Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-014-0354-x

ICTSS 2013

A toolset for conformance testing against UML sequence diagrams based on event-driven colored Petri nets

João Pascoal Faria · Ana C. R. Paiva

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, describing both external interactions with users or client applications and internal interactions between objects in the system. Test code is automatically generated from the sequence diagrams and executed on the 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, taking advantage of aspect-oriented programming techniques. Incremental conformance checking is achieved by first translating sequence diagrams to Extended Petri Nets that combine the characteristics of Colored Petri Nets and Event-Driven Petri Nets.

Keywords Sequence diagrams · Conformance testing · Petri nets

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 Sect. 8).

We have been working for some time on the development of techniques and tools for enabling the automatic conformance testing of software implementations against UML SDs, overcoming some of those limitations. In [4], we presented a tool to automatically generate JUnit [5] tests from SDs, for execution by the user in the development environment with the support of a run-time test library. However, it lacked the support for important UML 2 features, namely weak sequencing and a number of interaction operators. In [6], we introduced the ability to invoke test execution and obtain test results directly in the modeling environment, and presented a new conformance checking approach, based on the translation of SDs into extended Parallel Finite Automata [7], in order to comply with the default weak sequencing semantics of UML SDs [1], with implicit parallelism between lifelines. However, lifelines were forced to synchronize at decision points of `alt`, `opt` and `loop` combined fragments, which is a common but non-standard and restrictive interpretation of such interaction operators [8]. The current paper is an extended and significantly improved version of our previous work [6]. A new translation approach from SDs to extended Petri nets, combining the features of colored Petri nets (CPN) [9] and event-driven Petri nets (EDPN) [10], removes the restriction of lifeline synchronization at decision points and supports the full range of UML interaction operators.

J. P. Faria · A. C. R. Paiva (✉)
INESC TEC and Department of Informatics Engineering, Faculty of Engineering, University of Porto, Porto, Portugal
e-mail: apaiva@fe.up.pt

J. P. Faria
e-mail: jpf@fe.up.pt

Published online: 09 December 2014

Hence the main contributions of this paper are:

– novel techniques for incremental conformance checking of software implementations against UML SDs, complying with the default weak sequencing semantics [1] without lifeline synchronization at decision points, and supporting the full range of interaction operators, based on the translation of SDs to extended Petri nets that combine the features of colored Petri nets (CPN) [9] and event-driven Petri nets (EDPN) [10] (this is a major improvement over our previous work [6]);
– 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 (with minor improvements over our previous work [6]);
– 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 (with minor improvements over our previous work [6]).

The rest of the paper is organized as follows: Sect. 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 several experiments. Section 8 presents a comparison with related work. Section 9 concludes the paper. An "Appendix" presents the translation rules in a visual notation. Because of its size and because it is not essential for understanding the approach presented, we did not include in this paper the detailed formal specification of our conformance checking engine, which can be consulted in an accompanying technical report [11]. The most recent version of our tool implementation is available at https://blogs.fe.up.pt/sdbt/.

2 Overview of the toolset architecture and functioning

Our toolset, named UML Checker, has a modular architecture, comprising two independent tools (see Fig. 1):

– a front-end tool, integrated as an add-in with the Enterprise Architect (EA) modeling environment [12], chosen for its accessibility and functionality;
– a back-end tool, integrated as a run-time test library with the target development environment, implemented in Java and AspectJ [13].

The back-end tool has itself a modular architecture, comprising two major components:

– an execution tracing engine, that takes advantage of AspectJ to trace the execution of the application under test (AUT), inject the behavior of test stubs, and simulate user interaction via the console (command line interface) for testing purposes;
– a conformance checking engine, that translates the expected behavior specified in a SD into an extended Petri net (EPN) and incrementally checks the conformance of AUT execution traces against the expected behavior.

The general functioning of the toolset is as follow: When the user requests conformance testing from the modeling environment, 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 gen-

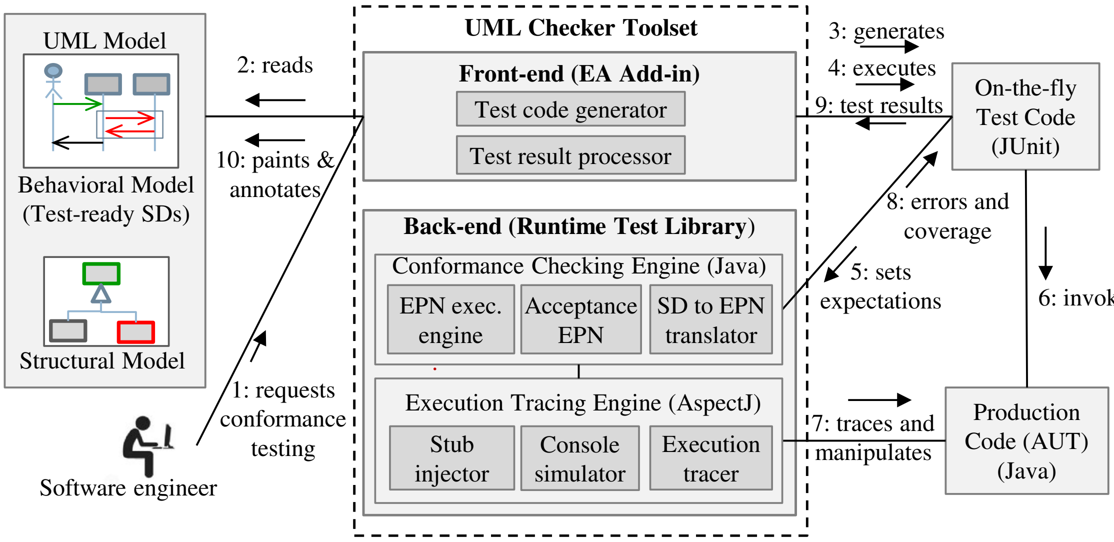


Fig. 1 Communication diagram illustrating the toolset architecture and functioning

erated test code is compiled and executed over the AUT. During test execution, the behavior of the AUT in response to the test inputs (namely internal messages) is traced by the execution tracing engine, and compared against the expected behavior by the conformance checking engine. 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 for visual inspection by the user.

After describing the required characteristics of UML SDs in our approach in the next section, we describe the front-end tool and the two components of the back-end tool in the subsequent sections.

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. Almost all 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`»).

