Techniques and Toolset for Conformance Testing against UML Sequence Diagrams*

João Pascoal Faria[1,2], Ana C.R. Paiva[1], and Mário Ventura de Castro[1,2]

[1] Department of Informatics Engineering, Faculty of Engineering, University of Porto, Portugal
{jpf,apaiva,ei06064}@fe.up.pt
[2] INESC TEC, Porto, Portugal

Abstract. Novel techniques and a toolset are presented for automatically testing the conformance of software implementations against partial behavioral models constituted by a set of parameterized UML sequence diagrams (SDs), describing both external and internal interactions. Test code is automatically generated from the SDs and executed on the Java implementation under test, and test results and coverage information are presented back visually in the model. A runtime test library handles internal interaction checking, test stubs, and user interaction testing. Incremental conformance checking is achieved by first translating SDs to non-deterministic acceptance automata with parallelism.

Keywords: conformance testing, UML, sequence diagrams, automata.

1 Introduction

UML sequence diagrams (SDs) [1] allow building partial, lightweight, behavioral models of software systems, focusing on important scenarios and interactions, occurring at system boundaries or inside the system, capturing important requirements and design decisions. Such partial behavioral models may be not sufficient as input for code generation [2], but can be used as input for automatic test generation (as test specifications), using model-based testing (MBT) techniques [3]. However, existing MBT techniques from SDs have several limitations, namely in the final stages of test automation, dealing with the generation of executable tests and conformance analysis, taking into account the features of UML 2 (see Related Work section).

The overcome some of those limitations, in previous work [4], we developed a prototype tool that generates automatically JUnit [5] tests from SDs, to be executed by the user in the development environment with the support of a run-time test library. However, the test code and test results were difficult to interpret by the user and the test library had important limitations in terms of its design and functionality (namely, it lacked the support for weak sequencing). In this paper, we completely redesigned the whole approach, bringing the following contributions for enabling the automatic

* This work is part-funded by the ERDF – European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project «FCOMP - 01-0124-FEDER-022701».

H. Yenigün, C. Yilmaz, and A. Ulrich (Eds.): ICTSS 2013, LNCS 8254, pp. 180–195, 2013.
© IFIP International Federation for Information Processing 2013

conformance testing of software implementations (currently in Java) against UML SDs, in a modular and extensible way:

- novel techniques for incremental conformance checking, complying with the default weak sequencing semantics of UML SDs [1], based on the translation of SDs to non-deterministic automata with parallelism, that are executed stepwise;
- related techniques for execution tracing and manipulation, namely internal interaction tracing, test stub injection and user interaction tracing, taking advantage of aspect-oriented programming (AOP) techniques and reflection;
- related techniques for test code generation from the model and test results visualization in the model (conformance errors and coverage information), raising the level of abstraction of the user feedback and improving usability.

The rest of the paper is organized as follows: section 2 presents an overview of the approach; section 3 describes the characteristics of test-ready SDs; sections 4, 5 and 6 present the main contributions; section 7 presents a case study; section 8 presents a comparison with related work; section 9 concludes the paper.

2 Approach and Toolset Overview

Our toolset, named UML Checker, comprises two independent tools (see Fig. 1): an add-in for the Enterprise Architect (EA) modeling tool [6], chosen for its accessibility and functionality; and a reusable test library, implemented in Java and AspectJ [7]. The add-in gets the needed information from the model via the EA API and generates JUnit test driver code, including traceability links to the UML model (message identifiers) and expectations about internal interactions. The test code is then compiled and executed over the application under test (AUT). The behavior of the AUT in response to the test inputs (namely internal messages) is traced by the test library using AOP, and compared against the expected behavior. All discrepancies and exceptions occurred and messages effectively executed are listed in the execution result that is processed by the EA add-in, which annotates the model accordingly.

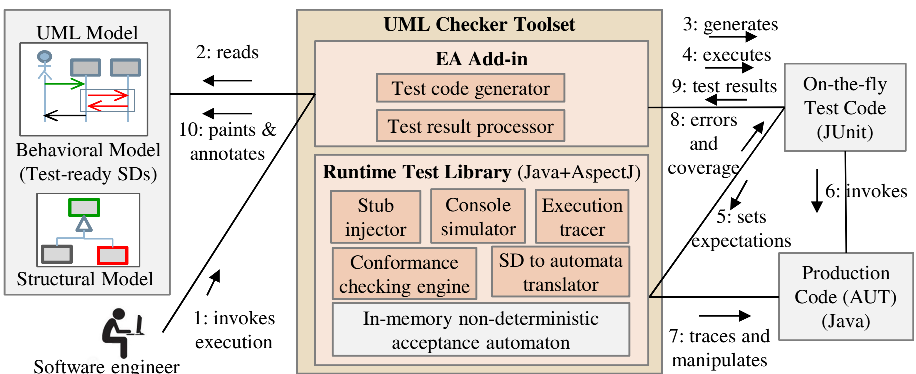


Fig. 1. Communication diagram illustrating the toolset architecture and functioning

3 Test-Ready Sequence Diagrams

This section describes the characteristics that SDs should have to be used as test specifications for automated conformance testing in our approach.

The usual modeling features of SDs [1] are supported, with some restrictions and extensions. As illustrated in Fig. 2, the following types of interactions can be modeled and automatically tested in our approach:

- external interactions with client applications through an API;
- external interactions with users through a user interface (UI);
- internal interactions among objects in the system;
- interactions with objects not yet implemented (marked as «`stub`»).

