge, neither a formal nor an informal method for verifying pub/sub applications has emerged.

The framework RAY proposed in this paper aims at alleviating the task of testing pub/sub applications. Our long term goal is to provide facilities for automating test generation, unit testing, integration testing, oracle construction, test application, and test evaluation. The contribution of this paper is to give an outline of the initial steps towards this goal. In the first place, the design of RAY is presented. This design relies on specifications written in LTL and translated into JML. We show how such specifications are used in combination with JMLUnit and JUnit.

The remainder of this paper is organized as follows. We briefly introduce JML and LTL in Section 2, and position our work among related approaches in Section 3. The design of the RAY framework is presented in Section 4, while the use of this framework for unit testing purposes is illustrated in Section 5. We outline our early experiences and results in Section 6. Finally, Section 7 concludes the paper and points the way to future work.

2 Background

The Java Modeling Language (JML) [4] is a formal behavioral interface specification language for Java programs. By adding annotations to the Java source code (either after `//@` or between `/*@` and `@*/`), developers are able to specify not only pre- and post-conditions (*requires, ensures*) of methods, but also their normal behavior (*normal_behavior*), exceptional behavior (*signals, signals_only*), class invariants (*invariant*), frame axioms (*assignable*), and assertion statements (*assert*).

From a theoretical point of view, JML can be seen as a porting of the Design by Contract paradigm [14] 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 [5], Tobias [12], and Korat [3]. In this paper, the JMLUnit tool is used to automatically generate test oracles and test drivers from JML specifications that, on the other hand, are translated from LTL specifications.

Linear temporal logic (LTL) 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"*), which are defined in Figure 1. In this definition, an infinite trace h maps to each position i the set of propositions that hold at that position. The notation $(h, i) \models p$ means that the LTL formula p is true at position i of trace h. A trace h satisfies a formula p, noted $h \models p$, if it satisfies p at the initial position 0 (i.e., $h \models p \Longleftrightarrow (h, 0) \models p$).

$$\begin{array}{lcl}
(h,i) \models X\,p & \Longleftrightarrow & (h,i+1) \models p \\
(h,i) \models G\,p & \Longleftrightarrow & (h,j) \models p \text{ for all } j \geq i \\
(h,i) \models F\,p & \Longleftrightarrow & (h,j) \models p \text{ for some } j \geq i \\
(h,i) \models p\,U\,q & \Longleftrightarrow & (h,j) \models q \text{ for some } j \geq i \,\wedge \\
 & & (h,k) \models p \text{ for all } k \text{ s.t. } i \leq k < j \\
(h,i) \models p\,W\,q & \Longleftrightarrow & (h,i) \models p\,U\,q \,\vee \\
 & & (h,i) \models G\,p
\end{array}$$

Figure 1. Linear Temporal Logic (LTL)

Many extensions of LTL exist. Among them, the Metric Temporal Logic (MTL), which is also used in [13], is of particular interest for this paper, because it extends LTL with bounded temporal operators. For instance, the formula $F_{\leq 3}\ p$ is equivalent to $(\ p \vee X\ p \vee X\ X\ p \vee X\ X\ X\ p\ )$. MTL is also supported by the RAY framework.

3 Related Work

Few attempts have been proposed for specifying and verifying event-based applications in the past years [6, 9]. Some of these approaches 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 the pub/sub paradigm than about practical use. We consider the work in this paper as one way to leverage the experience gained in the development of such theories.

In general, model checking aims at finding a representative finite state model of a system, to explore all possible execution states for satisfaction of some properties. These properties are usually described as temporal logic formulas. Many attempts exist that apply model checking to event-based applications in general, and pub/sub applications in particular [1, 11, 18]; we overview two of them below.

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), instead of LTL formulas as usual in this context. These properties are transformed into automata, bundled with dispatcher and component descriptions, and transformed into Promela, the input language of SPIN.

When applied to software systems, current model checking approaches are more suitable for checking abstract models of software. Establishing a bridge between the correctness of the implementation of software systems and their model-checked abstract models is not mature.

Based on the work on model checking publish/subscribe by Garlan et al. [11], Zhang et al. [18] 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 real programming language and leverages existing automated test generation and instrumentation frameworks.