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

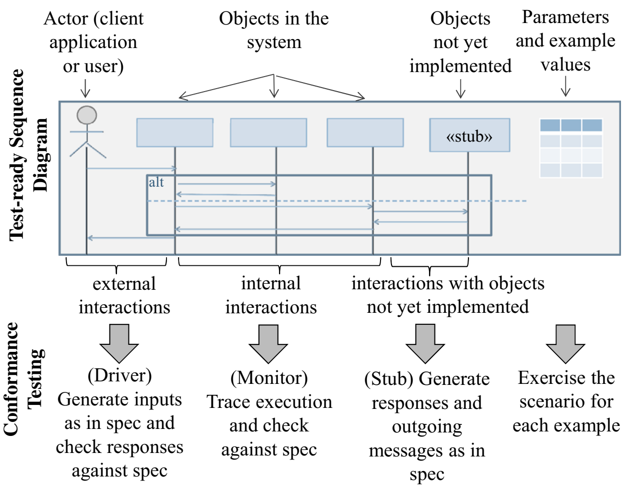
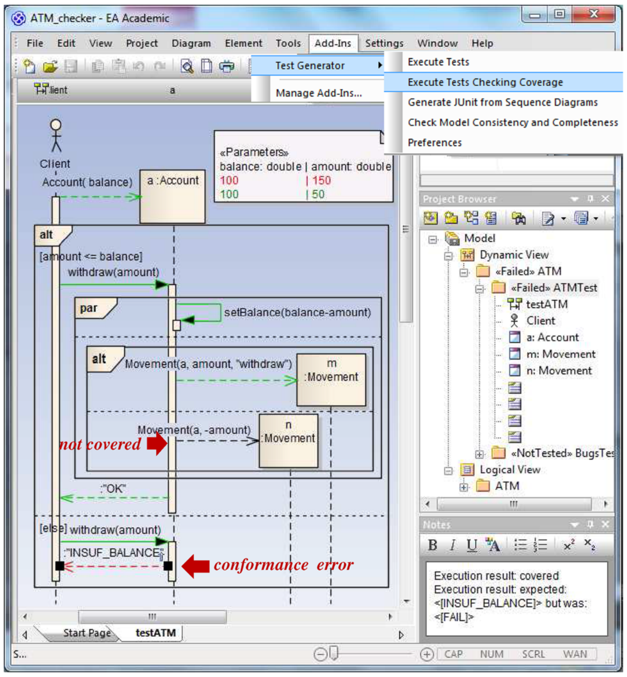

Test-ready Sequence Diagram
Actor (client application or user)
Objects in the system
Objects not yet implemented
Parameters and example values
«stub»
alt
external interactions
internal interactions
interactions with objects not yet implemented
Conformance Testing
(Driver) Generate inputs as in spec and check responses against spec
(Monitor) Trace execution and check against spec
(Stub) Generate responses and outgoing messages as in spec
Exercise the scenario for each example

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



«Parameters»

balance: double	amount: double
100	150
100	50

Execution result: covered
Execution result: expected: <[INSUF_BALANCE]> but was: <[FAIL]>

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

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. An example of user interaction modeling is presented in Fig. 13.

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.

Loose and strict conformance modes 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, i.e., the SD is interpreted as specifying *partial traces* [8]. The other conformance mode is *strict conformance mode*, in which additional messages are not allowed, i.e., the SD is interpreted as specifying *total traces* [8], which is the standard interpretation according to the UML specification [1].

Stubs 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 return messages, but also the outgoing messages specified in the SD for any incoming messages.We call stubs with outgoing messages *stubs in the middle*. This allows testing partial implementations and simulating additional actors.

Interaction operators The full range of interaction operators are supported, allowing the specification of more generic scenarios with control flow variants (with the `alt`, `opt`, `loop`, `break`, `par`, `critical`, `seq` and `strict` interaction operators), the reuse of interactions (with the `ref` operator) and a finer control of the conformance relation (with the `consider`, `ignore`, `assert` and `negate` operators). Conditions of `alt`, `opt`, `loop` and `break` 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.), written in the target language (currently Java), 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 (such as `double` and `Date`).

4 Test code generation and test results visualization in the model

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).

4.1 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 [14] 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 Sect. 5), they are encoded with `Lifeline`—a proxy [14] for the actual object.

4.2 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:

```
public class ATMTest extends InteractionTestCase {
 // Declares lifelines occurring in SD
 private Account a = null;
 Lifeline<Account> aLifeline=new Lifeline<Account>(){
    public Account get() {return a;}
    public void set(Account value) {a = value;}
 }; //...similar for other lifelines

 public void testATM(final double balance,
      final double amount) throws Exception {
  //Declares non-constant expressions occurring in SD
  ValueSpec exp0 = new ValueSpec() {
    public Object get() {return balance-amount;}
  }; //...similar for other expressions
  // Declares expected internal interactions
  expect(/*encoding of SD fragments and messages*/);
  // Traditional JUnit test driver code
  a = new Account(balance);
  if (amount <= balance)
    assertEquals("OK",a.withdraw(amount));
  else
    assertEquals("INSUF_BALANCE",a.withdraw(amount));
  // Final checking of internal interactions missing
  finalCheck();
 }
 public void testATM_0() throws Exception
 { testATM(100, 150); }
 public void testATM_1() throws Exception
 { testATM(100, 50); }
}
```

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

– 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 (not 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.

The behavioral model packages are marked with self-explanatory stereotypes, depending on the status of contained SDs:

– «`Failed`»—a contained SD was tested and failed;
– «`Passed` »—all contained SDs were tested and passed;
– «`NotTested`»—no contained SD was tested;
– «`Incomplete`»—otherwise.

The stereotypes are visible in the project browser for a quick check of conformance status (see Fig. 3).

The classes and operations in the structural model that are not exercised by the behavioral model are also marked as «`NotCovered`», to help assessing the completeness of the behavioral model.

Table 1 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	Operation or constructor execution bar
Unexpected call (in strict conformance mode)	Operation or constructor execution bar
Missing call	Call message or mandatory combined fragment
Missing or incorrect output	`display` message
Missing input	`enter` message
Prohibited behavior encountered	`neg` combined fragment

5 Incremental conformance checking with extended Petri nets

The core of our toolset and approach is the conformance checking engine, which is responsible for translating SDs to extended Petri nets (EPNs) and execute them stepwise as AUT execution events are collected by the execution tracing engine.

5.1 Translating sequence diagrams to extended Petri nets

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, without lifeline synchronization at decision points, we first translate SDs to extended Petri nets (EPN).

The main reason for using Petri nets is because they are a convenient and consolidated means (supported by a large body of research) for expressing concurrency and parallelism. However, since basic Petri nets (PNs) do not have enough expressive power to model all the complex features of SDs, we use a special purpose class of extended Petri nets, combining features from the following sources:

– *Colored Petri nets (CPN)* [9] Whilst in basic PNs tokens are indistinguishable, in CPNs tokens have data values (also called *colors*) attached to them. In our case, there is the need to store auxiliary data, namely the identifiers of pending message occurrences (sent but not received or called but not replied) and loop counters, so tokens also hold data values, being 1 the default value.
– *Event-driven Petri nets (EDPN)* [10] EDPNs are an extension of Petri nets for situations in which the net is not a closed system, but instead interacts with an environment through input and output events. EDPNs use so-called

port event places for holding occurrences of such events. For each input event occurrence placed by the environment in a port event place, the net executes until a new *quiescent state* is reached, i.e., a state where no transition is enabled. We have a similar need for performing conformance checking, where the environment is the AUT mediated by the execution tracing engine. To simplify the representation, in the translation process we do not generate port event places, but instead a notational shorthand—*event-driven transitions*, which are transitions annotated with an *event pattern*.

- *Reset/inhibitor nets* Reset and inhibitor arcs (a syntactic sugar feature once CPNs are used) are useful for handling the translation of some features of SDs, namely critical regions (see Table 7 in the “Appendix”).