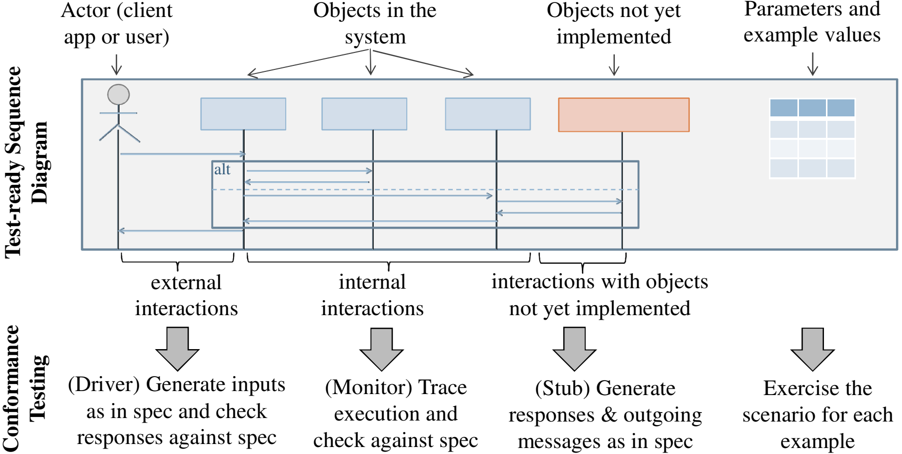


Fig. 2. Major constituents of test-ready sequence diagrams and usage for conformance testing

For example, the SD in Fig. 3 includes external interactions with a client application (messages `Account` and `withdraw`), as well as some internal interactions (messages `setBalance` and `Movement`).

Next we describe in more detail the major constituents of test-ready SDs and how they are treated in conformance testing automation.

Interaction Parameters. Parameterization of SDs allows defining more generic scenarios in a rigorous way. A set of parameters, with their names and types, may be defined in each SD, accompanied by example values. E.g., the note marked «`Parameters`» in Fig. 3 defines two parameters and two combinations of parameter values. Parameters have the scope of the SD and can be used anywhere (including as lifelines). For test execution, each parameterized SD is treated as a parameterized test scenario and each combination of parameter values as a test case. If no parameters are defined (i.e., values are hardcoded in the messages), the SD defines a single test case.

Actors. Test-ready SDs should have a single actor, representing a user or a client application that interacts with the AUT through a user interface or API, respectively.

During test execution, the actor is treated as a test driver, responsible to send the specified outgoing messages to the AUT, taking into account any guard conditions defined, and to check the responses against the expected values specified in the diagram.

User Interaction Testing. Since the UML does not prescribe a standard way for that purpose, we adopted a set of keywords (signals) to model user interaction through the console in an abstract way (possibly since the requirements phase):

- `start(args)` – the user starts the application (indicated by its main class);
- `enter(v)` – the user enters the value specified through the standard input;
- `display(v)` – the application displays the value specified to the standard output.

During test execution, the test harness injects the values specified by `enter` messages, simulating a user, and compares the actual AUT responses against the expectations specified by `display` messages.

Internal Interactions Checking. Besides external interactions with client applications or users, test-ready SDs may also describe interactions among objects in the AUT, capturing significant design decisions. During test execution, for each message sent to the AUT, the test harness also checks that internal messages among objects in the AUT occur as specified and internal objects are created and passed as specified. The benefits are improved conformance checking and fault localization.

In order to allow keeping SDs as minimalist as wanted, focusing only on relevant interactions, and enable the scalability of the approach, we support by default a **loose conformance** mode, in which additional messages are allowed in the AUT, besides the ones specified in the diagram (differently from what happens with the other supported conformance mode - **strict** conformance).

Stubs in the Middle. Lifelines may be marked as «`stub`», to indicate that the corresponding classes (possibly external to the AUT) are not yet implemented or one does not want to use the existing implementation. During test execution, the test harness generates not only the reply messages, but also the outgoing messages (hence "stub in the middle") specified in the SD for any incoming messages. This allows testing partial implementations and simulating additional actors.

Interaction Operators. The most commonly used combined fragments are supported, allowing the specification of more generic scenarios with control flow variants (with the `alt`, `opt`, `loop`, `par`, `seq` and `strict` interaction operators). Conditions of `alt` and `opt` operators may be omitted, to model situations in which the implementation has the freedom to choose the path to follow and to support partial specifications (see, e.g., the inner `alt` fragment in Fig. 3).

Value Specifications. Message parameters, return values and guards may be specified by any computable expression in the context of the interaction (involving constants, interaction parameters, lifelines, etc.), as long as it has no side-effects on participating objects. Otherwise, the evaluation of expected parameter and return values or guards during test execution could change the behavior of the AUT. Looseness in the specification of parameter and return values can be indicated by means of the "-" symbol (matching any value), and by omitting the return value, respectively. During test execution, the semantics of value checking depends on the implementation of `equals` and the comparison precision defined for some data types in the conformance settings.

4 Test Code Generation and Test Results Visualization

This section describes the test code generation and results' visualization techniques. The techniques are illustrated with the running example of Fig. 3, referring to a simple application that exposes an API for creating bank accounts (with an initial balance) and withdrawing money (with alternative execution paths, depending on the money available and the way chosen by the implementation to record movements).

