# Testing Abstract Behavioural Specifications[☆]

Peter Y. H. Wong[a], Richard Bubel[b], Frank S. de Boer[c], Miguel Gómez-Zamalloa[d], Stijn de Gouw[c], Reiner Hähnle[b], Karl Meinke[e], Muddassar Azam Sindhu[e]

[a] *SDL Fredhopper, Amsterdam, The Netherlands*
[b] *Department of Computer Science, Technische Universität Darmstadt*
[c] *CWI, Amsterdam*
[d] *DSIC, Complutense University of Madrid*
[e] *School of Computer Science and Communication, KTH Royal Institute of Technology, Stockholm*

**Abstract**

We present a range of testing techniques for the Abstract Behavioural Specification (ABS) language and apply them to an industrial case study. ABS is a formal modelling language for highly variable, concurrent, component-based systems. The nature of these systems makes them susceptible to introduction of subtle bugs hard to spot and oversee in particular 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.

*Keywords:* automated testing; industrial case study; blackbox testing; glassbox testing; runtime assertion checking

## 1. Introduction

The Abstract Behavioural Specification (ABS) language [7, 12] is a formal executable modelling language intended for highly variable, concurrent, component-based systems. ABS models abstract away from implementation details, but retain essential behavioural aspects. ABS has been carefully designed to make static analysis techniques feasible, including type checking, resource analysis, and even functional verification. These analyses provide formal statements about the quality, correctness and trustworthiness of ABS models. Yet they do not render testing obsolete: functional verification is often expensive and non-automatic—one cannot afford to run expensive analyses every time after an ABS model has been modified. In addition, analysis techniques do not cover binary code or 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.

[☆] Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu`).

This is of particular importance as ABS targets complex concurrent and *highly variable* software systems. The nature of these systems makes them susceptible to the introduction of subtle bugs which are hard to spot and oversee, in particular, in the presence of steady adaptation. When developing variable software systems, such as those in product line engineering [21], it is common that systems are developed which provide different implementations to achieve the same result (commonality) but differ in aspects such as resource requirements, security etc. (variability). In this context, having a rich set of test cases, such as a set of unit tests with a good degree of code coverage, is crucial. Since such a set would allow to check that the different implementations compute the same results. For example, to guard against regression, one may generate such test cases from one version of the software (core product) to validate the correctness of other versions. While variability in software systems increases the need for testing, we can also make use of the primitives provided by ABS to describe such variability to separate testing code from production code as done by the ABSUnit framework discussed in Sect. 5.

Testing and test generation techniques require the system under test to be (symbolically) run. Hence, testing is closer to the product level than to the family level. In this paper we do not discuss testing on the family level, which is still an active research question. We defer the discussion of possible future works in Sect. 9.

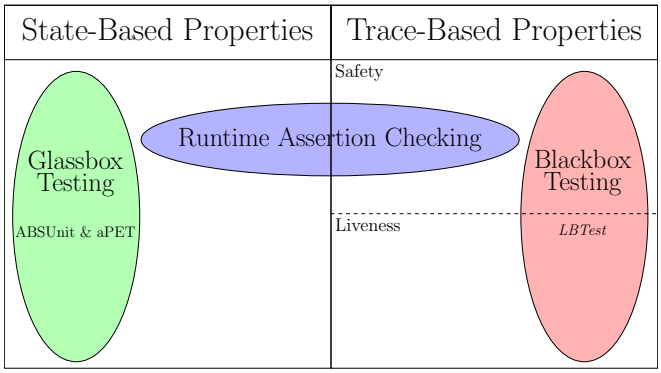


Figure 1: An overview of ABS testing techniques

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. Glassbox techniques need access to the source code under test and are mainly suitable for testing state-based functional properties. Conversely, blackbox testing is used to test whether an ABS model satisfies trace-based safety or liveness properties. To this end, the learning-based testing tool *LBTest* is used. *LBTest* does not require access to the ABS source code and incrementally learns instead a model by observing system runs. Runtime assertion checking (RAC) is used as an intermediate between glassbox and blackbox testing. It allows to test for 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.