Besides combining existing extensions, we also faced the need to use the following special purpose features:

- *Target AUT language as the expression language* In CPNs, arc expressions and transition guards are written in the functional CPN ML language. In our case, since most of the expressions originate from the SD (for message arguments, return values and guard conditions), they are written in the target AUT language and may query the AUT state. On the other hand, in CPNs an arc expression evaluates to a multiset of token colors, whilst in our case we only need a single token.
- *Global variables* In CPNs, variables may be used in arc expressions and transition guards, with transition scope. For a transition to be enabled it must be possible to find a binding of the variables that appear in the surrounding arc expressions of the transition, such that the arc expression of each input arc evaluates to a multiset of token colors that is present on the corresponding input place and the guard condition holds [9]. In our case, besides variables with transition scope, we also need variables with global scope, corresponding to lifelines that are instantiated dynamically (with create messages) and output interaction parameters. Hence, an EPN also has a set of *global variables*, corresponding to the interaction lifelines and parameters. All the occurrences of a lifeline or parameter name must be bound to the same value at run-time. We use the prefix ‘$’ to distinguish variables with transition scope from global scope. The state of an EPN becomes a pair of a *marking* and a *binding* of global variables to values.
- *Place specializations* For convenience, to simplify the diagrams and improve readability, we specialize the notation of places to indicate LIFO, FIFO, start, acceptance and failure places (see Table 2).

Table 2 Notation and features used in the extended Petri nets (EPN)

Name	Notation	Description
Place	○	A place may contain a multiset of tokens. Each token holds a value as in CPNs. When omited, the value 1 is assumed.
FIFO place	⊖	Decoration to indicate that tokens flow in/out according to a first-in first-out policy.
LIFO place	⦶	Decoration to indicate that tokens flow in/out according to a last-in first-out policy.
Acceptance place	◎	Decoration to indicate a success state.
Failure place	⊗	Decoration to indicate a failure state.
Start place	start →○	Decoration to indicate the place that will hold a unit token at begin of execution.
Transition	▬ event pattern [guard]	Transitions may be optionally labeled with a guard condition (as in CPNs) and an event pattern (in which case we call it an *event-driven transition*).
Port event place	▽	A feature from Event-Driven Petri Nets [10] that we use to describe the meaning and processing of event-driven transitions.
Arc	value specification →	Arcs connect places and transitions (as *input* or *output arcs*) and may be labeled with a ValueSpecification, as in CPNs. In our case, the ValueSpecification refers always to a single token, and should not be confused with a arc weight. When omitted, the value 1 is assumed.
Inhibitor arc	○—∘▮	Connects a place and a transition, meaning that the transition cannot fire if the place is occupied.
Reset arc	○—↠▮	Connects a place and a transition, meaning that the transition will remove all tokens (0 or more) from the place.
Any value	-	A value specification that matches any value (as in UML).
Any event	*	An event pattern that matches any event occurrence (as in UML).
Any signature in set	{s1, s2, ...}	An event pattern that matches any occurrence with a signature in the set.
Any signature not in set	$\overline{\{s1, s2, \ldots\}}$	An event pattern that matches any occurrence with a signature not in the set.
Group identifier	$g	A built-in variable, with the scope of a transition, that holds the group identifier of the event occurrence being processed. A group is a pair of send and receive events (for asynchronous messages), or call and reply events (for synchronous messages).

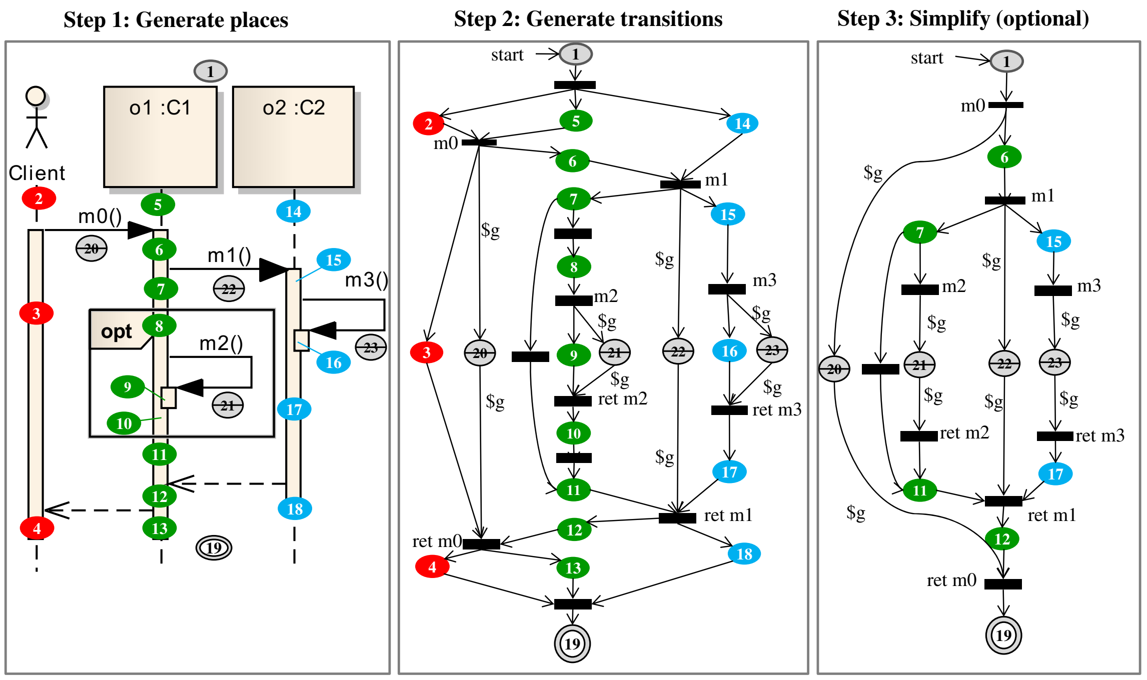


Fig. 5 Three-step translation process from SD to extended Petri net

– *Event patterns* Following the approach of [15], we use a single event to model synchronous communication (instantaneous message passing) and a pair of send and receive events to model asynchronous communication (non-instantaneous message passing). An event pattern combines an event type and a message. Event types are: `call` (default)—exchanging a synchronous call or create message; `ret`—exchanging a synchronous reply message; `send`—sending an asynchronous message; `rcv`—receiving an asynchronous message. Messages have a target lifeline, a signature, and a list of value specifications for the arguments or return value. Wildcards are used to represent any value ('-') and any event ('*'). Set-oriented event patterns, are also used for handling `consider` and `ignore` interaction operators.

Table 2 summarizes the notation and features used. In the rest of the paper the term *extended Petri net* and the acronym *EPN* will be used to refer to such special purpose class.

5.2 Translation procedure

Since SDs may have a complex, non-hierarchical, structure, we follow a three-step translation process from SDs to EPNs as explained next and illustrated in Fig. 5.

1. *Generate places* Possible places are generated in each lifeline before and after each message end (including implicit reply messages from synchronous calls), combined fragment boundary and operand boundary. Intuitively, a lifeline place corresponds to a lifeline state. Additionally, a (global) start place and a (global) acceptance place are introduced for the whole diagram. Auxiliary places, are also generated for each asynchronous message, for each pair of call/reply messages, for each general ordering constraint and for some combined fragments.
2. *Generate transitions* Transitions linking the lifelines' places, possibly with multiple source and/or target places (to handle parallelism and synchronization), are generated according to a set of rules shown visually in the "Appendix" and formalized in [11]. An event-driven transition is generated for each synchronous message, synchronizing the lifelines involved. Regarding asynchronous messages, two event-driven transitions are generated, for the sending and receiving events. Regarding combined fragments, automatic (spontaneous) transitions (without events) are generated to enter and exit the combined fragment and its operands along the lifelines covered. Since boundaries of combined fragments are not considered synchronization points in the UML specification (to properly handle interactions as illustrated in Fig. 6), a translation pattern is used in the decision points of `alt`, `opt`, `loop`, `break` and `neg` fragments to asynchronously coordinate the decisions between the lifelines involved (see "Appendix"). Additionally, it is generated a transition linking the start place of the SD to the first place in all lifelines, and another linking the last place in all lifelines to the final place of the SD.