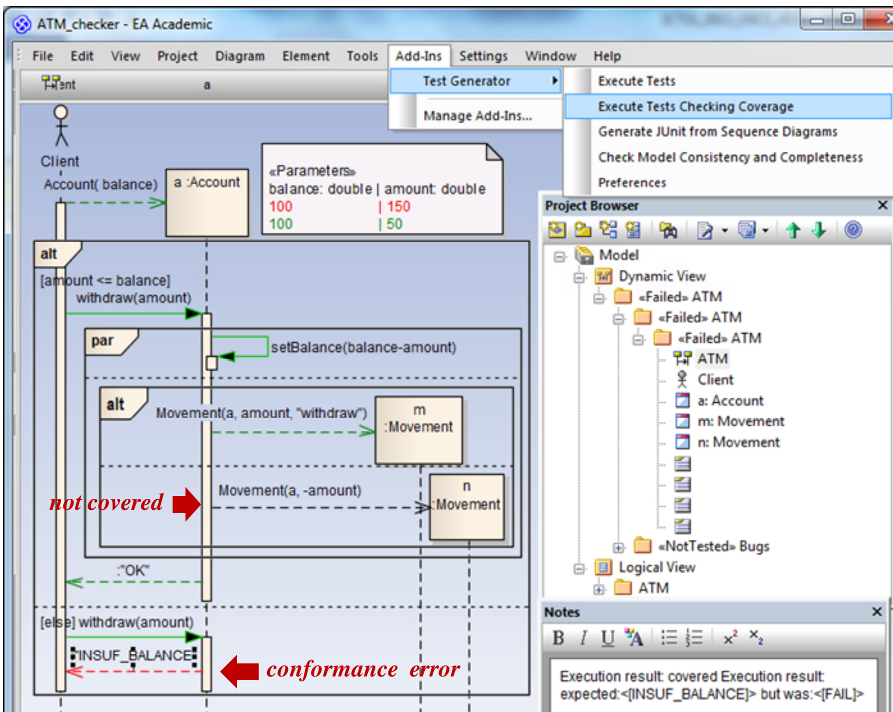


Fig. 3. Sequence diagram of the running example, painted and annotated after test execution

Test Code Generation. A test class is generated from each SD, with the general self-explanatory structure illustrated in Fig. 4, containing a parameterized test method corresponding to the SD and a plain test method for each combination of parameter values. `InteractionTestCase` is a facade [8] that extends JUnit3 `TestCase`. To assure that expressions of message arguments, return values and guards (possibly dependent on the execution state) are evaluated at proper moments, they are encoded with `ValueSpec`. To allow the incremental binding of lifeline names to actual objects (see sec.5), they are encoded with `Lifeline` - a proxy [8] for the actual object.

Test Results Visualization. The results of test execution are presented visually in the model, using a combination of graphical and textual information, as illustrated in Fig. 3. The following color scheme is used for painting each combination of parameter values and each message: black - not exercised, green - exercised without errors, red - exercised with errors. For each message exercised with errors, the error

information (plus the AUT stack trace if wanted) is shown in the message notes. Possible error types and locations are shown in Table 1. The information about messages not covered (exercised) is important in the presence of conditional paths, to check the adequacy of test data (parameter values), and in the presence of unconstrained 'opt' or 'alt' fragments, to analyze implementation choices.

Table 1. List of conformance errors and locations in the model where they are signaled

Conformance error	Location in the model
Wrong argument	Call message.
Wrong return value	Reply message, if it exists; call message, otherwise.
Unexpected exception	Method or constructor execution bar.
Unexpected call (strict conformance)	Method or constructor execution bar.
Missing call	Call message or mandatory combined fragment.
Missing or incorrect output	`display` message
Missing input	`enter` message

The behavioral model packages are marked with self-explanatory stereotypes, depending on the status of contained SDs: «Failed», «Passed», «NotTested» and «Incomplete». The stereotypes are visible in the project browser for a quick check of conformance status. The classes and methods in the structural model (class diagrams) that are not covered (exercised) by the behavioral model are also marked as «NotCovered», to help assessing the completeness of the behavioral model.

```
public class ATMTest extends InteractionTestCase {
  private Account a = null; // similar for lifelines m,n
  public void testATM(final double balance,final double amount){
    ValueSpec exp0 = new ValueSpec() {
       public Object get() {return balance-amount;}
    }; // similar for other expressions occurring in SD
    Lifeline aLifeline = new Lifeline() {
       public void set(Object value) {a = (Account)value; }
       public Object get() {return a;}
    }; // similar for other lifelines occurring in SD
    // Declares expected interactions to conform. check. engine:
    expect(/*encoding of SD fragments and messages here*/);
    // Traditional JUnit test driver code (actor messages):
    a = new Account(balance);
    if (amount <= balance)
       assertEquals("OK",a.withdraw(amount));
    else
       assertEquals("INSUF_BALANCE", a.withdraw(amount));
    // Final check of interactions missing:
    finalCheck();
  }
  public void testATM_0() { testATM(100, 150); }
  public void testATM_1() { testATM(100, 50); }
}
```

Fig. 4. Skeleton of test code generated from the SD in Fig. 3

5 Techniques for Incremental Conformance Checking

Translation to Automata. To handle uniformly the variety of interaction operators allowed in SDs, and comply with the default weak sequencing semantics of UML SDs [1] with implicit parallelism between lifelines, SDs are first translated to non-deterministic automata according to the following steps (also illustrated in Fig. 5):

1. **Generate states.** Possible states are generated in each lifeline before and after each message end (including implicit reply messages from synchronous calls), combined fragment boundary and operand boundary. Additionally, a (global) start state and a (global) final state are introduced for the whole diagram. An auxiliary state is also generated for each asynchronous message (see Table 2-j).
2. **Generate transitions.** Transitions linking lifeline states, possibly with multiple source and/or target states (as in parallel finite automata [9]), are generated according to the rules shown in Table 2. A transition is generated for each synchronous message, synchronizing the lifelines involved. Regarding combined fragments, automatic transitions (without events) are generated to enter and exit the combined fragment and its operands along the lifelines covered. Following a common semantic choice [10], the lifelines involved are synchronized in the decision points of 'alt', 'opt' and 'loop'. Otherwise, it is followed the default weak sequencing semantics of SDs (except obviously for 'strict'). Additionally, it is generated a transition linking the start state of the SD to the first state in all lifelines, and another linking the last state in all lifelines to the final state of the SD.
3. **Simplify** (optional). The resulting automaton is simplified by removing transitions with empty labels and redundant states, resulting in an equivalent automaton that accepts the same traces. Another example partially simplified is shown in Fig. 6.

