Testing Abstract Behavioral Specifications *

Peter Y. H. Wong[1]**, Richard Bubel**[2]**, Frank S. de Boer**[3]**, Miguel Gómez-Zamalloa**[4]**, Stijn de Gouw**[3]**,
Reiner Hähnle**[2]**, Karl Meinke**[5]**, Muddassar Azam Sindhu**[6]

[1] SDL, Amsterdam
[2] Department of Computer Science, Technische Universität Darmstadt
[3] CWI, Amsterdam
[4] DSIC, Complutense University of Madrid
[5] School of Computer Science and Communication, KTH Royal Institute of Technology, Stockholm
[6] Department of Computer Science, Quaid-i-Azam University, Islamabad

Abstract We present a range of testing techniques for the Abstract Behavioral Specification (ABS) language and apply them to an industrial case study. ABS is a formal modeling language for highly variable, concurrent, component-based systems. The nature of these systems makes them susceptible to the introduction of subtle bugs that are hard to detect in the presence of steady adaptation. While static analysis techniques are available for an abstract language such as ABS, testing is still indispensable and complements analytic methods. We focus on fully automated testing techniques including blackbox and glassbox test generation as well as runtime assertion checking, which are shown to be effective in an industrial setting.

1 Introduction

Model-based testing is of particular importance in the context of complex *concurrent* and *highly variable* software systems. The nature of these systems makes them susceptible to the introduction of subtle bugs that are hard to spot and easy to overlook in the presence of steady adaptation. When developing software systems with high variability, for example, in the context of product line engineering [28], typically different products are generated that compute the same result (commonality) but which have differing non-functional requirements (variability), such as security levels, performance, etc. The availability of test cases with a good degree of code coverage is essential to ensure that these different products compute the same result.

In this paper we work with a model-centric approach based on the *Abstract Behavioral Specification* (ABS) language [18,15]. ABS is an industry-strength, executable modeling language intended for highly variable, concurrent, component-based systems. ABS software models abstract away from implementation details, but retain essential behavioral aspects. ABS has an easy-to-understand concurrency model, yet permits to model precisely synchronous as well as asynchronous operations with state changes. It has been carefully designed to make static analysis techniques feasible, including type checking, deadlock analysis, resource analysis, and even functional verification [12]. Static analyses provide formal assurances of the quality, correctness and trustworthiness of ABS models. Yet they do not render testing obsolete: functional verification is often expensive and non-automatic—formal verification cannot keep up with frequent changes that typically occur during development. In addition, analysis techniques address the correctness of source code or bytecode, but do not cover compilation to machine executable code or possible bugs in runtime environments. This is where model-based testing becomes important. A selection of tests with good coverage that are run on a regularly (e.g., nightly) basis, help to discover bugs at an early stage. In addition, to guard against regression, one may generate test cases from one product variant to validate the behavior of other or later versions. Variability in software systems clearly increases the need for testing. For this reason it is very valuable that the primitives provided by ABS to describe variability also allow one to cleanly separate testing code from production code as illustrated in the ABSUnit framework in Sect. 5 below.

Testing and glassbox test generation require the system under test to be executable. This renders testing a product-level rather than a family-level activity in product line engineering [28]. In this paper we do not discuss
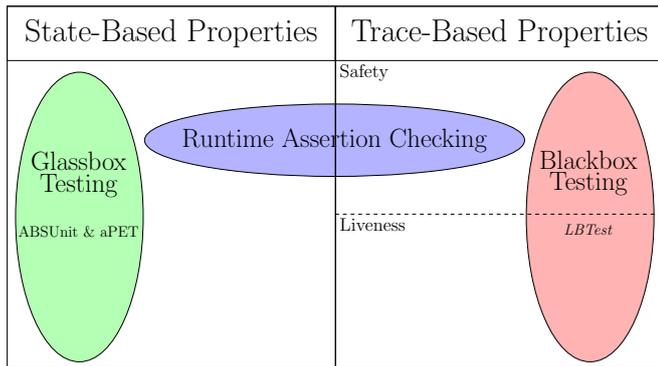

Figure 1. An overview of ABS testing techniques

testing at the family level, which is still an open research challenge. The issue is discussed further in Sect. 9.

Fig. 1 gives an overview of the ABS testing techniques and how they complement each other. Glassbox testing and test generation are realised on top of the ABSUnit framework and the aPET automatic test generator [1]. Glassbox techniques need access to the source code under test and are mainly suitable for testing state-based functional properties. In contrast to this, blackbox testing is used to test whether an ABS model satisfies trace-based safety or liveness properties. For this the learning-based testing tool *LBTest* [23] is used. *LBTest* does not require access to the source code and incrementally learns instead a model by observing system runs. Finally, runtime assertion checking (RAC) is used to complement glassbox and blackbox testing. It allows to check safety properties as well as state-based functional properties. Runtime assertion checking does not need explicit test cases, but instruments ABS models with assertions derived from given requirements.

Both static and dynamic analysis techniques are made available through the ABS tool suite [31]. The tool suite provides compiler backends that take ABS models and generate either executable programs in implementation languages such as Java and Scala, or rewriting systems in the language of Maude for simulation-based analyses.

In the following sections, we illustrate our testing techniques and the associated tools with an industrial case study that has been modelled with ABS [32]. The case study is described in Sect. 2. An overview of the ABS language is provided in Sect. 3. In Sect. 4 we focus on a language feature of ABS called Delta Modeling that permits modular and incremental specification of variability as well as systematic code reuse. The subsequent sections each cover one of our three testing techniques for ABS: Sect. 5 describes glassbox test generation; Sect. 6 describes run-time assertion checking, and Sect. 7 describes blackbox testing.

The purpose of this paper is *not* a detailed presentation of the theoretical foundations or the tools themselves, but to show how they are applied to a common case study and how they complement each other to in-

crease confidence in the correctness of a model. For the theory behind the employed testing techniques and detailed tool descriptions we refer to [1, 2, 11, 10, 24, 23].

Together, the technologies discussed in this paper constitute a comprehensive tool box for test automation suitable for a wide range of scenarios. While most testing approaches focus on one class of properties or on one testing approach, in this paper we demonstrate that the ABS platform plus Delta Modeling allow tightly integrated blackbox *and* glassbox, state-based *and* trace-based, static *and* dynamic testing. In Sect. 8 we show this to be the basis for a *concerted* usage of different testing approaches that exploits their complementarity.

2 An Industrial Case Study

The Fredhopper Access Server (FAS) is a distributed, concurrent OO system that provides search and merchandising services to e-Commerce companies. FAS provides to its clients structured search and navigation capabilities within the client's data. Fig. 2(a) shows the architecture used to deploy FAS at a customer site.

FAS consists of a set of live environments and a single staging environment. A live environment processes queries from client web applications via web services. A staging environment is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments according to a *Replication Protocol*. The Replication Protocol is implemented by a *Replication System*, which consists of a *SyncServer* at the staging environment and one *SyncClient* for each live environment. The SyncServer determines the schedule of replication jobs, as well as their contents, while SyncClient receives data and configuration updates according to the schedule.