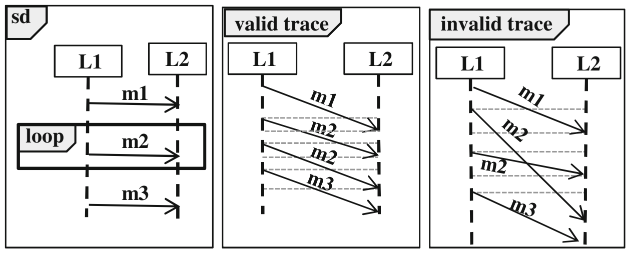


Fig. 6 Example illustrating the need to avoid synchronization at combined fragment boundaries or decision points. If L1 and L2 were forced to synchronize at the boundaries and at each iteration of the loop, the valid trace shown would not be accepted

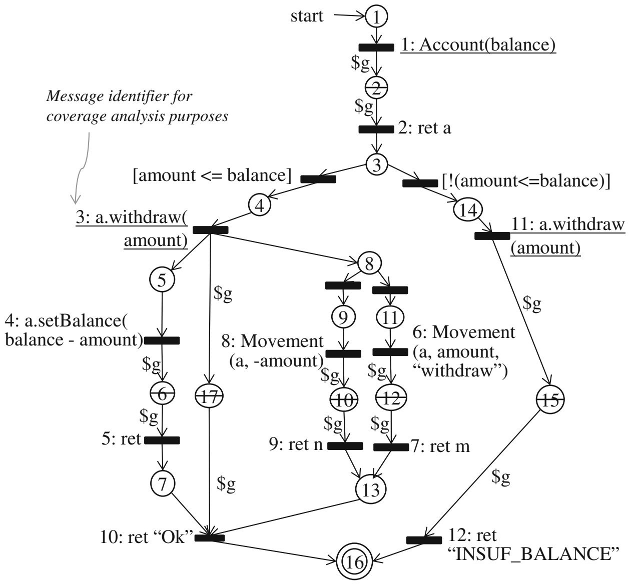


Fig. 7 Extended Petri net generated from the example SD of Fig. 3 (only partially simplified)

3. *Simplify* (optional) The resulting EPN is simplified by removing transitions with empty labels and redundant places, resulting in an equivalent EPN that accepts the same traces.

Another example partially simplified is shown in Fig. 7.

5.3 Extended Petri net execution

5.3.1 Strict conformance mode

We start by describing the execution in strict conformance mode.

As explained before, event-driven transitions are just a shorthand notation. Their meaning may be described by the rewriting illustrated in the top of Fig. 8. Two (global) places

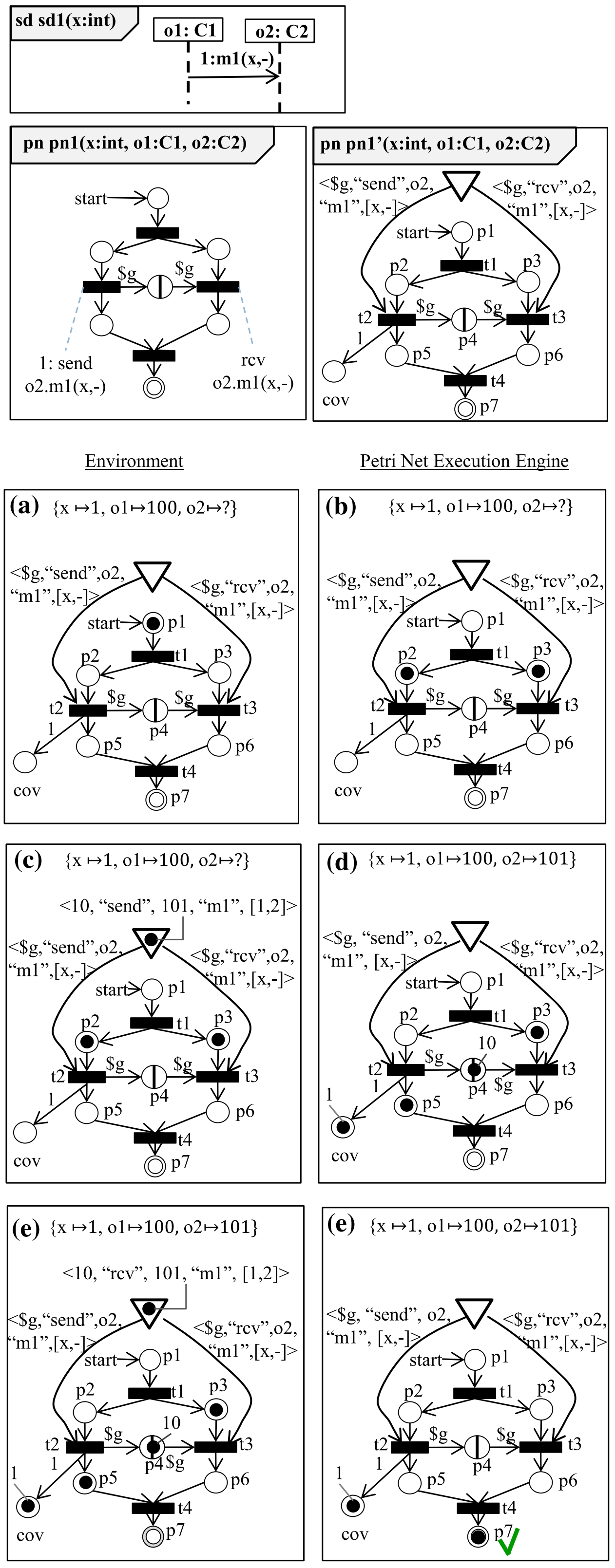

sd sd1(x:int)
o1: C1
o2: C2
1:m1(x,-)
pn pn1(x:int, o1:C1, o2:C2)
pn pn1'(x:int, o1:C1, o2:C2)
start
1: send o2.m1(x,-)
rcv o2.m1(x,-)
<$g,"send",o2,"m1",[x,-]>
<$g,"rcv",o2,"m1",[x,-]>
Environment
Petri Net Execution Engine
(a) {x ↦1, o1↦100, o2↦?}
(b) {x ↦1, o1↦100, o2↦?}
(c) {x ↦1, o1↦100, o2↦?}
(d) {x ↦1, o1↦100, o2↦101}
(e) {x ↦1, o1↦100, o2↦101}
<10, "send", 101, "m1", [1,2]>
<10, "rcv", 101, "m1", [1,2]>
p1 p2 p3 p4 p5 p6 p7
t1 t2 t3 t4
cov

Fig. 8 EPN execution described in terms of an event-driven PN

are introduced: a port event place, to store incoming events from the environment (only one at a time) and a place labeled *cov* to store the identifiers of message covered. Each event-driven transition is rewritten by connecting it to the port event place by an input arc labeled with the event pattern rewritten as a 5-tuple with the following elements:

- a variable with transition scope for the group identifier of incoming event occurrences;
- a constant that indicates the event type;
- a global variable that identifies the target lifeline of the message, in case of a non-static lifeline, or the name of the target classifier, in case of a static lifeline;
- the name of the message (operation or signal);
- a list of value specifications for the return values (in case of reply messages) or arguments (in all other cases).

Arcs are also introduced to connect each event-driven transitions to the *cov* place, annotated with the message identifier.