Fiege et al. [10] introduce the notion of *scopes* to build modular event-based systems that delimit the visibility of published events. The authors give a formal specification of their concept within a trace-based formalism adapted from temporal logic. While we focus on pub/sub applications, the authors specify a pub/sub middleware.

4 The RAY Testing Framework

The goal of the RAY framework presented in this section is to support and facilitate specification and testing of pub/sub applications.

4.1 Architectural Overview

The testing activity is normally divided into test planning, test generation, test application, 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. This framework is built by reusing existing testing frameworks where possible. The current focus is on unit testing where the application under test is tested without taking the pub/sub middleware into consideration. We assume that the middleware is already tested by the middleware vendor, and mainly concentrate on verifying the application logic.

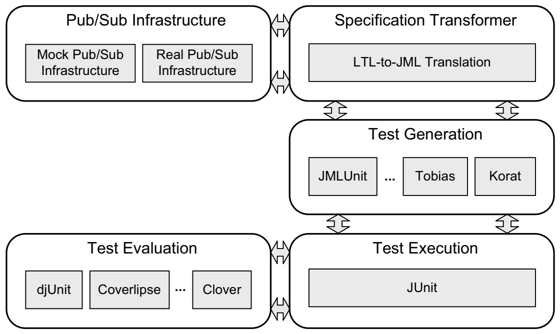

Figure 2. RAY Overview

Figure 2 illustrates the overview of the RAY framework. Statically, it includes a mock pub/sub infrastructure, a specification transformer, a test generator, a test instrumentation framework, and a test coverage tool. Testing a component requires writing an LTL specification that defines the behavior of said component. The specification transformer then translates such LTL formulas into JML. The mapping to JML allows us to reuse JML specific frameworks such as the test generators JMLUnit, Tobias, and Korat. In addition, we use the test instrumentation framework JUnit and the test coverage tools djUnit, Clover, and Coverlipse. One of the key components of our framework is the mock pub/sub infrastructure that allows us to abstract away the complexity of the pub/sub middleware and concentrate on the behavior of the unit under test.

4.2 Mock Publish/Subscribe

Unit testing a component is ideally done in isolation within a simple context. To reach this goal, stubs and mock objects are used as place holders for the components that do not need to be tested yet. In general, mock objects are determinant factors for the quality and simplicity of the unit testing process. 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.

These challenging requirements have guided the design of the RAY mock pub/sub infrastructure. After analysis of the many facets of pub/sub infrastructures, we retained two details that seem to be key to the testing process: 1) the set of events published by the component under test, and 2) the state of the publishing object. This information is extracted by our mock pub/sub infrastructure at runtime, and builds a sequence of states on which temporal reasoning is possible. In the meantime, this sequence of states constructs a meta-state on which traditional pre- and post-condition reasoning may be applied.

Letier et al. [13] 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 infrastructure supports the second technique.

```
1   public void publish(Event e, Object publisher) {
2      Vector publisherState = new Vector();
3      Class publisherClass = publisher.getClass();
4      Field[] f = publisherClass.getDeclaredFields();
5
6      for (int i=0; i<f.length; i++)
7          publisherState.add(cloneField(f[i]));
8      StateSnapShot s = new StateSnapShot(
9                           publisherClass.getName(), e,
10                          System.currentTimeMillis(),
11                          publisherState);
12     metaState.add(s);
13  }
```

Figure 3. Mock Pub/Sub Skeleton

A skeleton of the *publish* method of the RAY mock pub/-sub infrastructure is shown in Figure 3. The complete mock pub/sub has 200 lines of Java code. The second line of this skeleton declares a vector for storing the current state of the publisher. This is done on Line 6 and 7. Between Line 8 and 11, the complete state snapshot is constructed by adding the current time, the published event, and the name of the publishing object. The meta-state is finally updated on Line 12.

So far, we have focused primarily on one type of pub/sub participants: the publishers. In the same way, subscribers may also be tested by our framework. For this purpose, in addition to event publications, the mock pub/sub infrastructure captures all subscriptions including the state of the subscribing object. Therefore, the use of RAY also enables to verify if a component subscribes correctly, and if it behaves as expected on being notified.