Fig. 2(b) shows the interactions in the Replication System. Informally, the Replication Protocol is as follows: the SyncServer begins by listening for connections from SyncClients. A SyncClient creates and schedules a *ClientJob* object that connects to the SyncServer. The SyncServer then creates a *ConnectionThread* to communicate with the SyncClient's ClientJob. The ClientJob asks the ConnectionThread for a *replication*, receives a sequence of file updates according to the schedule from the ConnectionThread and terminates. A complete description of the protocol can be found in [32]. In this paper we focus on the behavior of SyncClient and ClientJob.

Previously we have modeled the Replication System in ABS [32]. In this paper we specify some high-level behavioral properties about the model from which test cases, test runs and assertions are derived. The model and the specifications are provided by software engineers at SDL Fredhopper. We have also taken this case study as a usability exercise of ABS language and its tool suite [31]. While the current production version of the
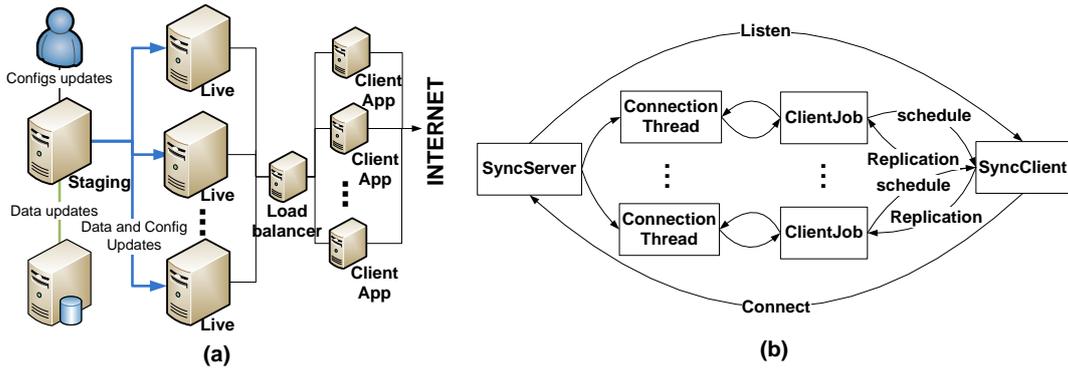
Figure 2. (a) An example FAS deployment and (b) Interactions in the Replication System

Replication System is implemented in Java, our aim is to conduct analyses on the ABS model and to generate executable production code of the Replication System from ABS. By conducting analyses on the ABS model, the generated production code would have much better guarantees over both verified and tested properties than the existing system.

3 Abstract Behavioral Modeling

ABS is an abstract, executable, object-oriented modeling language with a formal SOS-style semantics [18], targeting distributed systems with a high degree of variability. Many complex software systems, such as distributed services and consumer appliance software fall in this category.

Fig. 3 shows those parts of the layered architecture of ABS that are used throughout this paper: at the base are functional abstractions around a standard notion of parametric algebraic data types (ADTs). Next we have an OO-imperative layer similar to (but much simpler than) Java. The concurrency model of ABS is two-tiered: at the lower level it is similar to that of JCoBox [30] that generalizes the concurrency model of Creol [19] from single concurrent objects to concurrent object groups (COGs). COGs encapsulate synchronous, multi-threaded, shared state computation on a single processor. On top of this is an actor-based model with asynchronous calls, message passing, active waiting, and future types. An essential difference to thread-based concurrency is that task scheduling is *cooperative*, i.e., switching between tasks of the same object happens only at specific scheduling points during the execution, which are explicit in the source code and can be syntactically identified. This allows to write concurrent programs in a much less error-prone way than in a thread-based model and makes ABS models suitable for static analysis. Specifically, the ABS concurrency model excludes race conditions on shared data.

Fig. 4 shows some data types and interfaces used in the case study. The interface `ClientJob` models a Client-
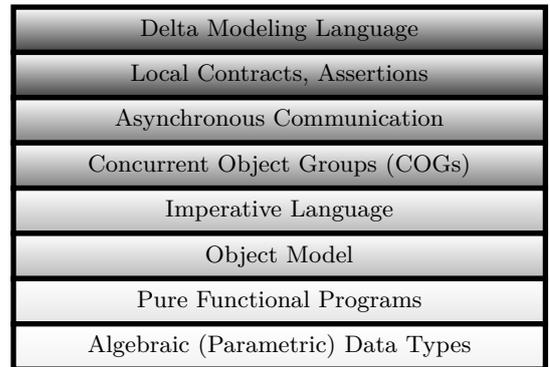

Figure 3. Layered Architecture of ABS

Job, while interface `DataBase` models the database of the underlying file system of the SyncClient. The algebraic data type (ADT) `Content` models the file system of FAS environments in ABS. ADTs allow specifying immutable values in functional expressions and to abstract away from implementation details such as hardware environment, file content, or operating system specifics. Specifically, `Content` is either a `File`, where an integer (e.g., its size) is taken to represent the content of a single file, or it is a directory `Dir` with a mapping of names to `Content`, thereby, modeling a file system structure with hierarchical name space.

Interface `ClientJob` has two methods: `register(sid)` takes an integer parameter that identifies the version of the data the replication would update the live environment to; it tests whether the live environment already contains this update (it also prepares the underlying database for a possible new incoming update, but this is irrelevant for our presentation). Method `file(id)` takes a `String` value specifying the absolute path to a file stored in the live environment and returns a `Maybe` value which is either an integer representing the file content or the value `Nothing` if no such file exists.

In interface `DataBase` the method `hasFile(id)` takes the absolute path to a file and tests whether this file exists in the live environment; `getContent(id)` also takes

```
1  data Content =
2    File(Int content) | Dir(Map<String,Content>);
3
4  interface ClientJob {
5    Bool register(Int sid);
6    Maybe<Int> file(String id);
7  }
8
9  interface DataBase {
10   Bool hasFile(String id);
11   Content getContent(String id);
12 }
```

Figure 4. Data types and Interfaces

```
1  def Bool isFile(Content c) =
2    case { File(_) => True; _ => False; };
3
4  class ClientJobImpl(DataBase db)
5  implements ClientJob {
6    Maybe<Int> file(String id) {
7      Fut<Bool> he = db!hasFile(id); await he?;
8      Bool hasfile = he.get;
9      Maybe<Int> result = Nothing;
10     if (hasfile) { // if1
11       Fut<Content> f = db!getContent(id);
12       await f?; Content c = f.get;
13       if (isFile(c)) { //if2
14         result = Just(content(c));
15       }
16     }
17     return result;
18   }
19 }
```

Figure 5. Method `file` and auxiliary function

a path to a file and returns a `Content` value representing the content of the file identified by the input parameter.