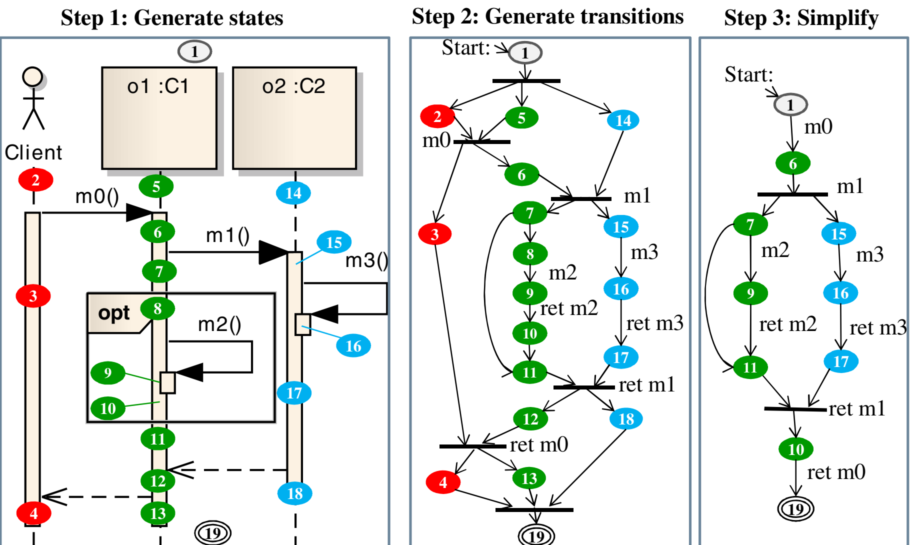
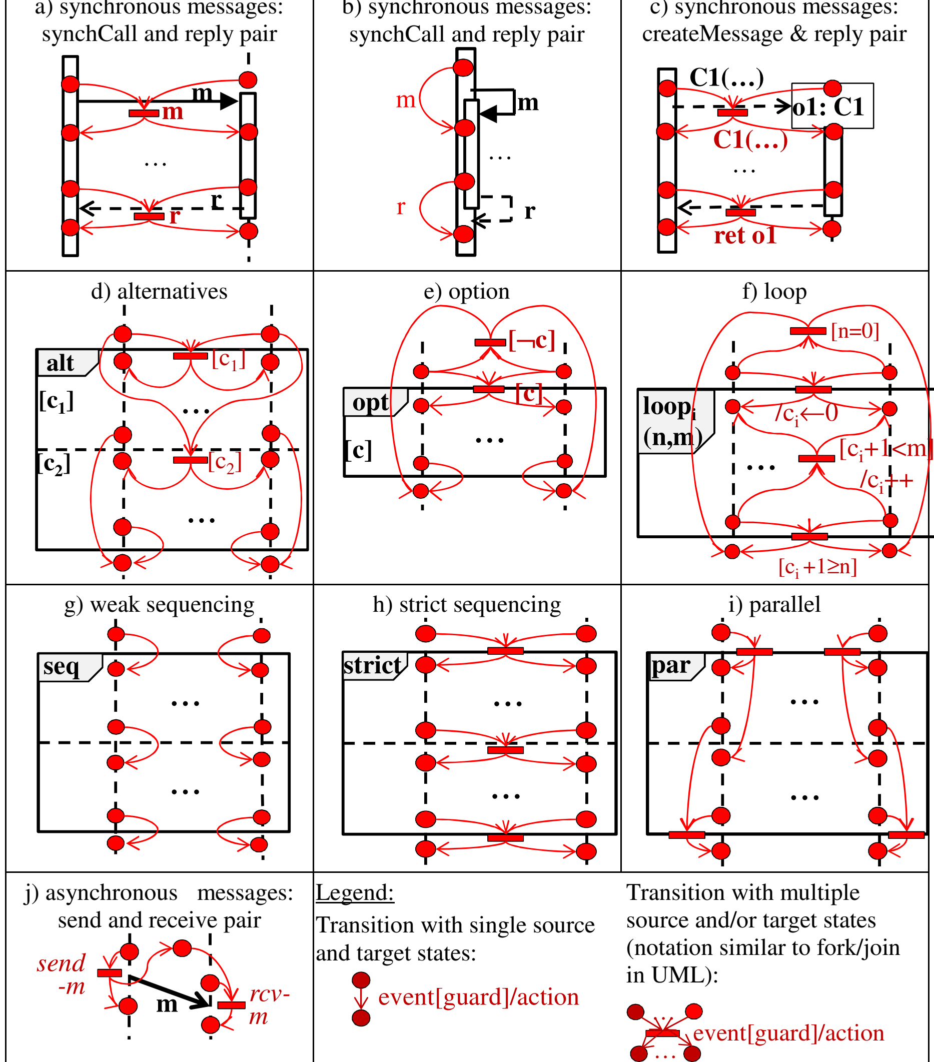


Fig. 5. Three-step translation process (with the 3rd optional) from SD to acceptance automaton

Table 2. Transition generation rules (superimposed in red) for different fragments



a, b, c) Even if not indicated, reply messages are always assumed after synchronous calls.
d, e) In the absence of guards in the SD, all guards are also omitted in the generated transitions.
f) c_i is a counter variable for loop i. Counters are not needed if n≤1 and m='*'.
d, e, f, g, h, i) These rules extend trivially to more than two operands and/or lifelines.
i) A coregion can also be treated as a parallel combined fragment over a single lifeline, having as operands the message ends enclosed in the coregion.
j) Currently implemented only for the translation of user interaction messages modeled with the `start`, `enter` and `display` signals. An auxiliary state is introduced for ordering the message sending and receiving events. Even inside loops, given our choice for synchronization at decision points, at most one sent message occurrence may be waiting to be received.