4.3 Method Specification

Traditionally, pre- and post-conditions specify conditions about the initial and final states of a method. They are typically not intended for temporal formulas. The RAY mock pub/sub constructs a temporal logic layer on top of traditional pre- and post-condition specifications. This is done by translating temporal logic formulas expressed on the states of the unit under test into JML conditions expressed on the meta-state constructed by the mock pub/sub infrastructure.

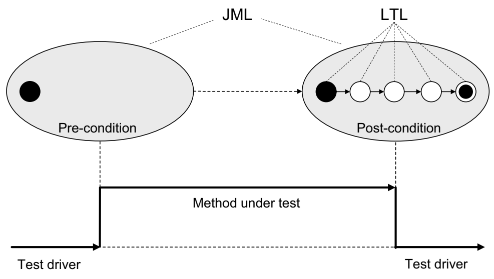

Figure 4. Meta-States of a sample method

This concept is illustrated in Figure 4. In this figure, a sample method is called by the test driver. The initial meta-state on which the pre-condition is formulated contains only the initial state of the method to be tested. The final meta-state on which the post-condition is formulated contains the sequence of states captured between the initial and final states of the method under test. In this way, the LTL formulas expressed on the captured traces are translated into first order logic formulas expressed on the meta-states. While we are currently using future temporal logic for specification, past temporal logic might also be used, as demonstrated in [7].

We assume that the components under test are terminating. Clearly, if the execution of a method does not terminate, the post-condition cannot be verified. Our current approach therefore does not cope with non-terminating methods. Consequently, the RAY framework considers finite traces, and hence, despite the use of LTL, only checks safety properties.

4.4 Specification Transformation

Attempts to enhance JML with temporal logic constructs have already be done [17]. The authors introduce additional keywords for specifying temporal properties in JML. In contrast to this, we propose to construct a layer on top of JML that is automatically translated into JML. This has the advantage of keeping JML to its core set while giving those needing LTL the possibility to benefit from the whole JML research. We are currently in the process of establishing the soundness of the transformation rules underlying our LTL-to-JML translator, which are presented in Figure 5.

In RAY, LTL specifications appear as special comments in the Java source code (either after `//#` or between `/*#` and `#*/`), that are processed by our LTL-to-JML translator and treated as usual comments by the JML and Java compilers. The pre-condition (on the meta-state) is declared using the keyword *PRE*, while *POST* refers to the post-condition. An example is given in Section 5.2.

$$\begin{array}{lll}
X\,p & \Longleftrightarrow & \texttt{(mock.getMetaState().getSnapShot(current+1); p)} \\
G\,p & \Longleftrightarrow & \texttt{(\textbackslash forall int i; i} \geq \texttt{current \&\&} \\
 & & \texttt{\&\& i} < \texttt{mock.getMetaState().size(); p)} \\
F\,p & \Longleftrightarrow & \texttt{(\textbackslash exists int i; i} \geq \texttt{current \&\&} \\
 & & \texttt{\&\& i} < \texttt{mock.getMetaState().size(); p)} \\
p\,U\,q & \Longleftrightarrow & \texttt{(\textbackslash exists int i; i} \geq \texttt{current \&\&} \\
 & & \texttt{\&\& i} < \texttt{mock.getMetaState().size(); q} \wedge \\
 & & \wedge \texttt{ (\textbackslash forall int j; j} \geq \texttt{current \&\& j} < \texttt{i; p))} \\
p\,W\,q & \Longleftrightarrow & \texttt{(\textbackslash exists int i; i} \geq \texttt{current \&\&} \\
 & & \texttt{\&\& i} < \texttt{mock.getMetaState().size(); q} \wedge \\
 & & \wedge \texttt{ (\textbackslash forall int j; j} \geq \texttt{current \&\& j} < \texttt{i; p))} \vee \\
 & & \vee \texttt{ (\textbackslash forall int k; k} \geq \texttt{current \&\&} \\
 & & \texttt{\&\& k} < \texttt{mock.getMetaState().size(); p)}
\end{array}$$

Figure 5. LTL-to-JML Translation rules

4.5 Implementation

We implemented a prototype of the RAY framework as plug-in for the Eclipse platform. The main reason for this choice was the pluggable architecture of Eclipse that eases the integration of existing plug-ins for test instrumentation (e.g., JUnit) and test coverage (e.g., djUnit, Clover, Coverlipse). In addition, our plug-in requires the JMLEclipse plug-in coordinated by the SAnToS laboratory at Kansas State University.