In the following, we present tool-supported testing techniques for ABS applied to

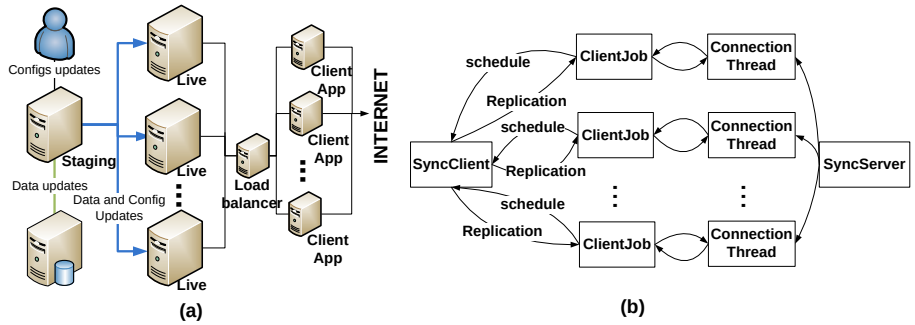


Figure 2: (a) An example FAS deployment and (b) Interactions in the Replication System

an industrial case study [24] (described in Sect. 2). An overview of the ABS language, illustrated with the the case study is in Sect. 3. Sect. 4 focuses on a sub-language of ABS called Delta Modelling Language (DML) that allows modular and incremental specification of variability as well as code reuse. The following sections cover our three testing techniques for ABS in turn: Sect. 5 describes glassbox test generation; Sect. 6 describes run-time assertion checking, and Sect. 7 describes blackbox testing. Together, these technologies constitute a comprehensive tool box for test automation suitable for a wide range of scenarios. Sect. 8 discusses the relevance of these testing techniques for the case study and how each testing technique complements each other in the context of industrial software development. We conclude this paper in Sect. 9.

## 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 deployment architecture used to deploy FAS to a customer.

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 the *Replication Protocol*. The Replication Protocol is implemented by the *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

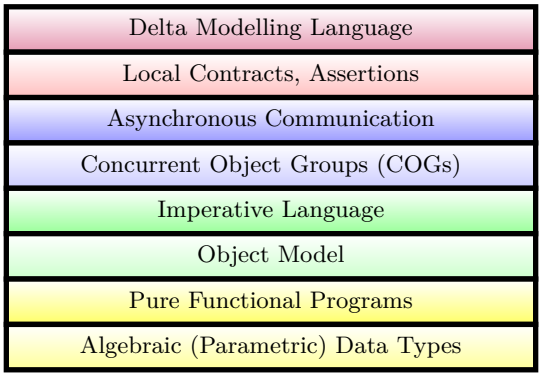


Figure 3: Layered Architecture of ABS

and terminates. A complete description of the protocol can be found in [24]. In this paper we focus on the behaviour of SyncClient and ClientJob.

## 3. Abstract Behavioural Specification

ABS is an abstract, executable, object-oriented modeling language with a formal semantics [12], targeting distributed systems. 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 [23] that generalizes the concurrency model of Creol [13] 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.

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

```
data Content = File(Int content) | Dir(Map<String,Content>);

interface ClientJob {
  Bool register(Int sid);
  Maybe<Int> file(String id);
}

interface DataBase {
  Bool hasFile(String id);
  Content getContent(String id);
}
```

Figure 4: Data types and Interfaces

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 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 the `c` records a file; `content(c)` is a partial *selector* function that returns the argument of the constructor `File`.

```
def Bool isFile(Content c) = case { File(_) => True; _ => False; };

class ClientJobImpl(Database db) implements ClientJob {
  Maybe<Int> file(String id) {
    Fut<Bool> he = db!hasFile(id); await he?;
    Bool hasfile = he.get;
    Maybe<Int> result = Nothing;
    if (hasfile) { // if1
      Fut<Content> f = db!getContent(id);
      await f?; Content c = f.get;
      if (isFile(c)) { //if2
        result = Just(content(c));
      }
    }
    return result;
  }
}
```

Figure 5: Method `file` and auxiliary function

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. 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**.

## 4. Delta Modelling

ABS classes do not implement 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 [22]. The extension Delta Modelling Language (DML) [6] 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.

```
delta AlternativePath;
modifies class ClientJobImpl {
  modifies Maybe<Int> file(String id) {
    id = "data2/" + id;
    Maybe<Int> res = original(id);
    return res;
  }
}
```

Figure 6: Delta `AlternativePath`

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.

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.

*5.1. Fundamental Approach*

*5.1.1.* **The ABSUnit Framework**

ABSUnit is an instance of the well-known XUnit test framework [11]. 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.