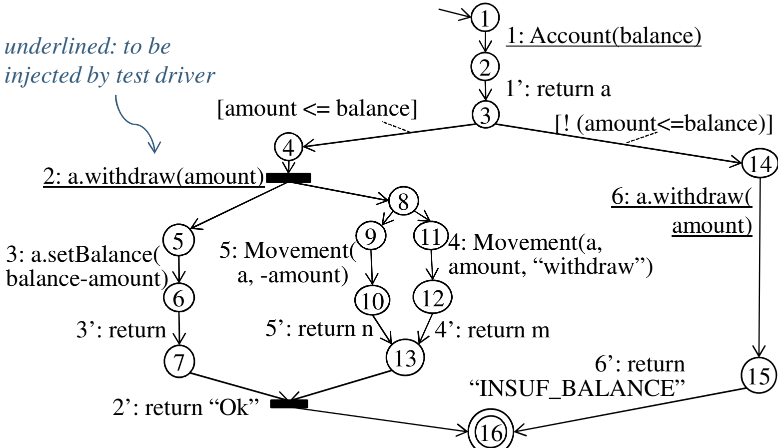


Fig. 6. Automaton generated from the example SD of Fig. 3 (only partially simplified)

Automata Structure. In our approach, a conformance checking automaton is a tuple $\prec S, s_o, F, T\succ$, where S is the set of states, $s_o\in S$ is the initial (start) state, $F\subseteq S$ is the set of final (accepting) states, and T is the set of transitions. Each transition is a tuple $\prec\sigma, \lambda, \tau\succ$, where $\sigma\subseteq S$ is the set of source states of the transition, $\tau\subseteq S$ is the set of target states of the transition, and λ is the transition label. Transitions with multiple source and/or target states are used to handle parallelism and synchronization (see Table 2). A transition label is a triple *event[guard]/action,* all of which components are optional. Transitions without event are automatic. The automaton may be non-deterministic, i.e., different transitions $<\sigma_i, \lambda_i, \tau_i>$ and $<\sigma_j, \lambda_j, \tau_j>$ may exist with $\sigma_i=\sigma_j \wedge \lambda_i=\lambda_j \wedge \tau_i\neq\tau_j$, or with $\lambda_i\neq\lambda_j$ but simultaneously satisfiable (e.g., call specifications *m(1)* and *m(-)* are both satisfiable by the occurrence *m(1)*).

Automata Execution. An automaton run state is a tuple $\prec A, \beta, \rho, C\succ$, where:

- A is the set of active automaton states (multiple active states may exist because of parallelism), starting with $\{s_0\}$; each time a transition $\prec\sigma, \lambda, \tau\succ$ is performed, requiring $\sigma\subseteq A$, the new set of active states becomes $(A\backslash\sigma)\cup\tau$;
- $\beta=\beta_P\cup\beta_L\cup\beta_C$ is a binding of variable names to actual values, starting with the binding β_P of interaction parameters to actual values, and incrementally extended with the binding β_L of lifeline names to actual objects and the binding β_C of loop counters to actual values; β_L is extended as message occurrences are encountered involving lifeline names as target, argument or return value; subsequent occurrences of a previously bound lifeline name must refer to the same object;
- ρ is a mapping from identifiers of matched call or send event occurrences to identifiers of corresponding events in the automaton; this is needed to assure that reply or receive occurrences corresponding to ignored call or send occurrences (in loose conformance mode only) are also ignored, and that reply or receive occurrences corresponding to considered call or send occurrences are matched against the

correct event in the automaton; we assume that all call-reply and send-receive pairs have related identifiers, like n and n' (see Fig. 6);

- C is the set of identifiers of messages covered so far, starting with the empty set, for coverage analysis purposes.

Because the same event occurrence may match multiple event specifications (non-determinism), it is kept a set of possible run states $R=\{r_1, ..., r_n\}$. Conformance checking fails when R becomes empty at any point of execution, or, at the end of execution, there is no run state $r_i \in R$ such that all its active states A_i are accepting states (i.e., $\neg\exists r_i \in R \bullet A_i \subseteq F$). An example execution is illustrated in Fig. 7.

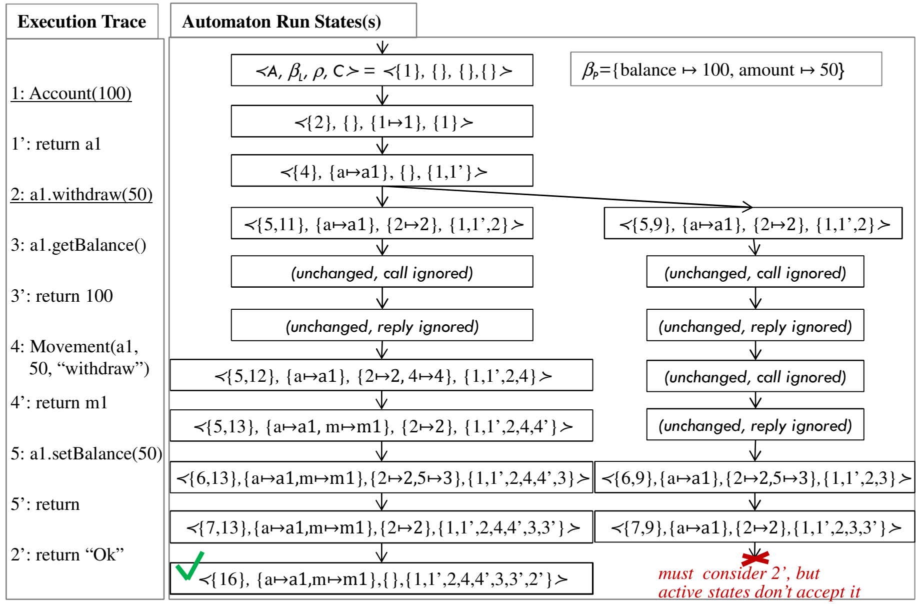


Fig. 7. Example of successful execution of the automaton of Fig. 6 for a possible test case and AUT response in loose conformance mode. The test driver stimuli are underlined in the trace.

6 Techniques for Execution Tracing and Manipulation

In this section we present techniques, based on AOP with load time weaving, to enable execution tracing, stub injection, and user interaction testing in a minimally intrusive way.