Fig. 5 shows the implementation of method `file(id)` in class `ClientJobImpl`. It has an instance field `db` of type `DataBase`. The ADT function `isFile(c)` takes a `Content` value and returns `True` iff value `c` records a file; `content(c)` is a partial *selector* function that returns the argument of the constructor `File` (line 14).

Method `file` is implemented using the ABS features of asynchronous calls, message passing, active waiting, and future types. It first calls `hasFile(id)` on object `db` asynchronously to access the underlying file system (line 7). This call spawns a new task and returns a *future* variable `he` as a place-holder for the result of the call to `hasFile(id)`. The statement "**await** he?" suspends the current task until `he` is resolved. The result can now safely (without blocking) be accessed with `he.get` (line 8).

```
1  delta AlternativePath;
2  modifies class ClientJobImpl {
3    modifies Maybe<Int> file(String id) {
4      id = "data2/" + id;
5      Maybe<Int> res = original(id);
6      return res;
7    }
8  }
```

Figure 6. Delta `AlternativePath`

4 Delta Modeling

ABS classes do not admit code inheritance and do not define types: all object type declarations are strictly to interfaces. Code reuse is, instead, realized in the paradigm of Delta-Oriented Programming [29]. The ABS Delta Modeling Language (DML) feature [7] implements delta-oriented programming in ABS. Deltas are named entities that describe the code changes associated with the realization of new features. The result is a separation of concern between variabilty at the architecural/design level and algorithmic/data type aspects. This helps early prototyping and avoids a disconnect between a system's architecture and its implementation.

For example, suppose we provide an alternative implementation of `ClientJobImpl` that accesses replication data at a different top-level directory. Fig. 6 shows a code delta `AlternativePath` that modifies the method `file` of class `ClientJobImpl`. Here the method takes a `String` value specifying the absolute path to a file and a new top-level directory as its prefix. The call **original** invokes the original implementation of `file` shown in Fig. 5, thereby achieving code reuse.

Deltas have similarities to aspects, however, their granularity is coarser (it is at the method level), they are more structured, and their application is explicitly invoked. This makes it possible to reason about the effect of changes to behavior caused by delta application [16].

Apart from addressing code reuse and variability, the DML also helps glassbox testing, in particular, for obtaining the preconditions (and invariants) of the system under test as well as for asserting its postconditions (and invariants). In the next section we shall see how deltas help to implement unit tests without code cluttering.

5 Glassbox Testing

Glassbox testing takes the software's internal structure into account, which is typical for unit testing or regression testing. We present an approach for (automated) test case generation (TCG) of glassbox tests for ABS. This comprises the tools ABSUnit—a JUnit-like testing framework—and aPET, a TCG tool.

```
1  [Suite] interface AbsUnitTest {
2    [Before] Unit setup();
3    [DataPoint] Set<Pair<Int,Int>> inputData();
4    [Test] Unit testMethod1(Pair<Int,Int> comp);
5  }
```

Figure 7. Typical ABSUnit test interface

5.1 Fundamental Approach

5.1.1 **The ABSUnit Framework**

ABSUnit is an instance of the well-known XUnit test framework [17]. As usual, the first step is to implement the ABSUnit tests and to group them into test suites. ABSUnit provides the annotations [DataPoint], [Before], [After] and [Test] to indicate the purpose of a method as data input provider for parametric tests, as a fixture to set up or shut down the test environment, or as an actual unit test. The annotation [Suite] is used for an interface representing a test collection.

Fig. 7 shows a typical annotated interface for a test suite. The actual test is provided by a class implementing the interface. To specify test oracles, ABSUnit provides assertion methods such as `assertEquals(Comparator)` or `assertThat(Matcher)` (inspired by Hamcrest, see `http://code.google.com/p/hamcrest/`).

As explained in Sect. 4, ABS strictly separates subtyping and code reuse. Only interfaces declare types and can subtype each other. For testing this has two main consequences: first, there is *no* root object and thus one cannot rely on a common interface and the presence of, for example, an `equals` method. Instead, `assertEquals` uses a comparator that knows how to compare two instances of a specific kind. Second, and more importantly, implementing tests often requires to access or to change class internals (e.g., to check intermediate results or to shortcut complex initialization procedures). Here, the DML of the previous section provides an elegant solution: instead of cluttering the code base with auxiliary code, all test-related changes are organized into separate test deltas. Those deltas are only selected during product testing, but are absent from the actually shipped product. In short, in ABS *test code becomes a product feature* that is selectable at product generation time.

ABSUnit generates glue code which is responsible for test creation, test invocation (with the input provided by datapoint methods) and for setting up the test environment using fixtures. The ABSUnit test executor runs the tests and records events such as test start, passed input parameters, scheduling decisions and the test status (pass, violated assertion, or deadlock). This information is used to present and explain the test outcome.

5.1.2 **Automatic TCG with aPET**

Automatic test generation is done with aPET. By analyzing the source code, glassbox TCG aims at automatically obtaining a small set of tests with a high code coverage degree. This is in contrast to random input data generators requiring an impractically large number of inputs to reach acceptable coverage. Moreover, the maintenance of vast test suites is also impractical.

Glassbox TCG is usually done by means of *symbolic execution* [20], which represents all program execution paths up to a certain threshold, obtaining a constraint system for each symbolic path. Constraints can be seen as path conditions whose fulfillment by input data ensures that execution takes such path. Hence, solutions to path constraints can be considered as test cases.

The system aPET realizes the *Constraint Logic Programming* (CLP)-based approach to TCG [14]. The backtracking-based evaluation mechanism and constraint solving facilities of CLP are well matched to the purpose of symbolic execution. The core schema consists of two independent phases: (i) the ABS program under test is translated into an equivalent CLP program, and (ii) the CLP program is symbolically executed in CLP relying on CLP's execution mechanism. This schema has the important property of being *flexible* and *generic*, in the sense that the second phase is essentially independent of the language for which symbolic execution has to be performed. The concrete features of the target language are abstracted in the translation and uniformly represented in CLP.

Application of this schema to the concurrent language ABS involves four steps:

1. Define an ABS to CLP compiler.
2. Implement the ABS concurrency-related operations in CLP. The scheduling policy definition is left parametric.
3. Define an appropriate coverage criterion for concurrent objects, with independent limits on both the number of task interleavings allowed and the number of loop unwindings performed in each parallel component.
4. Implement the generation of interleavings with tasks that could be initially present in the object's queue and whose execution can affect the execution of the method under test in case it suspends. See [1] for details.

5.2 Tool Description

The aPET engine is implemented in the Prolog CLP system. It is packaged as a binary executable with a command-line interface. Its integration within the ABS tool suite, which is implemented in Java as an Eclipse plugin, is realized as follows: In the ABS tool suite, a handler is activated when the user requests to generate

tests for a selected set of methods in the current ABS file. The handler collects a set of user-defined parameters and the *abstract syntax tree* of the ABS program under-test, and invokes the aPET engine. The parameters include among others, the coverage criterion, the scheduling policy and the level of task interleavings to be considered. The aPET engine then compiles the provided ABS program into a CLP program and symbolically executes it according to the the provided parameters. As a result, a set of tests is generated automatically for each requested method via XML. The aPET handler finally generates ABSUnit executable tests from the XML.