As a result, the RAY Eclipse plug-in provides a complete CASE tool for specification, implementation and validation of pub/sub applications.

5 Unit Testing with RAY

The following section uses Palantír as an example to illustrate how unit testing is performed in RAY.

5.1 Palantír

Configuration Management Systems, such as CVS, aim at coordinating the changes on a shared repository made by different developers in different workspaces. This coordination is achieved in a way, that no interference occurs during the changes. Palantír [16] complements existing configuration management tools by offering an awareness mechanism that provides insight into other workspaces, and gives information about which developers currently change which artifacts. The architecture of Palantír is based on the event notification service Siena that announces the activities of the developers by using different event types.

5.2 Method Specification

The specification transformation performed by our LTL-to-JML translator essentially maps temporal logic formulas into JML *requires* and *ensures* clauses. We consider the Palantír method *sendChangesCommittedEvent* as example.

As the name suggests, this method is responsible for publishing a *ChangesCommitted* event to notify that a new version of an artifact has been stored in the repository.

```
1  /*#
2  PRE m:artifactID!=null && m:nextArtifactID!=null;
3  POST F[e:type EQ "ChangesCommitted"] &&
4       G[sequencer!=null] && pub_counter<=1;
5  #*/
6  /*@ // --- generated by ltl2jml
7  requires artifactID != null;
8  requires nextArtifactID != null;
9  ensures ( \exists int i1;
10    i1>=\old(mock.getMetaState().size())
11        && i1<mock.getMetaState().size();
12    mock.getMetaState().getSnapShot(i1).
13    getEventStringAttribute("type").
14    equals("ChangesCommitted") );
15 ensures ( \forall int i1;
16    i1>=\old(mock.getMetaState().size())
17       && i1<mock.getMetaState().size();
18    mock.getMetaState().getSnapShot(i1).
19    getAttribute("sequencer") != null );
20 ensures mock.getPubCounter() <=
21    \old(mock.getPubCounter()) + 1;
22 @*/
23 public void sendChangesCommittedEvent(
24       ArtifactID artifactID, String comment,
25       ArtifactID nextArtifactID, String nextVersion,
26       String WSDest) throws SienaException {
27   ...
28 }
```

Figure 6. Palantír method specification

Figure 6 shows the LTL specification of this method (Line 1 to 5) and its corresponding translation into JML (Line 6 to 22). The pre-condition on Line 2 states that the method arguments *artifactID* and *nextArtifactID* must not be *null*. The post-condition on Line 3 instructs this method to finally publish an event of type *ChangesCommitted*. In addition, Line 4 defines that the state variable *sequencer* must not be *null* globally, and specifies that at most one event may be published ($pub\_counter <= 1$).

While the *requires* clauses on Line 7 and 8 are straightforward, the post-condition needs further explanations. According to the rules presented in Figure 5, the translation of the F operator between Line 9 and 14 uses the JML existential quantifier (*exists*), while the G operator is translated between Line 15 and 19 using the universal quantifier (*forall*). The *pub_counter* operator is translated by comparing the number of publications before method invocation and after method return (Line 20 and 21).

5.3 Test Generation

The test generator uses these JML specifications to construct the test oracle and the test driver. While the post-condition defines the expected behavior of a method, the pre-condition states if a test is meaningless. Meaningless test cases follow the argumentation that one cannot reason about correct or incorrect behavior, if the pre-condition of a method is not fulfilled.

Considering the Palantír case study, JMLUnit generates two test classes for each class under test. The first class contains the test oracle, that decides about the successful or unsuccessful test outcome, while the test data can be found in the second class. These two classes also provide a mechanism for running the test suite using JUnit, whereas the actual test data is filled by the tester. After calling the class constructor, the generated test driver invokes one method under test with a combination of the test data as arguments. JMLUnit therefore provides different strategies that define how method calls are generated from the test data.