Execution Tracing and Stub Injection. Method and constructor invocation and execution in the AUT are intercepted with the AspectJ [7] code depicted in Fig. 8. Method invocations are traced with an `execution` pointcut (line 5), when the control focus is already on the target object, because it also captures reflective invocations. In the case of constructors, operations invoked by super-constructors execute

before the self-constructor (and not nested), so we use `call` pointcuts instead (when the control focus is still on the sender object) for proper nesting, with two versions for normal and reflective calls (lines 19 and 20). The invocation and reply occurrences intercepted by the aspect code are sent to the conformance checking engine for incremental checking (lines 6 and 14). Regarding stubs, we assume that objects marked as «`stub`» in the SD have compilable method skeletons; instead of executing the actual method body, the outgoing messages specified in the SD (constructor and method calls) and enabled in the automaton are issued through reflection (line 12), and it is returned the value specified in the SD and enabled in the automaton (line 13).

```
1: public priviliged aspect TracingAspect {
2:  // Aux. def. to filter points of interest and avoid infinite recursion:
3:  pointcut mayTrace():  /* definition omitted */ ;
4:  // Intercepts normal and reflective method invocations:
5:  Object around(): mayTrace() && execution(* *(..)) {
6:   Process invocation (call) occurr. by automaton (via synchronized method)
7:     If the automaton failed, throw the failure
8:     If no match was found, proceed with normal execution and return
9:   If target object is not marked as «stub» or this is a constructor call,
10:    Proceed with normal execution and get return value
11:  If target object is marked as «stub», perform stub injection, i.e.,
12:    Execute outgoing calls spec./enabled in SD/automaton via reflection
13:    If this isn't a constructor call, get return value from SD/automaton
14:  Process reply (return) occurrence by automaton (via synchronized method)
15:    If the automaton failed, throw the failure
16:  Return the return value
17: }
18: // Intercepts normal and reflective constructor invocations:
19: Object around(): mayTrace() && call(new(..) {similar template}
20: Object around():mayTrace()&&call(Object Constructor.newInstance(..)){idem}
21:}
```

Fig. 8. Skeleton of AspectJ code responsible for execution tracing and stub injection

User Interaction Testing. The mechanisms for user interaction testing of console applications are illustrated in Fig. 9. A console simulator (from our test library) starts the AUT in a thread separate from the test driver and creates input and output blocking queues for communication and synchronization between both. AUT calls to read and write operations on `System.in` and `System.out` are intercepted with `around` pointcuts, and replaced by `poll` and `put` operations on the input and output queues, respectively. User interaction messages specified in the SD with the `enter` and `display` keywords originate `put` and `poll` operations that are performed by the test driver on the input and output queues. `Poll` operations are subject to a timeout. Although the test driver already checks displayed values (with `assertEquals`), the relevant events are also sent to the conformance checking automaton for checking their proper ordering with respect to other execution occurrences.

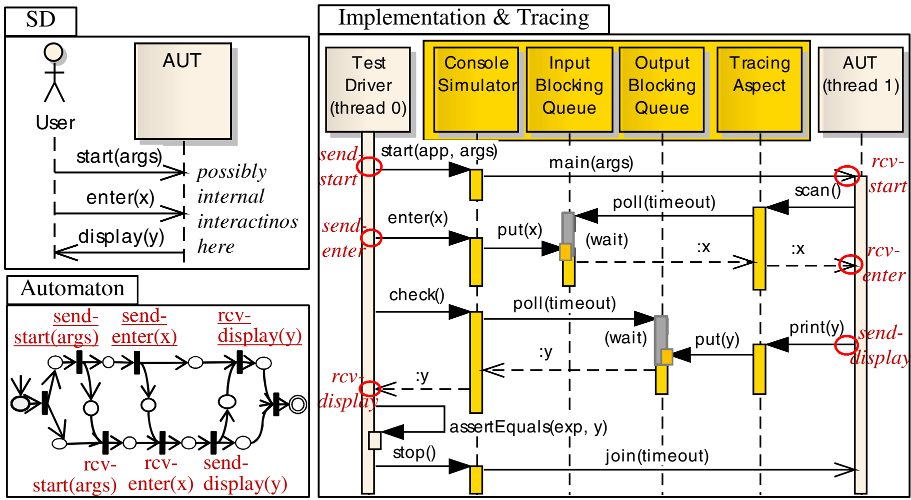


Fig. 9. User interaction specification and conformance testing mechanism for console apps

7 Case Study

To assess our approach we conducted a case study on a Java application, developed at our university and used since 2009 by approximately 200 software engineering students for program size measurement. The application, named “FileDiff”, computes the difference with minimum cost between two source files, in terms of lines added (cost 1), modified (cost 1) or deleted (cost 0), ignoring blank lines and comments. An accompanying UML model contains a SD that exercises all classes and methods. Some input files for manual testing purposes also accompany the application.

The goal of the case study was to confirm the main benefits of our approach: the ability to take advantage of existing behavioral models for test automation (hence reducing the test effort); the ability to find discrepancies between the model and the implementation (hence enabling improvements in their quality).

The initial SD ([11], pg.67, not shown here for space limitation reasons) was not test-ready, due to the usage of pseudo-code and a "procedural" feature - attribute assignment ([1], pg.507) - not currently supported by our tool. These problems were solved by making minor changes to the SD. Test parameters and test data were added to exercise the SD for existing test files, resulting in a test-ready SD with 34 messages, 8 lifelines (4 of which instantiated dynamically), and 2 combined fragments ([11], pg.72). Conformance test execution revealed a message not covered (caused by an incorrect loop modeling) and a message exercised with errors (caused by an incorrect sequencing in the implementation), which were fixed in the model and in the implementation, respectively.

Hence, the benefits of our approach could be demonstrated for this case study. Other case studies and acceptance tests performed to validate the approach for all supported modeling features, error types and coverage levels can be found in [11].