A run state of the rewritten EPN is a pair of a marking M (a mapping from places to collections of tokens) and a binding B of global variables—lifelines (B_L) and interaction parameters (B_P)—to actual values. Initially, a single unit token is placed in the start place, and input interaction parameters and previously instantiated lifelines are bound to actual values or objects. This binding is incrementally extended as message occurrences are encountered involving lifeline names as target, argument or return value or output parameters as argument or return value; subsequent occurrences of a previously bound lifeline name must refer to the same value.

The semantics of transition firing is the same as with CPNs, with the difference that variables may have transition scope or global scope. To comply with the interleaving semantics of interactions in UML, one transition executes at a time. For each event occurrence from the AUT, transitions are fired until a new quiescent state of the EPN is reached.

The execution steps of the rewritten EPN (pn1') in Fig. 8 for a valid trace are also illustrated there and are explained next:

- Step 1. The EPN is initialized from the environment by placing a unit token in the start place $p1$ and providing bindings for some global variables (in this case, all variables except $o2$)—run state a. Then the EPN is executed one transition at a time by the execution engine until a new quiescent state is reached. The only enabled transition in state a is $t1$ which is fired, removing the unit token from $p1$ and adding a unit token to $p2$ and $p3$—run state b. Since no further transitions are enabled, a quiescent state is reached.
- Step 2. An event occurrence from the environment is placed in the port event place—run state c. The event occurrence is described by a 5-tuple with constants for the group identifier, the event type, the identifier of the message target, the message name, and the arguments. In this case, only transition $t2$ becomes enabled, with the bindings $\{x \mapsto 1$(previously bound), $\$g \mapsto 10$(temporary binding) , $o2 \mapsto 101$(newly bound)$\}$. The transition $t2$ fires, placing the value 1 (message identifier) in the *cov* place, the group identifier (10) in $p4$ and a unit token in $p5$—state d. The global variable $o2$ also becomes bound. Since there is no further enabled transition, execution stops in the new quiescent state d.
- Step 3. A new event occurrence from the environment is placed in the port event place—run state e. Only transition $t3$ becomes enabled, with the bindings $\{x \mapsto 1$ (previously bound), $\$g \mapsto 10$, $o2 \mapsto 101$(previously bound)$\}$. After firing $t3$, transition $t4$ becomes enabled. After firing $t4$ a final quiescent state f is reached, which is an acceptance state.

5.3.2 Loose conformance mode

The meaning of the loose conformance mode can be described by the rewriting indicated in Fig. 9, where a new place *ign* and two transitions are introduced. In this mode, any event may potentially be ignored. Even when a *call* or *send* event occurrence could be considered, the alternative of ignoring it is also explored, because that may be needed for the complete execution trace to be accepted. Hence, a new place *ign* is introduced to store the group identifiers of ignored *call* or *send* event occurrences (waiting for the corresponding *ret* or *rcv* occurrences). The *ret* and *rcv* occurrences may be ignored iff the corresponding *call* or *send* occurrences (with the same group identifier) were previously ignored. For that reason, the right-hand transition has an input arc from the *ign* place annotated with a variable that must match the group identifier of the incoming occurrence.

An example of execution in loose conformance mode is illustrated in Fig. 10. In this case we did not rewrite the EPN, so the covered messages (C) and ignored event occurrences (I) are also kept in the run state.

5.3.3 Disambiguation and verdict determination

EPNs may be non-deterministic, i.e., multiple transitions may be enabled at the same time, not only because of uncon-

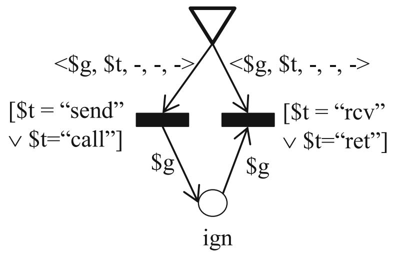


Fig. 9 Loose conformance mode adds two transitions and one place

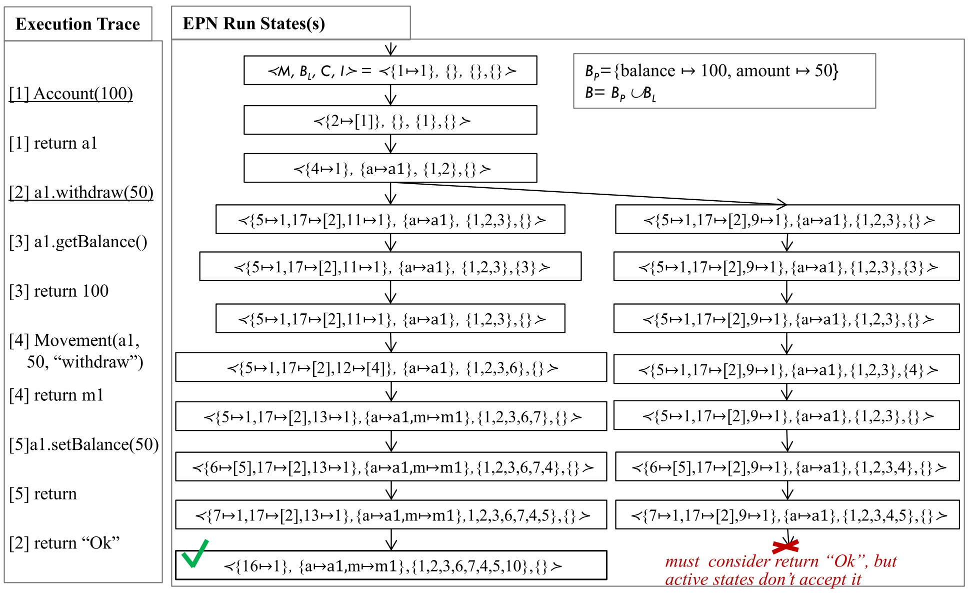

Execution Trace
[1] Account(100)
[1] return a1
[2] a1.withdraw(50)
[3] a1.getBalance()
[3] return 100
[4] Movement(a1, 50, "withdraw")
[4] return m1
[5]a1.setBalance(50)
[5] return
[2] return "Ok"
EPN Run States(s)
<M, B_L, C, I> = <{1↦1}, {}, {},{}>
B_P={balance ↦ 100, amount ↦ 50}
B= B_P ∪B_L
<{2↦[1]}, {}, {1},{}>
<{4↦1}, {a↦a1}, {1,2},{}>
<{5↦1,17↦[2],11↦1}, {a↦a1}, {1,2,3},{}>
<{5↦1,17↦[2],9↦1}, {a↦a1}, {1,2,3},{}>
<{5↦1,17↦[2],11↦1}, {a↦a1}, {1,2,3},{3}>
<{5↦1,17↦[2],9↦1}, {a↦a1}, {1,2,3},{3}>
<{5↦1,17↦[2],11↦1}, {a↦a1}, {1,2,3},{}>
<{5↦1,17↦[2],9↦1}, {a↦a1}, {1,2,3},{}>
<{5↦1,17↦[2],12↦[4]}, {a↦a1}, {1,2,3,6},{}>
<{5↦1,17↦[2],9↦1}, {a↦a1}, {1,2,3},{4}>
<{5↦1,17↦[2],13↦1}, {a↦a1,m↦m1}, {1,2,3,6,7},{}>
<{5↦1,17↦[2],9↦1}, {a↦a1}, {1,2,3},{}>
<{6↦[5],17↦[2],13↦1}, {a↦a1,m↦m1}, {1,2,3,6,7,4},{}>
<{6↦[5],17↦[2],9↦1}, {a↦a1}, {1,2,3,4},{}>
<{7↦1,17↦[2],13↦1}, {a↦a1,m↦m1},1,2,3,6,7,4,5},{}>
<{7↦1,17↦[2],9↦1}, {a↦a1}, {1,2,3,4,5},{}>
<{16↦1}, {a↦a1,m↦m1},{1,2,3,6,7,4,5,10},{}>
must consider return "Ok", but active states don't accept it