Let us observe that each test exercises a different path of execution and include an automatically synthesized test oracle. As no specifications are used, aPET generates the test oracles from the actual results of the program induced by the corresponding path constraints. With such test oracles all tests will trivially pass. Therefore, the test oracles can be seen as templates that the user has to confirm or to modify.

5.3 Case Study

Let us consider method `file` of class `ClientJobImpl` (see Fig. 5), and as coverage criterion, *path-coverage* limited to paths with at most one loop iteration or recursive call. Note that several functions involved in the computation of method `file` are recursive. Using this coverage criterion, aPET generates 6 tests, that correspond to the following situations:

(i) a file named "" is searched in an empty file system;
(ii) file "a" is searched in an empty file system;
(iii) file "a" is searched in a file system with just an empty folder named "a";
(iv) file "a" is searched in a file system with a folder named "a" that contains a file named "a";
(v) file "a" is searched in a file system with a folder named "" that contains a file named ""; and
(vi) file "a" is searched in a file system that just contains a file named "a".

In the first 5 tests the return value is `Nothing`, whereas in the last one the return value is `Just(0)` (0 being the content of the file). Strings are generated starting with the empty string, then generating alphabetically strings of length 1, etc.

Fig. 8 shows the test method `testFile` that is automatically generated for test case (vi) above. Its implementation first invokes `setHeap` (line 10) to set up the initial heap, which consists of two objects `c` and `b` of types `ClientJob` and `DataBase`. Next, method `file(id)` is called on `c` and asserts that the return value is as expected. It also invokes the generated method `assertHeap` to assert that the invocation of `file(id)` changed the heap as expected.

In addition, two delta modules are automatically created to provide additional infrastructure for executing

```
1  [Fixture] interface JobTest {
2    [Test] Unit testFile();
3  }
4
5  [Suite]
6  class JobTestImpl implements JobTest {
7    ClientJob c; DataBase b; ABSAssert aut;
8    { aut = new ABSAssertImpl(); }
9    Unit testFile() {
10     this.setHeap();
11     Maybe<Int> r = c.file("a");
12     aut.assertTrue(Just(0) == r);
13     this.assertHeap();
14   }
15   Unit setHeap() { }
16   Unit assertHeap() { }
17 }
```

Figure 8. Generated test case

```
1  delta MDeltaForClientJob;
2  adds interface MClientJob extends ClientJob {
3    Unit setDB(DataBase b);
4    DataBase getDB();
5  }
6  modifies class ClientJobImpl adds MClientJob {
7    adds Unit setDB(DataBase b) { this.db = b; }
8    adds DataBase getDB() { return db; }
9  }
```

Figure 9. Modification Delta

```
1  delta TestDelta;
2  modifies class JobTestImpl {
3   modifies Unit setHeap() {
4    b = new DataBase();
5    b.setRdir(Pair("r", Entries(InsertAssoc(
6      Pair("a",Content(0)),EmptyMap))));
7    c = new ClientJobImpl(null);
8    c.setDB(b);
9   }
10  modifies Unit assertHeap() {
11   DataBase x = c.getDB();
12   Pair<String,Content> p = x.getRdir();
13   aut.assertTrue(p == Pair("r", Entries(InsertAssoc(
14     Pair("a",Content(0)),EmptyMap))));
15  }
16 }
```

Figure 10. Test Delta

test cases. Delta module `MDeltaForClientJob`, displayed in Fig. 9, completes existing interfaces and classes to permit easy setup of their initial state. For example, it provides getter and setter methods for the database object. Similar delta modules exist for the other interfaces. The delta `TestDelta`, depicted in Fig. 10, modifies the methods `setHeap` (lines 3 − 9) and `assertHeap` (lines 10 - 15) to set up the initial heap and check the final heap. Here `TestDelta` initializes the underlying file system to a pair of `String` value "r" and `Entries(...)`, where "r" is the name of the top level directory of the file system and the `Entries` value models a file named "a" with content `0` (lines 5 − 8). The delta also asserts that this value does not change when `file(id)` is executed (lines 11 − 14).

6 Run-Time Assertion Checking

Run-time assertion checking (RAC) is a very useful technique for detecting faults, and it is applicable during any program execution context, including debugging, testing, and production. Compared to program logics, RAC emphasizes *executable specifications*. While program logics statically cover all possible execution paths, RAC is a fully automated, on-demand validation process which applies to the actual program runs.

Assertions are inherently state-based in that they describe properties of the program variables, i.e., fields of classes and local variables of methods. As such, assertions in general cannot be used to specify the *interaction protocol* or *history* (i.e., the trace of incoming and outgoing method calls or returns) between objects. This is in contrast to other formalisms such as message sequence charts and sequence diagrams. Nor do assertions support interface specifications (fundamental in ABS, as all object references are typed by interfaces), since interfaces are stateless and contain only method signatures. There exist many interesting approaches to run-time monitoring of histories, including PQL [22], Tracematches [3], JmSeq [26], LARVA [8], Jass [4], and JavaMOP [5]. However, none of these address the integration into the general context of run-time assertion checking: they allow specifying protocol-oriented properties, but do not provide a systematic solution to specify the data-flow of the valid histories. Hence, the question arises how to integrate protocol-oriented properties and assertions into a single formalism, in a manner amenable to automated verification, in particular to run-time checking.

6.1 Fundamental Approach

In [11] we identified attribute grammars with conditional productions and annotated with assertions as powerful and user-friendly specifications of histories. This approach was extended to coboxes in [10]. Grammars specify *invariant* properties of the ongoing behavior (of a single object, a COG, or an entire ABS model) and as such must be prefix-closed. Context-free grammars express the protocol structure (i.e., orderings between events) of the valid histories in a declarative manner. Context-free grammars, however, do not take data into account, such as actual parameters and return values of method calls. The question arises how to specify the *data flow* of the valid histories. To this end we extend the grammars with attributes. Terminals in the grammar have *built-in* attributes such as the actual parameters, return value and the identity of the caller and callee. Non-terminals have *user-defined* attributes which define data properties of sequences of terminals. Assertions annotating this attribute grammar then provide a natural way to express user-defined properties of these attributes. In other words, assertions specify the allowed attribute values of histories. This does not yet allow to directly express *data-dependent* protocols. Such protocols are quite common in practice, for example, the `next` method of a Java Iterator may not be called, whenever method `hasNext` was called directly before and returned false. Conditional productions address this problem.

To support focussing on a particular behavioral aspect of communication involving data-dependent protocols, we use the general mechanism of a *communication view*. A communication view is a partial mapping from events to grammar terminals. Events not associated to terminals are projected away and play no role in the grammar. This reduces the size of the histories, allows using intuitive names for the selected events and keeps the size and complexity of the grammars low. Moreover, communication views enable the introduction of abstractions of the communication by identifying two distinct events with the same grammar terminal.