8 Related Work

There are several research works that attempt to use UML SDs either to help understand the systems or for quality purposes, like model checking and model-based testing. However, the use of SDs for testing or other rigorous verification methods demands for a rigorous definition of the language semantics [10].

Based on a survey of proposed semantics for UML SDs, Micskei *et al.* [10] point out several problems or challenges with the current natural language semantics, and categorize the choices taken by 13 selected approaches to address them. The semantic choices taken in our approach are inspired by their work and can be classified as follows: execution traces are either valid or invalid, i.e., no inconclusive traces exist; the underlying formalism is based on the encoding of the partial orders into a finite structure (automaton), for efficient processing, with interleaving as the concurrency model (as in the UML standard); both complete (in strict conformance mode) and partial trace specifications (in loose mode) are supported; fragments are combined using standard interpretation with weak sequencing; choices and guards are handled globally, i.e., the involved lifelines synchronize at decision points for evaluating guards and/or choosing a path to follow; the SD is processed by analyzing it as a whole using locations (lifeline states in our case), for higher flexibility.

Currently, we do not support the 'assert', 'neg', 'ignore' and 'consider' operators, but the approach can be easily extended, in particular, 'ignore' and 'consider' operators can be dealt by allowing an explicit conformance mode, besides the loose and strict modes, and 'assert' and 'neg' operators will lead to irrevocable failure states.

As pointed out in [10], the first step of many proposed semantics (e.g., [12]), is to find all the legal cuts of a diagram (global SD states, i.e., combinations of lifeline locations), but finding cuts can get complicated in the presence of complex fragments and asynchronous communication. One advantage of our approach is that, by allowing transitions with multiple source and/or target states, we avoid determining those cuts, as well as the potential explosion of states and transitions. A class of automata with that type of transitions, named parallel finite automata (PFA), was first proposed in [9] as a convenient way to express the interleaving parallelism inherent in Petri net notation without admitting the possibility of an infinite state space (in other words, without admitting multiple tokens per place); the authors also show the equivalence and translation procedure of PFA to deterministic finite automata (DFA). Our conformance checking automata are inspired by the concept and properties of PFA, with the addition of several features found in extended state machines (such as UML state machines [1]), namely state variables and event-guard-action transition labels.

Regarding the representation of execution traces, we follow the approach of [13], which uses a single event to model synchronous (instantaneous) communication and a pair of send and receive events to model asynchronous (non-instantaneous) communication. In [13], it is also proposed a translation procedure of a partially ordered execution trace, containing both synchronous and asynchronous communications (but not interaction operators), into a system of communicating automata, with one automaton per process (lifeline) and one 'message delay' automaton per asynchronous communication, which product yields the possible ways of interleaving events. Despite the different outputs, our translation procedure follows some of the principles of their approach, adding the support for interaction operators and other UML SDs' features.

Despite the challenges with the SD semantics, there are different approaches in the literature that use SD for quality purposes. For instance, [14] translate UML 2 interactions into automata for model checking with respect to specified requirements.

There are also approaches that extract SDs from a dynamic analysis of the system for comparison with design SDs. These approaches are split into four phases: instrumentation; logging; merging (in the case of distributed systems); and comparison. The work of [15] uses AOP to support the instrumentation of Java systems' bytecode. To deal with the fact that SDs are not a straightforward representation of the extracted traces, they define two metamodels (one for traces and another for SDs) and define mapping rules between them using OCL. In our approach, AOP is also used, not only for execution monitoring, but also for user interaction redirection and stub injection.

Other approaches use SDs in the context of model-based testing, either by focusing in test case generation, data generation and/or code generation.

Since SDs show objects and messages exchanged among them along time, test cases generated from them may be adequate to find errors concerning the sequence of executed messages and the values passed [16]. The interaction operators introduced in UML 2 allow the description of a number of traces in a compact and concise manner. Because of that, there are several examples in the literature that use an intermediate notation to represent the set of possible executions within a SD and afterwards, test cases are generated from this representation according to coverage criteria. Some examples of such representations are "sequence dependency graphs" [17], "message dependency graphs" [18], and "structured composite graphs" [19].

Besides generating test sequences, there are some approaches that also generate test data. Nayak *et al.* [19] enrich SDs with attribute and constraint information derived from class diagrams and OCL constraints and use a constraint solver to generate test data to cover paths along scenarios. Samuel *et al.* [18] create dynamic slices according to conditional predicates associated with messages in a SD and generate test data satisfying each slice. Benauttou *et al.* [20] generate test data based on partition analysis of method contracts expressed in the Disjunctive Normal Form.

Another important feature is the generation of test code at the end. There are approaches that generate assertions to check consistency of models with manually derived code at run time [21] and others that generate test code, for instance, as unit tests. These approaches can be used in combination. Some of the latter examples are: a tool generating test code from SDs (SeDiTeC tool [22]); a tool generating functional test drivers from SDs (SCENTOR [23]); a Model-Driven Architecture based approach for generating test code for multiple unit testing frameworks [24].

Javed *et al.* [24] apply a model-to-model transformation from SDs into a xUnit model independent form a particular unit testing framework and, afterwards, apply a model-to-text transformation into JUnit or SUnit. However, their approach has several limitations: the checking of returned values is performed in an intrusive way by constructing additional objects, which is problematic when constructors have side-effects; the gathering of execution traces is not integrated into the approach and they do not automate their verification; they do not deal with the novel features of UML 2.

One advantage of SeDiTeC [22] is the generation of stubs for parts of the AUT not implemented, hence allowing starting testing earlier. As far as we know, they do not deal directly with the novel features of UML 2. However, they combine different SDs which can be used as a way to represent, for instance, alternative blocks of messages.

SCENTOR [23] tool creates functional test drivers for e-business applications from SDs that have test data (parameters and expected values of method calls) embedded in them. However it does not check internal interactions and does not generate test stubs.