Fig. 10 Example of successful execution of the EPN of Fig. 7 for a possible test case and AUT response in loose conformance mode. The test driver stimuli are *underlined* in the trace. Group identifiers of event occurrences are indicated *inside square brackets* in the execution trace

strained choices, but also because multiple event patterns may be matched by the same event occurrence. E.g., the patterns `m(1)` and `m(-)` are both matched by the occurrence `m(1)`. In case multiple transitions are enabled, multiple run states are kept, so that all possible executions are explored. On the other hand, run states that are not able to consume an event occurrence placed in the port event place are discarded, reducing the set of existing run states.

Denoting by $R = \{r_1, \ldots, r_n\}$ the set of run states, we use the following criteria to determine a test verdict and result:

– *pass*: conformance checking succeeds (i.e., the execution trace is considered valid) when, at the end of execution, R contains no run state with a failure place occupied and contains at least one run state with the acceptance place occupied. In case multiple run states in R have the acceptance place occupied, it is selected one with a maximal set in the *cov* place to determine test coverage (messages covered);
– *fail*: conformance checking fails (i.e., the execution trace is considered invalid) when, at any point of execution, R becomes empty or contains at least one run state with a failure place occupied; in case a failure place becomes occupied, a corresponding error explanation is provided to the user (prohibited behavior encountered) and message coverage information is provided based on the contents of the *cov* place; for the cases where failure occurs because R becomes empty, in order to provide meaningful coverage and error information to the user, it is incrementally kept a *best match*, corresponding to a run state reached with a maximal set of covered messages (stored in the *cov* place) and a best-matching event occurrence (same target and signature but different arguments or return value, etc.).

An executable formal specification of the translation and execution procedures, together with a set of executable test cases, can be found in our technical report [11].

6 Execution tracing and manipulation with AOP

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

6.1 Execution tracing and stub injection

Method and constructor invocation and execution in the AUT are intercepted with the AspectJ [13] code depicted in Fig. 11. Method invocations are traced with an `execution` pointcut (line 6), 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 20 and 24). The call and reply occurrences intercepted by the aspect code are sent to the conformance checking engine for incremental checking (lines 8 and 15), with the same group identifier (generated in line 7).

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)

```
1.  public priviliged aspect TracingAspect {
2.    // Auxiliary definition to filter points of
3.    // interest and avoid infinite recursion
4.    pointcut mayTrace():  /* details omitted */;
5.    // Intercepts normal and reflective method calls
6.    Object around(): mayTrace() && execution(* *(..)) {
7.      Generate a unique identifier for the call-reply pair
8.      Handle call occurr.by EPN via synchronized method
9.      If conformance checking fails, throw the failure
10.     If the target matching object is marked as stub,
11.         Execute outgoing calls specified in SD
12.         Get specified return value in SD
13.     else
14.         Proceed normal execution and get return value
15.     Handle reply occurr. by EPN via synchronized method
16.     If conformance checking fails, throw the failure
17.     Return the return value
18.   }
19.   // Intercepts normal constructor calls
20.   Object around(): mayTrace()&& call(new(..) {
21.      similar template
22.   }
23.   // Intercepts reflective constructor calls
24.   Object around(): mayTrace()
25.    && call(Object Constructor.newInstance(..)) {
26.      similar template
27.   }
28. }
```

Fig. 11 Skeleton of AspectJ code responsible for execution tracing and stub injection

are issued through reflection (line 11), and it is returned the value specified in the SD (line 12).

6.2 User interaction testing

The mechanisms for user interaction testing of console applications are illustrated in Fig. 12.

A console simulator (from the runtime 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 `scan` and `print` 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 (in the points indicated by red circles in Fig. 12) to the conformance checking engine for checking their proper ordering with respect to other execution occurrences. A group identifier for each pair of *send* and *rcv* event occurrences is generated each time a value is put in a blocking queue. Currently, event occurrences of type *send* and *rcv* are generated

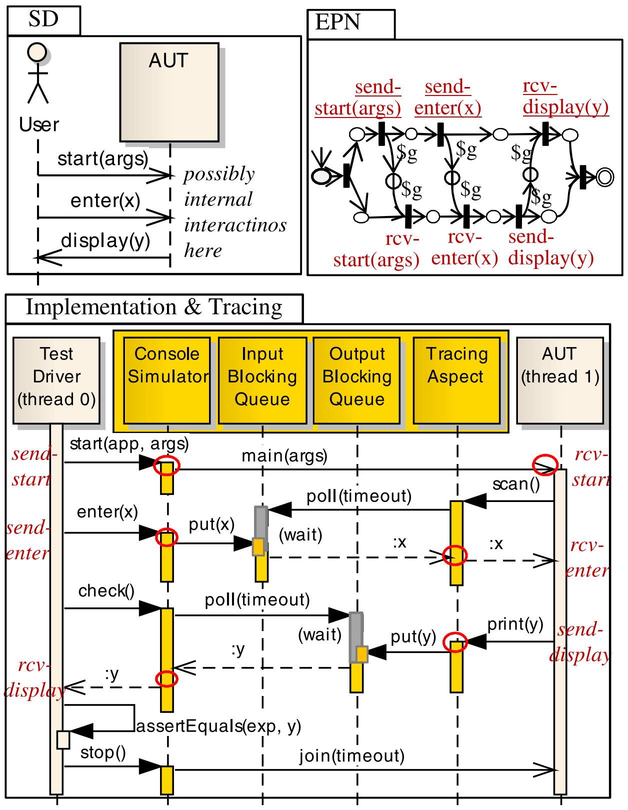


Fig. 12 User interaction specification and conformance testing mechanism for console applications

by the execution tracing engine only for user interaction, as described in this section.