In summary, the valid event histories are represented as words generated by an extended attribute grammar. Grammar productions (possibly conditional) specify the valid protocol structure of histories, while assertions express the valid data-flow of histories.

6.2 Tool Description

Our RAC combines three components: the parser generator ANTLR, the ABS compiler, and the meta programming system Rascal [21]. The ABS compiler generates JAVA code for the attribute definitions in the attribute grammar. The result is an attribute grammar defined in the syntax of ANTLR [27]. ANTLR, a JAVA parser generator, then generates a lexer and a parser for the grammar in JAVA.

Rascal is a general meta-programming language tailored for program transformations. We extended Rascal with support for ABS. Our RAC uses Rascal for several tasks: it first parses the communication view, the ABS method signatures, and the attribute grammar. Based on the parsing results, it generates code for a history

```
1  local view ClientJobProtocol specifies ClientJob {
2    return Bool register(Int sid) r,
3    call Maybe<Int> file(String id) f,
4    call Content DataBase.getContent(String id) c
5  }
```

Figure 11. Communication View

```
1  data List<A> = Nil | Cons(A head,List<A> tail);
2  def Bool contains<A>(List<A> ss, A e) =
3    case ss {
4      Nil => False ;
5      Cons(e, _) => True;
6      Cons(_, xs) => contains(xs, e);
7    };
```

Figure 13. `List` data type

class (a datatype suitable to represent the communication history of an ABS object or COG) and instruments ABS source code around method calls and returns to update the current history. The history class calls the JAVA parser (which was generated by ANTLR) when the history is updated to obtain new attribute values.

6.3 Case Study

We consider the `ClientJob` interface in Fig. 4 introduced in Sect. 3 with the following property: in a replication session, the `register(sid)` method is called initially with *sid* indicating the version of data the replication would update the client to. The method returns a `Bool` value indicating whether the client accepts this replication. If the returned value is `True` then the method `file(id)` may be called one or more times, each time with a unique `String` value representing the absolute path of a file. After each invocation of `file(id)`, an *outgoing* method invocation on `getContent(id)` of `Database` may be made with a value that must be the same absolute path as that supplied in the preceding method `file(id)`.

The communication view in Fig. 11 introduces the relevant events which can be referred to in the grammar by the terminals r, f, and c. Fig. 12 shows the attribute grammar formalizing the property stated informally above. Attribute definitions are written between normal brackets '(' and ')'. The first production formalizes the call to `register(sid)`, where the inherited attribute *rg* stores the return value and the attribute *ns* contains the `List` of file names processed so far by `file(id)` (initially, `Nil`). Note that epsilon productions are used to make the grammar prefix-closed, and that all attributes are inherited (i.e. passed down the parse tree) since the attributes of the non-terminals on the right-hand side of each grammar production are defined in terms of the attributes of the non-terminals on the left-hand side. The second production captures a call to `file(id)` and checks that the current `id` is new in *ns*. The condition "{ T.rg == True }?" formalizes that the value returned by `register(sid)` was `True`. The third production handles the outgoing call and checks that the filenames match. It also allows to call `file(id)` again via the non-terminal *T*.

Some data types used in the grammar are defined in Fig. 13. Function `contains(ss,e)` checks whether the list `ss` contains the element `e`, while `head(ss)` is a partial selector function that returns the first element of a non-empty list `ss`.

We have developed two versions of our RAC approach for Java and ABS. Using the Java version we have successfully integrated runtime assertion checking into the software lifecycle at SDL Fredhopper. Full detail can be found in [9]. Using the ABS version we have conducted experiments with the ABS model of the Replication System and have consequently detected crucial protocol violation in the model. Full detail of this case study can be found in [10].

7 Blackbox Testing

7.1 Fundamental Approach

Learning-based testing (LBT) [23] is an emerging paradigm for *black-box requirements-testing* that encompasses the three steps of: (i) automated test case generation (ATCG), (ii) test execution, and (iii) test verdict (the oracle step). LBT is related to model-based testing (MBT). However, where MBT starts from a system design model which is then used to generate test cases, in LBT a model is inferred automatically from an SUT implementation using computational learning methods (*reverse engineering*). This approach has advantages for testing systems which are undocumented, and for agile development methods where the cost of model development and model synchronisation with code updates is considered too high.

LBT is an iterative procedure that attempts to generate a large volume of high quality test cases. On each iteration, the currently inferred model is checked against a user requirement to search for a counterexample to requirement correctness. For this process, requirements must be formalised within a logic, such as first-order or temporal logic. This allows constraint solving or model checking technology to be used in the search for counterexamples. If a counterexample to correctness can be found, this must be executed on the system under test to determine whether it is a true negative or a false negative. True negatives can be returned as failed test cases. False negatives can be integrated, via a *learning algorithm*, into the inferred model to refine its accuracy. In

```
S    ::= ε |    r T (T.rg = r.result; T.ns = Nil;)
T    ::= ε |    { T.rg == True }? f { assert ! contains(T.ns, f.id); }
               V (V.ns = Cons(f.id, T.ns); V.rg = T.rg;)
V    ::= ε |    c { assert head(V.ns) == c.id; }
               T (T.ns = V.ns; T.rg = V.rg;)
```

Figure 12. Attribute Grammar for the ClientJob Behavior

this way, the inferred model will converge to a complete and correct model of the system under test, as increasing numbers of test cases are executed. Note that in the case that no counterexample can be found, some other test case generation method must be used to proceed with the iteration (see below). If the learning algorithm always converges correctly, and if counterexample search is a decidable problem, then LBT is a sound and complete method of testing. However, for large industrial SUTs, complete learning may not be feasible in the time available. For this reason, many optimisations of learning must be considered. One such optimisation is *incremental learning*, which can infer an incomplete model from relatively little test data. Such incomplete models can nevertheless uncover SUT errors. Further details about learning optimisation can be found in [23].

The Fredhopper access server is an example of a reactive system that can be learned as a state machine. Indeed any client-server architecture can be modeled and learned in this way. In this case, an *automata inference algorithm* is needed to reverse engineer models, and temporal logic is widely considered to be the most useful logic to formalise user requirements. Then efficient model checking algorithms can be employed to search for counterexamples. An early application of LBT to testing reactive systems was given in [24], and since then other classes of reactive systems have also been tested in this way (see e.g. [13]).