5.4 Test Execution and Evaluation

Finally, the generated unit tests are executed with the help of JUnit. The test driver takes combinations of the test data as input and the test oracle verifies the results with respect to the method's post-condition. Failed test cases are signaled by runtime exceptions (e.g., *JMLNormalPostconditionError*) that refer to the violated statement of the JML specification. Furthermore, test coverage tools are used to measure the quality of the executed test suite by inspecting which lines and branches of the unit under test are covered by the test cases.

6 Evaluation

The set/counter example has been widely used to illustrate verification approaches of pub/sub applications [18]. While this example is simple, it indeed exhibits many of the features of real-world applications that make verification difficult. Yet, testing Palantír recalled us the importance of real-world examples.

Our assumption in the development of RAY was a direct invocation of the pub/sub infrastructure methods by the pub/sub components. However, with Palantír it has turned out that this is not always a valid hypothesis. In this specific case, the publisher does not directly publish the event, but delegates it to a sequencer which forwards it to the middleware after adding a sequence number. While this didn't have a major impact on testing Palantír, it suggested that verification approaches must consider the existence of such layers as they can be sources of errors.

In addition, testing Palantír constrained us to admit that depending on the power that needs to be given to the mock pub/sub infrastructure, it might be necessary to deviate from the typical interface provided by the middleware. More precisely, as our mock pub/sub infrastructure needs to access the state of publishing/subscribing objects, a reference to said objects is required. This constrained us to extend the signature of the methods for publication/subscription with a reference to the publishing/subscribing component.

Our current prototype was already successfully used to verify long program traces with several thousands of publications and subscriptions. However, scalability and performance of our approach have to be further investigated.

7 Conclusion

Verifying event-based applications in general and pub/sub applications in particular remains a difficult task. And, in fact, testing such applications remains fairly unexplored. The framework presented in this paper aimed at evaluating the architecture-based testing arguments by building a test methodology for applications using the pub/sub paradigm.

The aim of this paper was not to present a complete mature framework, but instead to evaluate our initial steps in this direction. In particular, the paper focuses on unit testing. In this sense, the RAY framework shows promising results capable of indeed producing a practical testing environment for pub/sub applications. The implementation of RAY leverages the pluggable architecture of Eclipse by reusing existing testing frameworks where possible. In addition, our prototype has already been applied to test a real-world application.

Despite these satisfying results, a lot remains to be done. Among many other points, we would like to extend the framework to apply integration testing, integrate more powerful automated test generation and instrumentation modules for RAY, multiply the number of case studies, and evaluate the support for Fluent Temporal Logic (FTL) [13].

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] A. Bertolino and P. Inverardi. Architecture-based software testing. In *Joint proceedings of ISAW-2 and Viewpoints '96*, pages 62–64, New York, NY, USA, 1996. ACM Press.

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

[4] 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.

[5] 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.

[6] 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.

[7] J. Dingel and H. Liang. Automating comprehensive safety analysis of concurrent programs using verisoft and txl. In *Proceedings of the International Symposium on Foundations of Software Engineering (ACM SIGSOFT 2004/FSE-12)*, Nov. 2004.

[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 IEEE Symposium on Reliable Distributed Systems (SRDS 2003), Florence, Italy*. IEEE Computer Press, October 2003.

[10] L. Fiege, G. Mühl, and F. C. Gärtner. A modular approach to build structured event-based systems. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 385–392, New York, NY, USA, 2002. ACM Press.

[11] 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.

[12] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron. Filtering tobias combinatorial test suites. In *Proceedings of ETAPS/FASE04 - Fundamental Approaches to Software Engineering*, pages 281–294, Barcelona, Spain, Mar. 2004. Springer Verlag.

[13] 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, New York, NY, USA, 2005. ACM Press.

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

[15] D. J. Richardson and A. L. Wolf. Software testing at the architectural level. In *Joint proceedings of ISAW-2 and Viewpoints '96*, pages 68–71, New York, NY, USA, 1996. ACM Press.

[16] 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.

[17] 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.

[18] 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.