7 Experiments

7.1 Conformance testing 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).

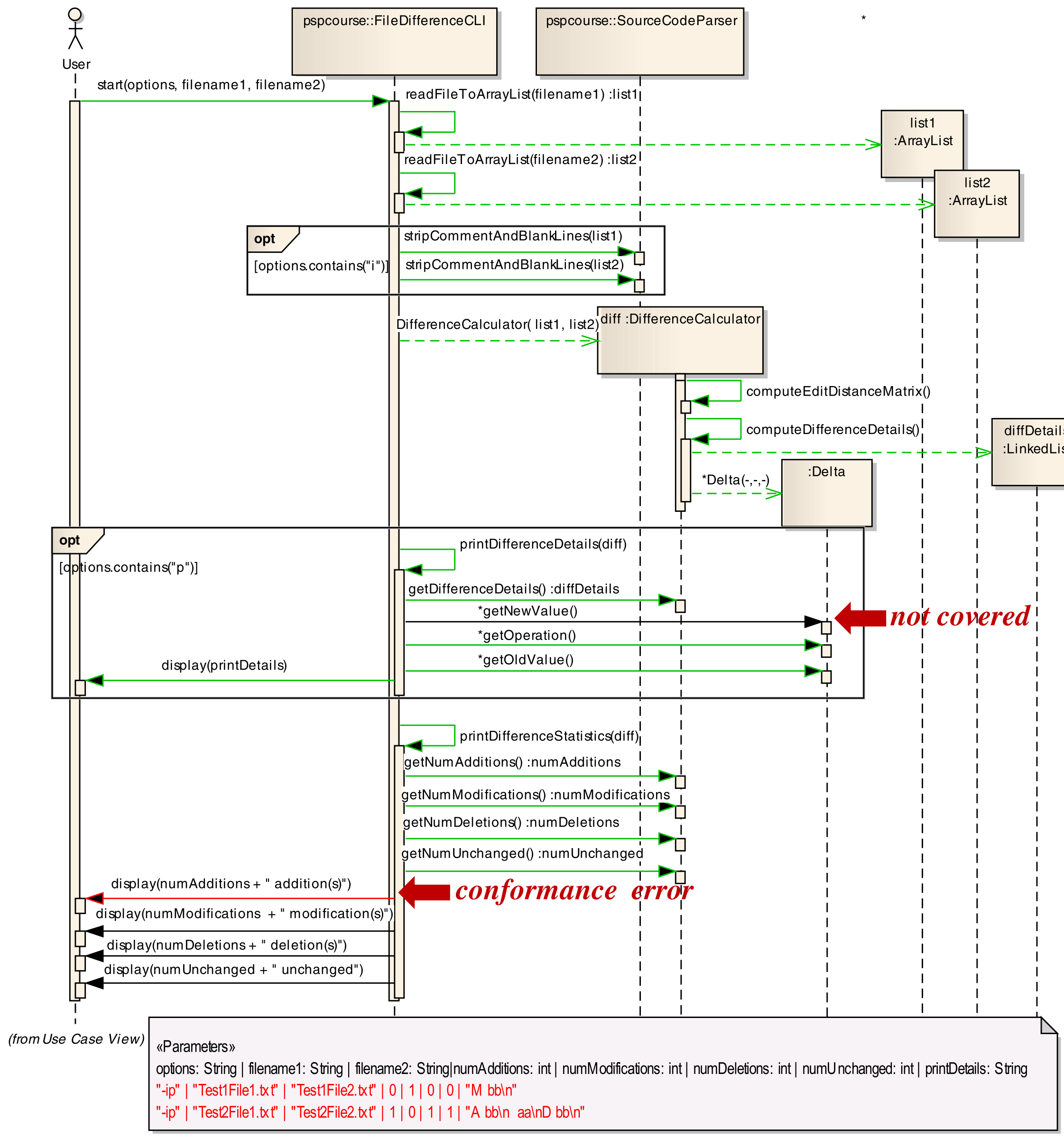

User
pspcourse::FileDifferenceCLI
pspcourse::SourceCodeParser
start(options, filename1, filename2)
readFileToArrayList(filename1) :list1
list1
:ArrayList
readFileToArrayList(filename2) :list2
list2
:ArrayList
opt
[options.contains("i")]
stripCommentAndBlankLines(list1)
stripCommentAndBlankLines(list2)
DifferenceCalculator(list1, list2)
diff :DifferenceCalculator
computeEditDistanceMatrix()
computeDifferenceDetails()
diffDetails
:LinkedList
:Delta
*Delta(-,-,-)
opt
[options.contains("p")]
printDifferenceDetails(diff)
getDifferenceDetails() :diffDetails
*getNewValue()
not covered
*getOperation()
*getOldValue()
display(printDetails)
printDifferenceStatistics(diff)
getNumAdditions() :numAdditions
getNumModifications() :numModifications
getNumDeletions() :numDeletions
getNumUnchanged() :numUnchanged
display(numAdditions + " addition(s)")
conformance error
display(numModifications + " modification(s)")
display(numDeletions + " deletion(s)")
display(numUnchanged + " unchanged")
(from Use Case View)
«Parameters»
options: String | filename1: String | filename2: String|numAdditions: int | numModifications: int | numDeletions: int | numUnchanged: int | printDetails: String
"-ip" | "Test1File1.txt" | "Test1File2.txt" | 0 | 1 | 0 | 0 | "M bb\n"
"-ip" | "Test2File1.txt" | "Test2File2.txt" | 1 | 0 | 1 | 1 | "A bb\n aa\nD bb\n"

Fig. 13 Test-ready sequence diagram of a case study, *colored* after test execution color figure online

The initial SD ([16, pg. 67]) was not test-ready, due to the usage of pseudo-code and a procedural feature (attribute assignment) not currently supported by our tool (because its verification would require tracing attribute assignment). 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 shown in Fig. 13. 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. After defect fixing, the final test-ready SD reached 86 % statement coverage (measured with Jacoco [17]), being the statements not covered related with the handling of exceptional conditions.

Hence, the benefits of our approach could be demonstrated for this case study. Other case studies performed to validate the approach for all supported modeling features, error types and coverage levels can be found in [16].

Furthermore, to assess the time efficiency of model-based versus manual test code generation, we computed the data presented in Table 3 for the FileDiff application. The data suggests that there is the potential to recover in the short-term the time invested in creating test-ready behavioral models (valuable also for specification and documentation purposes), assuming the same test code is to be generated both ways (manually or automatically). A performance compari-

Table 3 Effort estimates for model-based and manual test code generation in the FileDiff case study

Item	Value
A. Average productivity of the same developer in the Design phase over 8 projects	56 modeling elements per hour[a]
B. Average productivity of the same developer in the Coding phase over 8 projects	98 lines of code per hour[b]
C. Size of the test-ready behavioral model of Fig. 13	40 modeling elements
D. Size of the test code automatically generated from the behavioral model	82 lines of code
E. Estimated effort needed to create the behavioral model from scratch (and have the test code generated automatically) ($E = C/A$)	44 min
F. Estimated effort needed to manually write the same test code ($F = D/B$)	50 min

[a] The modeling elements counted are added or modified classes, packages, notes, relationships, operations, attributes, lifelines, messages, combined fragments, combinations of parameter values, etc

[b] The lines of code counted are added or modified source lines of code, ignoring blank lines and comments

son with testing approaches involving significantly different test code will be the subject of future studies.

7.2 Toolset performance analysis

In order to assess the performance and scalability of our toolset, we measured its execution time for two small case studies—the FileDiff case study previously presented and the SpreadsheetEngine case study presented in [4]—as well as a larger system—the Google's Guava libraries [18].

The Guava libraries are a set of core libraries, with approximately 62K source lines of code (ignoring comments and blank lines), that Google relies on in their Java-based projects. Since UML design models are not available in the Guava web site, we created a sample SD that partially illustrates the usage and internal design of one of its collections (see Fig. 14). Our goal with this subject system was to assess the impact of the AUT size on the performance of the UML Checker toolset, as well as its applicability for modeling and testing selected parts of larger real world software systems.

The performance results are presented in Table 4. For each system, it is presented the execution time for the different phases of the test generation, compilation, execution and result analysis process. The phases are executed together with a single invocation of our plug-in (inside the EA modeling environment), which presents in the end the total time elapsed and the time elapsed per phase.

The experiment was conducted on an ASUS U36S laptop, with a 64-bit Intel® Core™i5-2430 CPU at 2.4 GHz, with 4 GB RAM, and the Windows 7 operating system. The Java and UML files of the systems analyzed are available in the toolset web site at https://blogs.fe.up.pt/sdbt/.