To interpret the testing results obtained for the Fredhopper access server, it will be helpful to consider in more detail the abstract LBT algorithm used. An LBT architecture automatically generates a large number of high-quality test cases by combining a model checking algorithm with a learning algorithm and a random test case generator. Note that *active learning algorithms*, which can generate their own queries are both appropriate (i.e. they can generate test cases) and efficient (i.e. in polynomial time). These three algorithms are integrated with the system under test (SUT) in an iterative feedback loop (see Fig. 14). On each iteration of this loop, a new test case is generated by one of the three TCG methods, i.e.: (i) model checking the most recent learned model m_n of the SUT against a formal user requirement Φ and choosing any counter example to correctness; (ii) using the active learning algorithm to generate a membership query; (iii) random test case generation. The LBT tool must *interleave* these three TCG methods to achieve an overall testing strategy that is efficient.

Whichever TCG method is used, the new test case i_n is then executed on the SUT with outcome o_n. The outcome of a test case is judged as a *pass*, *fail* or *warning*. This is done after each model checking step, by generating a predicted output p_n (obtained from m_n) that can be compared with the observed output o_n (from the SUT). Each new input/output pair (i_n, o_n) is used to update the current model m_n to a refined model m_{n+1}, which ensures that the iteration can proceed again. The overall LBT architecture is illustrated by the diagram in Fig. 14.

7.2 Tool Description

A platform for learning-based testing known as *LBTest* (see [25]) has been developed for blackbox testing of ABS and other reactive systems models. The *LBTest* tool supports the integration of different model inference algorithms with different model checkers to conduct experiments in learning-based testing. The main inputs to the tool are the SUT and a set of formal user requirements to be tested. For formal requirements modeling, the main language currently supported is *propositional linear temporal logic* (PLTL). LTL is chosen, since it naturally models the black-box (input/output) behaviour of reactive systems. The restriction of LTL to propositions only (i.e. PLTL rather than full first-order LTL) is because: (i) PLTL model checking is decidable, and (ii) there exist fast algorithms for model checking PLTL formulas such as BDD based methods.

Note that PLTL formulas can express both *safety properties* which may not be violated, and *liveness properties*, including *use cases*, which specify intended behaviors. Some liveness properties cannot be refuted in finite time (for example termination properties). For such types of properties, *LBTest* is able to issue a *warning verdict* that a test case has never been seen to have passed. Therefore, both types of requirements are amenable to testing using *LBTest*.

Currently in *LBTest*, only one model checker is supported, which is NuSMV [6]. This model checker has been adopted mainly for its stability and wide user base. In principle, any other model checker or even a bounded model checker for PLTL could also be used. The learning algorithm currently available in *LBTest* is the IKL learning algorithm described in [24], which is an algorithm for learning deterministic Boolean-valued Kripke structures.
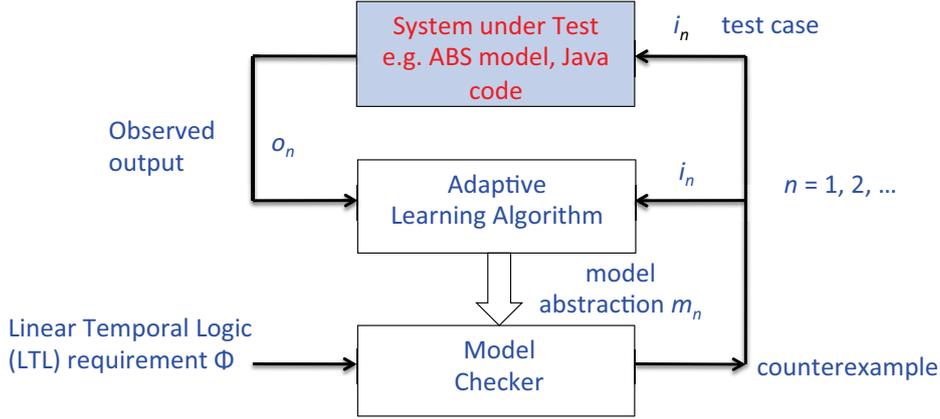
Figure 14. Architecture of learning-based testing

Other automata learning algorithms are currently being investigated for their performance in testing.

7.3 Tool Interface

For practical testing, propositional linear temporal logic is much too low level to express user requirements succinctly. Therefore, the language PLTL is augmented with finite symbolic data types, which support user defined data type declarations. A data type declaration Σ includes type declarations and value declarations. A type declaration defines a finite set of output types $t_1, ..., t_n$. (Note that a single input type, with the reserved type name *in*, is always assumed.) A value declaration for each type $input, t_1, ..., t_n$ consists of a finite set of constant symbols $c_1, ..., c_m$ of that type. This data type declaration constitutes the only interface specification needed for the system under test. Each symbolic data value (both input and output) must be mapped into a concrete native data value according to the programming language used for the SUT. This mapping code is stored in a thin wrapper between the SUT and LBTest, which also acts as a communication manager between LBTest as a server and the SUT as a client.

We were interested to test the interaction between a SyncClient and a ClientJob by learning the SyncClient as a deterministic Kripke structure (Moore machine) over the input data type

$$\Sigma_{in} = \{setAcceptor, \ schedule, \ searchjob,$$

$$businessJob, \ dataJob, \ connectThread,$$

$$noConnectionThread\}$$

Four relevant output data types were identified as follows:

$$\Sigma_{schedules} = \{\phi, \{search\}, \{business\},$$

$$\{business, \ search\}, \{data\}, \{data, \ search\},$$

$$\{data, \ business\}, \{data, \ business, \ search\}\}.$$

$$\Sigma_{state} = \{Start, WaitToBoot, Boot, WaitToReplicate,$$

$$WorkOnReplicate, End\},$$

$$\Sigma_{jobtype} = \{nojob, Boot, SR, BR, DR\},$$

$$\Sigma_{files} = \{readonly, writeable\}.$$

7.4 Case Study

The *LBTest* tool was applied to the problem of black-box testing an ABS model of the Fredhopper FAS case study described in Sect. 2. An executable SUT was obtained by compiling the ABS model into JAVA code, and writing a thin wrapper to encode the symbolic input and output data types in Java. A total of 11 user requirements were modeled in PLTL. For example, requirement 9 was: "*The SyncClient cannot modify its underlying file system ($files = readonly$) unless it is in state $WorkOnReplicate$.*" A PLTL formalisation is:

$$\mathbf{G} \ (state = WorkOnReplicate \rightarrow$$

$$\mathbf{X} \ (files = writable \ \mathbf{U} \ state \in \{End, WaitToReplicate\})$$

$$\wedge \ state \neq WorkOnReplicate \rightarrow$$

$$\mathbf{X} \ (files = readonly \ \mathbf{U} \ state = WorkOnReplicate))$$

Table 1 gives the results obtained by running *LBTest* to test these 11 user requirements on the FAS SyncClient. For each requirement, Table 1 breaks down the *total number* of test cases used into three figures (columns 5, 6 and 7) which count the test cases generated by each of the three different TCG methods: *model checker*, *learner* and *random*. The total testing time (column 3) is the total time taken to execute all three types of test cases, which were interleaved. For each requirement, Table 1 gives the final verdict (column 2) i.e. *pass/fail/warning*. Column 4 gives the size of the learned hypothesis model at test termination. To terminate each

experiment, a maximum time bound of 5 hours was chosen. However, if the hypothesis model size had not changed over 10 consecutive random tests, then testing was terminated earlier than this.