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

Figure 7: Typical ABSUnit test interface

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, implementing tests often requires to access or to change class internals (e.g., to check intermediate results or to shortcut complex initialization procedures). Here, Delta-Oriented Programming (see Sect. 4) provides an elegant solution: instead of cluttering the code base with auxiliary code, all test-related changes are organized into separate 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*.

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 analysing 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* [14], which represents all program execution paths up to a certain threshold, obtaining a constraint system for

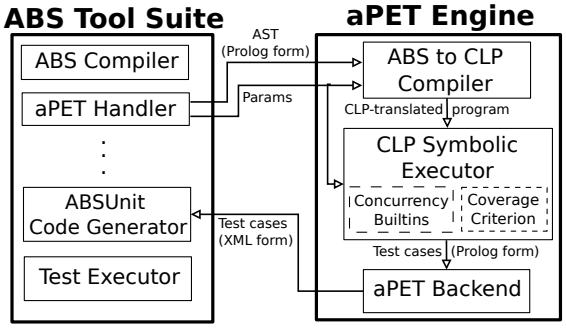


Figure 8: aPET architecture & integration

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 is an instantiation of the *Constraint Logic Programming* (CLP)-based approach to TCG [10]. CLP's backtracking-based evaluation mechanism and constraint solving facilities 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. Note that the concrete features of the considered language are abstracted in the translation and uniformly represented in CLP.

Application of this schema to concurrent ABS involved the following four steps: (i) Define an ABS to CLP compiler. (ii) Implement concurrency-related operations in CLP. The scheduling policy definition is left parametric. (iii) 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. (iv) 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*

Fig. 8 shows the basic architecture of aPET and its integration into the ABS tool suite; the latter is implemented in Java as an Eclipse plugin whereas the aPET engine is implemented in Prolog. The aPET handler is activated when the user requests to generate tests for a selected set of methods in the current ABS file. It collects a set of user-defined parameters and the *abstract syntax tree* of the ABS program and invokes the aPET engine. The latter compiles the ABS program under test into a CLP program, symbolically executes that with the given termination and coverage criterion, and generates CLP tests for each requested method. These are translated back, via XML, into ABSUnit tests, that can either be edited by the user or run by ABSUnit. As no specifications are used, aPET generates a trivial oracle from the result of running the

program that passes all tests. The oracle can be seen as a template that the user has to confirm or to modify.

*5.3. Case Study*

We consider method `file` of class `ClientJobImpl` (see Fig. 5). Setting the coverage criterion so that all feasible paths allowing one loop iteration or recursive call are expanded, 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). Note that strings are generated starting with the empty string, then generating alphabetically strings of length 1, and so on.

```
[Fixture] interface JobTest {
  [Test] Unit testFile();
}

[Suite]
class JobTestImpl implements JobTest {
  ClientJob c; DataBase b; ABSAssert aut;
  { aut = new ABSAssertImpl(); }
  Unit testFile() {
    this.setHeap();
    Maybe<Int> r = c.file("a");
    aut.assertTrue(Just(0) == r);
    this.assertHeap();
  }
  Unit setHeap() { }
  Unit assertHeap() { }
}
```

Figure 9: Generated test case

Fig. 9 shows the test method `testFile` that is generated for test case (vi) above. Its implementation first invokes `setHeap` 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 used to provide additional infrastructure for executing test cases. The first of these, `MDeltaForClientJob`, displayed in Fig. 10, 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. The second delta, `TestDelta`, depicted in Fig. 11, modifies the methods `setHeap` and `assertHeap` to set up the initial heap and check the final heap. Here `TestDelta` initializes the underlying file system to

```
delta MDeltaForClientJob;
adds interface MClientJob extends ClientJob {
  Unit setDB(DataBase b);
  DataBase getDB();
}
modifies class ClientJobImpl adds MClientJob {
  adds Unit setDB(DataBase b) { this.db = b; }
  adds DataBase getDB() { return db; }
}

delta MDeltaForDataBase;
adds interface MDataBase {
  Unit setRdir(Pair<String,Content> r);
  Pair getRdir();
}
modifies class DataBaseImpl adds MDataBase {
  adds Unit setRdir(Pair<String,Content> r) {
    this.rdir = r;
  }
  adds Pair getRdir() { return rdir; }
}
```

Figure 10: Modification Deltas

a pair of `String` value "r" and `Entries(InsertAssoc(Pair("a",Content(0)),EmptyMap))`, 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`. The delta also asserts that this value does not change after `file(id)` is executed.