Some commercial tools also support conformance testing based on SDs. To our knowledge, the IBM Rational Rhapsody TestConductor Add On [25] is one of the more advanced tools. Having as target real-time embedded applications, it supports many features in common with our approach (like internal interaction checking, visual feedback, etc.) and other features outside the scope of our approach. Despite its powerful features, it does not support several important features of our approach: incremental lifeline instantiation with create messages (all objects must be previously defined in a test architecture); non-deterministic 'alt', 'opt' and 'loop' operators (without guards); strict conformance mode (message types absent from a SD are not traced); stubs in the middle (only normal stubs are supported); user interaction testing.

The implementation of tests derived from SDs or similar formalisms in distributed asynchronous environments poses additional challenges for coordinating test drivers, monitors and stubs. An example of an approach for monitoring the execution of distributed Java applications with AOP was presented in [15]. An approach for coordinating distributed test components (namely test drivers) was presented in [26].

9 Conclusions and Future Work

It were presented a set of techniques and a toolset for the automatic conformance testing of software applications against behavioral models constituted by a set of parameterized UML 2 SDs. With a single click, test cases are automatically generated from the model, executed on the AUT and test results and coverage information presented back visually in the model. The conformance checking approach, based on the translation of SDs to nondeterministic acceptance automata with parallelism that are executed stepwise, provides several advantages over existing SD-based testing techniques, namely regarding the kinds of interactions, operators, conformance modes, and semantics (weak sequencing) supported. The tool was successfully experimented on a set of case studies, one of which was presented. Despite being implemented for specific technologies, the overall approach can be applied for other technologies.

As future work, we plan to: support other modeling environments (reusing the runtime library); support additional modeling features (such as duration constraints and the 'neg', 'ignore', and 'consider' operators); support a semantic option without lifeline synchronization at decision points; extend the abstract user interaction modeling and testing features for GUIs (which, currently, can be handled in a non-abstract way); integrate with approaches for the automatic generation of values for scenario parameters; extend the test execution engine to support the testing of distributed systems; conduct further experiments to assess our approach compared to others.

References

1. OMG Unified Modeling Language™ (OMG UML), Superstructure, v. 2.4.1, OMG (2011)
2. Mellor, S.J., Clark, A.N., Futagami, T.: Model-Driven Development. IEEE Software Magazine 20(5), 14–18 (2003)

3. Uttin, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann (2007)
4. Faria, J.P., Paiva, A., Yang, Z.: Test Generation from UML Sequence Diagrams. In: 8th Int. Conf. on the Quality of Information and Communications Technology, pp. 245–250 (2012)
5. JUnit testing framework, `http://www.junit.org`
6. Enterprise Architect, `http://www.sparxsystems.com.au`
7. AspectJ, `http://www.eclipse.org/aspectj`
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Education (1994)
9. Stotts, P.D., Pugh, W.: Parallel Finite Automata for Modeling Concurrent Software Systems. J. of Software and Systems 27, 27–43 (1994)
10. Micskei, Z., Waeselynck, H.: The Many Meanings of UML 2 Sequence Diagrams: a Survey. J. of Software and Systems Modeling 10, 489–514 (2011)
11. Castro, M.V.: Automating Scenario Based Testing with UML and AOP, `http://www.fe.up.pt/~ei06064/AutomatingSBTwithUMLandAOP.pdf` (in Portuguese)
12. Harel, D., Maoz, S.: Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. J. of Software and Systems Modeling 7(2), 237–253 (2008)
13. Hallal, H., Boroday, S., Petrenko, A., Ulrich, A.: A Formal Approach to Property Testing in Causally Consistent Distributed Traces. Formal Aspects of Computing 18(1), 63–83 (2006)
14. Knapp, A., Wuttke, J.: Model Checking of UML 2.0 Interactions. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 42–51. Springer, Heidelberg (2007)
15. Briand, L., Labiche, Y., Leduc, J.: Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. IEEE Trans. on Soft. Eng. 32(9), 642–663 (2006)
16. Kansomkeat, S., Offutt, J., Abdurazik, A., Baldini, A.: A Comparative Evaluation of Tests Generated from Different UML Diagrams. In: SNPD 2008, pp. 867–872 (2008)
17. Philip, S., Joseph, A.T.: Test Sequence Generation from UML Sequence Diagrams. In: SNPD 2008, pp. 879–887 (2008)
18. Samuel, P., Mall, R.: A Novel Test Case Design Technique using Dynamic Slicing of UML Sequence Diagrams. e-Informatica 2(1), 71–92 (2008)
19. Nayak, A., Samanta, D.: Automatic Test Data Synthesis using UML Sequence Diagrams. J. of Object Technology 9(2), 115–144 (2010)
20. Benattou, M., Bruel, J., Hameurlain, N.: Generating Test Data from OCL Specification. In: ECOOP Workshop Integration and Transformation of UML Models (2002)
21. Engels, G., Güldali, B., Lohmann, M.: Towards Model-Driven Unit Testing. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 182–192. Springer, Heidelberg (2007)
22. Fraikin, F., Leonhardt, T.: SeDiTeC-testing based on sequence diagrams. In: Proc. of the 17th IEEE Int. Conf. on Automated Software Engineering (ASE 2002). IEEE (2002)
23. Wittevrongel, J., Maurer, F.: SCENTOR: Scenario-Based Testing of E-Business Applications. In: 2nd Int. Workshop on Automation of Software Test (AST) (2007)
24. Javed, A., Strooper, P., Watson, G.: Automated Generation of Test Cases using Model-Driven Architecture. In: 2nd Int. Workshop on Automation of Software Test (AST) (2007)
25. IBM® Rational® Rhapsody® Automatic Test Conductor Add On User Guide, v2.5.2 (2013)
26. Boroday, S., Petrenko, A., Ulrich, A.: Implementing MSC Tests with Quiescence Observation. In: Núñez, M., Baker, P., Merayo, M.G. (eds.) TESTCOM 2009. LNCS, vol. 5826, pp. 49–65. Springer, Heidelberg (2009)