Thus for example: Requirement 1 was tested for a total of 5 hours using 50,942 test cases, of which 50,897 were generated by the learning algorithm, 45 were generated randomly, and 0 were generated by the model checker. We see that learner generated queries dominate, though generally this is influenced by the kind of learning algorithm used (here IKL). In fact, looking across all requirements we can see that the ratio of random plus model checker queries to learner queries is about 1:1000. This means that each new model m_{n+1} is inferred from m_n after intervals of about 1000 learner queries. This ratio is a property of the IKL learning algorithm itself, and can only be influenced by choosing other learning algorithms.

Around 10,000 test cases per hour were generated, executed and evaluated. We can see that this test throughput does not vary much across the 11 different requirements. On large SUTs, test throughput is mainly determined by the average execution speed of a single test case. Since Requirement 1 was passed, while 45 random and 0 model checker test cases were used, we can infer that the model checker was called 45 times, but on each occasion it failed to find a counterexample, so that a random test case was used instead.

Finally notice that the number of states in the final hypothesis automaton is rather small (8 states). The other requirements yield hypothesis automata of similar sizes. These figures suggest that while the total state space of the access server is almost certainly very large (a completely accurate model would have an infinite state space), the system abstraction learned by LBTest to analyse each specific property can be quite small. Nevertheless, it is not clear whether complete learning of the state space has been achieved for any requirement. Complete learning is not only difficult to achieve, in a black-box testing regime it is even difficult to detect. For we have no direct access to the SUT code, and in any case equivalence checking the SUT code (an arbitrary program) with the learned model is infeasible. This highlights the importance of incremental learning in black-box testing context. The development of appropriate *coverage models*, to answer this question in a relative way, is an important open problem for the field.

Nine out of eleven requirements were passed. For requirements 8 and 9, *LBTest* gave warnings corresponding to tests of liveness requirements that were never seen to have passed. A careful analysis of these requirements showed that both involved using the U (strong until) operator. When this was replaced with a W (weak until) operator no warnings for Requirement 9 were seen. Recall that under the *strong interpretation* of p until q, written pUq, then q must eventually become true. However under the *weak interpretation* of p until q, written

Table 2. Metrics of JAVA and ABS of the Replication System

Metrics	JAVA	ABS
Nr. of lines of code	6400	3300
Nr. of classes	44	40
Nr. of interfaces	2	43
Nr. of data types	N/A	17

pWq, then q may never become true if p holds forever. Thus using W instead of U usually gives a weaker user requirement that is easier to satisfy, i.e. less likely to yield test errors.

After replacing U by W, LBTest continued to produce warnings for Requirement 8. The final conclusion is that *LBTest* had successfully identified one error in the requirements and one error in the SUT.

8 Discussion

The ABS model of the Replication System considered in the case studies forms a part of the Fredhopper Access Server (FAS) whose current in-production JAVA implementation has over 150,000 lines of code, of which over 6,000 lines constitute the Replication System. Due to its concurrent behavior and the presence of numerous features, the Replication System is one of the most complex parts of FAS.

Table 2 shows metrics for the actual implementation and the ABS model of the Replication System. When comparing the numbers it is important to know that the ABS model includes modeling-level aspects such as deployment components and simulation of external inputs in the ABS model, which the JAVA implementation lacks. The ABS model includes also scheduling information, as well as models of file systems and data bases, whereas the JAVA implementation leverages libraries and its API. This accounts for >1,000 lines of ABS code. The construction of the first version of the ABS model took around 3 person months. The model was subsequently revised and extended to capture other behavioral aspects of the Replication System, such as timing information and variability. Furthermore, while the JAVA implementation is a relatively stable part of FAS, bugs had been identified and fixed. When there was a change in the JAVA implementation, the ABS model was then updated accordingly.

The quality assurance process at Fredhopper (as in many other software companies) includes automated testing. Unit tests are written manually to validate the behavior of methods and to detect regressions. A continuous integration server executes all unit tests every time a change is done to the code base of the product. To leverage the results reported in this paper, manually defined unit tests can be replaced by high coverage test cases *automatically* generated by aPET. System tests, on

Table 1. Performance of *LBTest* on the FAS case study

PLTL Req	Verdict	Total testing time (hours)	Hypothesis size (states)	MC queries	Learner queries	Random queries
Req 1	pass	5.0	8	0	50,897	45
Req 2	pass	5.0	15	2	49,226	13
Req 3	pass	1.7	11	0	16,543	17
Req 4	pass	2.1	11	0	20,114	14
Req 5	pass	2.5	11	0	24,944	17
Req 6	pass	2.3	11	0	23,215	16
Req 7	pass	2.1	11	0	18,287	17
Req 8	warning	1.9	8	15	18,263	12
Req 9	warning	3.8	15	18	35,831	18
Req 10	pass	2.7	11	0	26,596	19
Req 11	pass	4.6	11	0	45,937	21

the other hand, are executed twice a day on instances of FAS on a server farm. Two types of system tests are scenario and functional testing. Scenario testing executes a set of programs that emulate a user and interact with the system in predefined sequences of steps (scenarios). At each step they perform a configuration change or a query to FAS, make assertions about the response from the query, etc. Function testing executes sequences of queries, where each query-response pair is used to decide on the next query and the assertion to make about the response. Both types of tests require a running FAS instance and can be augmented with RAC techniques described in Sect. 6. Moreover, by formalising scenarios using PLTL, scenario testing can be augmented with blackbox testing using *LBTest*. In summary, the various testing approaches provided for ABS models have the potential to *substantially increase automation and coverage at the unit, scenario, and function testing level.*

The three test approaches discussed here should be used in concertation in such a way that their complementarity can be exploited. To give one example, given a high-level specification with ABS interfaces, one can generate test cases from class implementations using aPET to validate whether the implementations match the specification. We demonstrated this in Sect. 5 when we generated tests for the `ClientJobImpl` that cover all paths specified by a given coverage criteria. Another example is the combined application of *LBTest* and RAC during system testing. RAC makes assertions about object interaction which are specified in terms of attribute grammars as exemplified by our specification of a property of the `ClientJob` protocol. However, RAC checks those assertions only if corresponding execution paths are visited during a system run. Conversely, *LBTest* actively interacts with the SUT to learn a model that is then checked against PLTL formulae. This means *LBTest* attempts to trigger the execution paths corresponding to the formulae. Restricting the specification of properties to PLTL makes proving such properties on the model decidable. Note that *LBTest* checks both safety and liveness prop-

erties while run-time assertion checking aims merely at safety properties.

To achieve scalability and full automation at the same time, it was essential to work in a model-based framework. Our results would not have been possible at the level of implementation languages, such as JAVA or C++. On the other hand, it is perfectly possible to compile ABS test cases and runtime assertion checks into any of the target languages supported by ABS code generation, which includes JAVA.