## 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 [16], Tracematches [3], JmSeq [19], LARVA [8], Jass [4], and JavaMOP [5]. However, none of these address the integration into the general context of run-time assertion

```
delta TestDelta;
modifies class JobTestImpl {
 modifies Unit setHeap() {
  b = new DataBase();
  b.setRdir(Pair("r",
   Entries(InsertAssoc(Pair("a",Content(0)),EmptyMap))));
  c = new ClientJobImpl(null);
  c.setDB(b);
 }
 modifies Unit assertHeap() {
  DataBase x = c.getDB();
  Pair<String,Content> p = x.getRdir();
   aut.assertTrue(p == Pair("r",
    Entries(InsertAssoc(Pair("a",Content(0)),EmptyMap))));
 }
}
```

Figure 11: Test Delta

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 [9] we identified attribute grammars with conditional productions and annotated with assertions as powerful and user-friendly specifications of histories. 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 behavioural 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

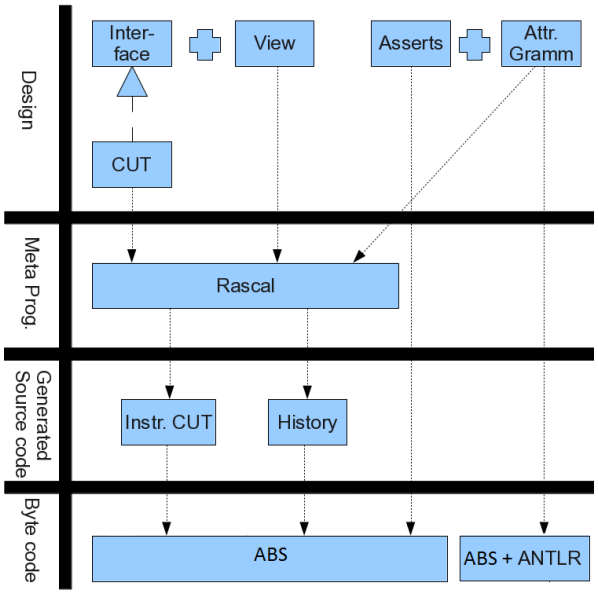


Figure 12: Tool architecture

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 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 [15], see Fig. 12. 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 [20]. 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 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.

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

Figure 13: Communication View

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

Figure 14: Attribute Grammar for the ClientJob Behaviour

### *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. 13 contains the relevant events which can be referred to in the grammar by the terminals `r`, `f`, and `c`. Fig. 14 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`). 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 $T$.

The data types used in the grammar are partially depicted in Fig. 15. 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`.

## 7. Blackbox Testing

### *7.1. Fundamental Approach*

Learning-based testing (LBT) is an emerging paradigm for *black-box requirements testing* that encompasses the three essential steps of : (1) automated test case generation

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

Figure 15: `List` data type

(ATCG), (2) test execution, and (3) test verdict (the oracle step). The first application of LBT to testing reactive systems was given in [18]. An introduction to the LBT method, which compares it with related approaches is [17].

The basic idea of LBT is to automatically generate a large number of high-quality test cases by combining a model checking algorithm with an *incremental model inference* or *active learning algorithm*. These two algorithms are integrated with the system under test (SUT) in an iterative feedback loop. On each iteration of this loop, a new test case can be generated by either of the following methods: (i) model check the most recent learned model $m_n$ of the SUT against a formal user requirement $\Phi$ and choose any counter example to correctness; (ii) use the active learning algorithm to generate a membership query; (iii) random generation. Whichever method is used, the new test case $tc_n$ 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 $(tc_n, o_n)$ is used to update the current model $m_n$ to a refined model $m_{n+1}$, which ensures that the iteration can proceed again. If the learning algorithm can be guaranteed to correctly learn in the limit, given enough information about the SUT, then LBT is a sound and complete method of testing. In practice, real-world systems are often too large for complete learning to be accomplished within a feasible time scale. By using incremental learning algorithms, that can focus on learning just that part of the SUT which is relevant to a given requirement $\Phi$, systematic testing becomes much more feasible. The overall architecture is illustrated by the diagram in Fig. 16.

### *7.2. Tool Description*

A platform for learning-based testing known as *LBTest* has been developed for black-box 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). PLTL formulas can express either *safety properties* which may not be violated, or *liveness properties*, including *use cases*, which specify intended behaviors. Note that some liveness properties cannot be refuted in any finite time (for example termination properties). For such types of properties, *LBTest* is able to issue a *warning verdict* that a test case has

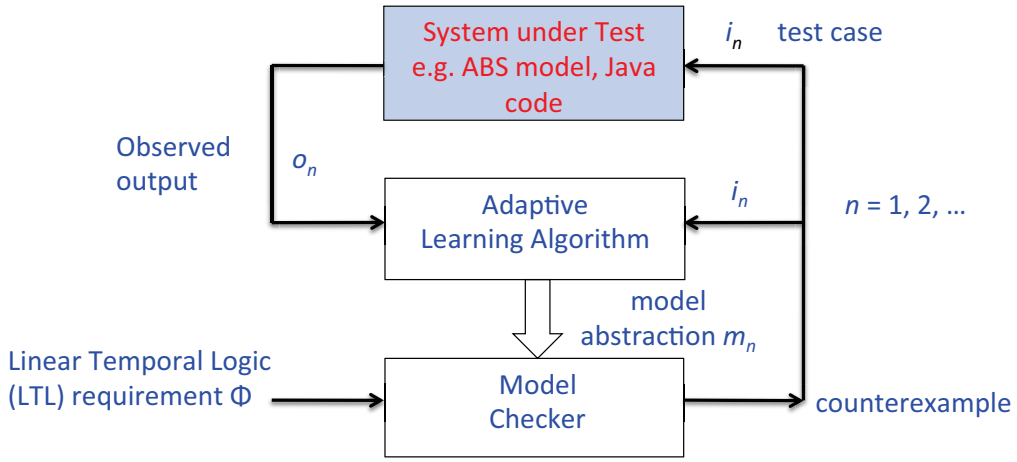


Figure 16: Architecture of learning-based testing

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. The learning algorithm currently available in *LBTest* is the IKL learning algorithm described in [18], which is an algorithm for learning deterministic Boolean-valued Kripke structures.

### *7.3. Tool Interface*

A gap exists between the low-level Boolean data type used in *LBTest* and the high-level data types supported by ABS. To bridge this gap, *LBTest* supports a simple data type declaration method that offers a flexible communication interface to an external SUT. This declaration establishes a communication protocol between *LBTest* and the SUT, in terms of data exchange. The declaration method supports user defined types, symbolic data names, and specific bit-vector data encodings. An example of the data type declaration needed to support testing of the SyncClient is given in Table 1 (for lack of space we cannot reproduce all parts of the ABS model related to the encoding, see [24] for details). Note that this data type encoding must be replicated by wrapper code around the SUT, which extracts the appropriate concrete data values (matching the symbolic values) from the bit-vector sequences which are produced by *LBTest* as test cases. Concrete output values from the SUT also need to be bit-vector encoded according the same protocol.

### *7.4. Case Study*

The *LBTest* tool was applied to the problem of black-box testing an ABS model of the Fredhopper case study described in Sect. 2. An SUT was obained by compiling the ABS model into executable JAVA code. 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:

Table 1: SyncClient data type encoding

| Data Types | Symbolic Values and Encodings | Description |
|---|---|---|
| Bits $1, \ldots, 3$ Schedules | $000 = \emptyset$,<br>$001 = \{\texttt{search}\}$,<br>$010 = \{\texttt{business}\}$,<br>$011 = \{\texttt{business}, \texttt{search}\}$,<br>$100 = \{\texttt{data}\}$,<br>$101 = \{\texttt{data}, \texttt{search}\}$,<br>$110 = \{\texttt{data}, \texttt{business}\}$,<br>$111 = \{\texttt{data}, \texttt{business}, \texttt{search}\}$ | Specifies the replication schedules to which the SyncClient should commit at any time. |
| Bits $4, \ldots, 6$ State | $000 = \texttt{Start}$,<br>$001 = \texttt{WaitToBoot}$,<br>$010 = \texttt{Boot}$,<br>$011 = \texttt{WaitToReplicate}$,<br>$100 = \texttt{WorkOnReplicate}$,<br>$101 = \texttt{End}$ | Specifies the state which the SyncClient is in as specified by the SyncClient State Machine. |
| Bits $7, \ldots, 9$ Jobtype | $000 = \texttt{nojob}$,<br>$001 = \texttt{Boot}$,<br>$010 = \texttt{SR}$,<br>$011 = \texttt{BR}$<br>$100 = \texttt{DR}$ | Specifies the type of client job scheduled by the SyncClient according to the replication schedules received. |
| Bit 10 Files | $0 = \texttt{readonly}$,<br>$1 = \texttt{writable}$ | Specifies whether the underlying file system shown be written to by the SyncClient |

$$\mathbf{G}\ (state = WorkOnReplicate \rightarrow$$
$$\mathbf{X}\ (files = writable\ \mathbf{U}\ state \in \{End, WaitToReplicate\})$$
$$\wedge\ state \neq WorkOnReplicate \rightarrow$$
$$\mathbf{X}\ (files = readonly\ \mathbf{U}\ state = WaitOnReplicate))$$