We stress that, while ABS is a modeling language, it implements such concepts as interfaces, shared heap access, and asynchronous concurrent execution. This permits precise modeling and realistic simulation [31]. Delta modeling not only permits to factor out the commonality in modeled software, but is pragmatically very useful to achieve a clean separation between test code and productive code at proudct build time.

9 Conclusion

We presented a modeling framework based on the language ABS that enables extensive test automation for a complementary, yet fully integrated set of testing approaches. All techniques are fully implemented and were evaluated with an industrial case study. The different testing techniques cover different kinds of properties and complement each other with respect to their requirements such as having access to source code, or the availability of specifications in the form of assertions or temporal logic formulas (see also Fig. 1). We showed in particular that testing can be performed on models of highly distributed systems, and, even further, how formal methods enable us to automate large parts of testing and test case generation.

As future work, we would like to lift automated testing techniques from the product to the family level in product line engineering [28]. First ideas on how to approach this exist, such as sharing test cases (and test

runs) between products in case the products are identical or overlap with respect to the executed code. Work in the direction of compositionality [2] of glassbox test generation exhibits further potential to produce reusable test cases. In black box requirements testing, it becomes important to integrate product variability points into formal requirements languages such that during application engineering [28], when variability points are being instantiated for specific products, requirements may also be instantiated for those products.

Acknowledgements

References

1. Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Towards Testing Concurrent Objects in CLP. In Agostino Dovier and Vítor Santos Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*, volume 17 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 98–108, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

2. Elvira Albert, Miguel Gómez-Zamalloa, José Miguel Rojas, and German Puebla. Compositional CLP-based test data generation for imperative languages. In *LOPSTR 2010 Revised Selected Papers*, volume 6564 of *LNCS*. Springer-Verlag, 2011.

3. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 345–364, New York, NY, USA, 2005. ACM.

4. Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass - Java with Assertions. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 55(2):103 – 117, 2001.

5. Feng Chen and Grigore Roşu. Mop: an efficient and generic runtime verification framework. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 569–588, New York, NY, USA, 2007. ACM.

6. Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A New Symbolic Model Verifier. In *Proc. of CAV 1999*, volume 1633 of *LNCS*, 1999.

7. Dave Clarke, Nikolay Diakov, Reiner Hähnle, Einar Broch Johnsen, Ina Schaefer, Jan Schäfer, Rudi Schlatte, and Peter Y. H. Wong. Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 417–457. Springer-Verlag, 2011.

8. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper). In *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, SEFM '09, pages 33–37, Washington, DC, USA, 2009. IEEE Computer Society.

9. Frank S. de Boer, Stijn de Gouw, Einar Broch Johnsen, and Peter Y. H. Wong. Run-time assertion checking of data- and protocol-oriented properties of java programs: An industrial case study. *LNCS Transactions on Aspect-Oriented Software Development (TAOSD)*, 2013. Special Issue on Runtime Verification and Analysis. To appear.

10. Frank S. de Boer, Stijn de Gouw, and Peter Y. H. Wong. Run-time verification of coboxes. In *Proceedings of 11th International Conference on Software Engineering and Formal Methods*, volume 8137 of *LNCS*, pages 259–273, 2013.

11. Stijn de Gouw, Jurgen Vinju, and Frank de Boer. Prototyping a tool environment for run-time assertion checking in JML with Communication Histories. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*, FTFJP '10, pages 6:1–6:7, New York, NY, USA, 2010. ACM.

12. Analysis Final Report, December 2012. Deliverable 2.7 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

13. Lei Feng, Simon Lundmark, Karl Meinke, Fei Niu, Muddassar A. Sindhu, and Peter Y. H. Wong. Case studies in learning-based testing. In *Proc. Twenty Fifth IFIP Int. Conf. on Testing Software and Systems (ICTSS 2013)*, volume 8254 of *LNCS*, pages 164–179. Springer, 2013.

14. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *Theory and Practice of Logic Programming, 26th Int'l. Conference on Logic Programming (ICLP'10) Special Issue*, 10 (4–6):659–674, July 2010.

15. Reiner Hähnle. The Abstract Behavioral Specification language: A tutorial introduction. In Marcello Bonsangue, Frank de Boer, Elena Giachino, and Reiner Hähnle, editors, *International School on Formal Models for Components and Objects: Post Proceedings*, volume 7866 of *Lecture Notes in Computer Science*, pages 1–37. Springer-Verlag, 2013.

16. Reiner Hähnle, Ina Schaefer, and Richard Bubel. Reuse in software verification by abstract method calls. In Maria Paola Bonacina, editor, *Proc. 24th Conference on Automated Deduction (CADE), Lake Placid, USA*, volume 7898 of *Lecture Notes in Computer Science*, pages 300–314. Springer-Verlag, 2013.

17. Paul Hamill. *Unit Test Frameworks*. O'Reilly Media, November 2004.

18. Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2011.

19. Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):35–58, March 2007.

20. J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

21. Paul Klint, Tijs van der Storm, and Jurgen Vinju. RAS-CAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 168–177, Washington, DC, USA, 2009. IEEE Computer Society.

22. Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 365–383, New York, NY, USA, 2005. ACM.

23. K. Meinke, F. Niu, and M. Sindhu. Learning-Based Software Testing: A Tutorial. In Reiner Hähnle, Jens Knoop, Tiziana Margaria, Dietmar Schreiner, and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, Communications in Computer and Information Science, pages 200–219. Springer-Verlag, 2012.

24. K. Meinke and M. Sindhu. Incremental learning-based testing for reactive systems. In *Proc Fifth Int. Conf. on Tests and Proofs (TAP2011)*, number 6706 in Lecture Notes in Computer Science, pages 134–151. Springer-Verlag, 2011.

25. Karl Meinke and Muddassar A. Sindhu. LBTest: A learning-based testing tool for reactive systems. In *Proc ICST-13, Sixth IEEE Int. Conf. on Software Testing, Verification and Validation*, pages 447–454. IEEE Computer Society, 2013.

26. B. Nobakht, M. M. Bonsangue, F. S. de Boer, and S. de Gouw. Monitoring method call sequences using annotations. In *Proceedings of the 7th international conference on Formal Aspects of Component Software*, FACS'10, pages 53–70, Berlin, Heidelberg, 2012. Springer-Verlag.

27. Terrence Parr. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, 2007.

28. Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

29. Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proc. of 14th Software Product Line Conference (SPLC 2010)*, September 2010.

30. Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *European Conference on Object-Oriented Programming (ECOOP'10)*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer-Verlag, June 2010.

31. Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *Journal on Software Tools for Technology Transfer*, 14(5):567–588, 2012.

32. Peter Y. H. Wong, Nikolay Diakov, and Ina Schaefer. Modelling Distributed Adaptable Object Oriented Systems using HATS Approach: A Fredhopper Case Study. In *Proc. of FoVeOOS 2011*, volume 7421 of *LNCS*, 2012.