Table 2 gives the results obtained by running *LBTest* on the 11 user requirements. For each requirement (PLTL Req), we recorded the verdict (pass/fail/warning), the total time spent testing, the size of the learned hypothesis model at test termination, and the total number of model checker-generated (MC), learner-generated and random test cases executed. 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.

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. However, LBTest continued to produce warnings

Table 2: Performance of *LBTest* on the Fredhopper 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 |

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

Table 3 shows metrics for the actual implementation and the ABS model of the Replication System. Note that the ABS model includes model-level information 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, while the Java implementation leverages libraries and its API. This accounts for >1,000 lines of ABS code.

Table 3: 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 |

The quality assurance process at Fredhopper (as in many other software companies) includes automated testing. Unit tests are written manually to validate the behaviour 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 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*.

The three test approaches discussed here should be used in concertation and complement each other. E.g., 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 of concertation 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 properties while run-time assertion checking aims merely at safety properties.

## 9. Conclusion

In this paper we presented tool-supported testing techniques for ABS applied to 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 almost completely automate testing and test case generation.

As possible future work, we would like to consider lifting automated testing techniques from product to family level in product line engineering [21]. Ideas to approach this topic exist, such as sharing test cases (and test runs) between products if the products are identical with respect to the code executed (or interacted with) by the tests as described above. Work done into the direction of compositionality [2] of glassbox test generation indicate further potential to produce reusable test cases. In black box requirement testing, it becomes important to integrate product variability points into formal requirements languages such that during application engineering [21], when variability points are being instantiated for specific products, requirements may also be instantiated for those products.

## Acknowledgements

We wish to thank Andreas Kohn for proof reading this paper.

## References

[1] E. Albert, P. Arenas, and M. Gómez-Zamalloa. Towards Testing Concurrent Objects in CLP. In *Proc. of ICLP 2012*, LIPIcs, 2012. To appear.

[2] E. Albert, M. Gómez-Zamalloa, J. Rojas, and G. Puebla. Compositional CLP-based test data generation for imperative languages. In *LOPSTR 2010 Revised Selected Papers*, volume 6564 of *LNCS*. Springer-Verlag, 2011.

[3] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. 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] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - Java with Assertions. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 55(2):103 – 117, 2001.

[5] F. Chen and G. 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] D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. 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.

[7] D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte. Variability modelling in the ABS language. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957. Springer-Verlag, 2011.

[8] C. Colombo, G. J. Pace, and G. 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] S. de Gouw, J. Vinju, and F. 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.

[10] 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.

[11] P. Hamill. *Unit Test Frameworks*. O'Reilly Media, Nov. 2004.

[12] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. 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.

[13] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):35–58, Mar. 2007.

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

[15] P. Klint, T. v. d. Storm, and J. Vinju. RASCAL: 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.

[16] M. Martin, B. Livshits, and M. 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.

[17] K. Meinke, F. Niu, and M. Sindhu. Learning-Based Software Testing: A Tutorial. In R. Hähnle, J. Knoop, T. Margaria, D. Schreiner, and B. Steffen, editors, *Leveraging Applications of Formal*

*Methods, Verification, and Validation*, Communications in Computer and Information Science, pages 200–219. Springer, 2012.

[18] 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 LNCS, pages 134–151. Springer, 2011.

[19] 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.

[20] T. Parr. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, 2007.

[21] K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[22] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proc. of 14th Software Product Line Conference (SPLC 2010)*, Sept. 2010.

[23] J. Schäfer and A. 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.

[24] P. Y. H. Wong, N. Diakov, and I. Schaefer. Modelling Distributed Adaptable Object Oriented Systems using HATS Approach: A Fredhopper Case Study. In *Proc. of FoVeOOS 2011*, volume 7421 of *LNCS*, 2012.