In order to assess the overhead of aspect weaving, we repeated the experiment with compile time and load time weaving. By comparing the execution times for both modes in Table 4 we conclude that: (1) the overall test code compilation and execution time (phases 2 and 3) is dominated by the aspect weaving overhead; (2) the aspect weaving overhead is smaller with compile time weaving. An explanation for the latter is that with compile time weaving only the test code has to be woven (the AUT is previously compiled and woven), whilst with load time weaving both the test code and the AUT code have to be woven.

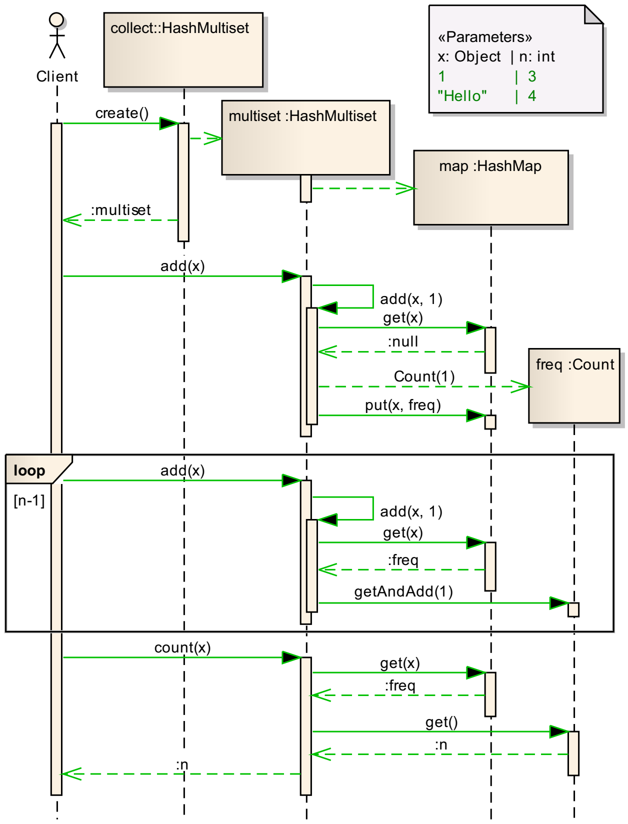


Fig. 14 Test-ready SD illustrating part of the internal design of HashMultisets in Google's Guava libraries (*colored* after test execution) color figure online

Table 4 Toolset execution times

	FileDiff	SpreadsheetEngine	Guava
Execution time information (elapsed time, in seconds)			
1. Generate test code from model	5.06	4.87	3.34
2. Compile test code			
With compile time weaving	2.08	2.15	1.72
With load time weaving	0.67	0.66	0.72
3. Execute test code			
With compile time weaving	0.28	0.76	0.50
With load time weaving	2.84	5.79	5.69
4. Annotate model w/ test results	4.23	6.18	2.99
Total			
With compile time weaving	11.7	13.9	8.5
With load time weaving	12.8	17.5	12.8
Size of behavioral model			
Number of elements	40 (1 SD)	64 (2SDs)	30 (1 SD)

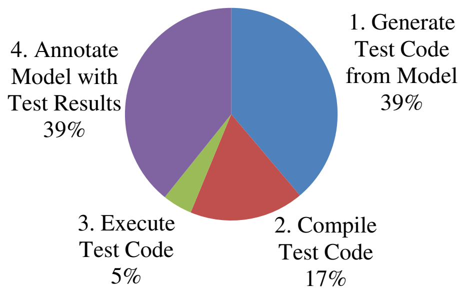


Fig. 15 Mean execution time distribution with compile time weaving

Nevertheless, in compile time weaving mode (the most efficient one), the overall execution time is dominated by the model processing phases (phases 1 and 4), consuming 78 % of the total time on average (see Fig. 15). To assess its impact on the execution time, we computed the size of the behavioral model of each subject system. Although this is a very small data set, it can be observed a trend for a linear relationship between the size of the behavioral model and the execution time, irrespective of the size of the AUT.

In a fine-grained analysis, we observed that most of the model processing time is consumed by calls to the OLE Automation (ActiveX) interface provided by EA, so we plan to explore in the future other ways to access the model (e.g., by accessing directly the database that stores the model).

8 Related work

8.1 Semantics and interpretation of sequence diagrams

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 [8].

Based on a survey of proposed semantics for UML SDs, Micskei et al. [8] 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 can be classified as follows: execution traces are either valid or invalid, i.e., no inconclusive traces exist, to avoid inconclusive tests; the underlying formalism is based on the encoding of the partial orders into a finite structure (extended Petri net), for incremental on line processing, with interleaving as the concurrency model, as in the UML standard; both complete (in strict conformance mode) and partial trace specifications (in loose conformance mode) are supported; fragments are combined using standard interpretation with weak sequencing; choices and guards are handled without requiring lifelines to 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 places in our case), for higher flexibility; event occurrences are represented by tuples + message IDs from external facilities (the tracing engine), so that pairs of send/receive and call/reply event occurrences are properly matched.

As pointed out in [8], the first step of many proposed semantics (e.g., [19]), 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 places (intuitively representing lifeline states), we avoid determining those cuts, as well as the potential explosion of states and transitions.

8.2 Usage of sequence diagrams for verification and validation

Despite the challenges with the SD semantics, there are different approaches in the literature that use SD for quality purposes. For instance, [20] translate UML 2 interactions into automata for model checking with respect to specified requirements.

Of particular interest in the context of this paper, are some approaches that translate SDs to Petri nets [21,22], for model-checking or simulation purposes.

In [21], it is presented an approach to translate SDs into CPNs, for model-checking purposes. Besides using different rules from our approach, their goal is to formalize the translation process in order to perform model checking (using PROD) to prove properties on the generated CPN, such as, deadlock, livelock, reject states, quasi-liveness, boundness and reinitializability. As they generate a CPN as an isolated system, the translation rules are significantly different from our approach. In particular, they store in the marked CPN the objects' state (attribute values), but then such approach seems to require the modeling of all side-effects in the SD (with attribute assignment), which is contrary to their usual focus—modeling messages exchanged between lifelines and not internal processing and updates in each lifeline (at least completely). Besides, they do not describe a number of interaction operators and features (such as `critical`, `assert`, `neg`, `ignore`, `consider`, and unbound loops).

In [22], the authors translate some basic UML 2.0 operators (`opt`, `alt`, `par`, `loop` and `ref`) to CPNs, but the transformation is not yet automated and, like in the previous case, the generated CPNs are isolated systems. Their goal is to build graphical animations on top of the CPN models for requirements elicitation, validation and specification.

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 [23] 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 testing and test 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 [24]. 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 [25], message dependency graphs [26], and structured composite graphs [27].

Besides generating test sequences, there are some approaches that also generate test data. Nayak et al. [27] 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. [26] create dynamic slices according to conditional predicates associated with messages in a SD and generate test data satisfying each slice. Benauttou et al. [28] generate test data based on partition analysis of method contracts expressed in the Disjunctive Normal Form.

Some of the above approaches can be combined with our approach to automatically generate test data for exercising different paths in complex SDs.

8.3 Usage of sequence diagrams as test specifications

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 [29] 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 [30]); a tool generating functional test drivers from SDs (SCENTOR [31]); a Model-Driven Architecture based approach for generating test code for multiple unit testing frameworks [32].

Javed et al. [32] 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 UML2.

One advantage of SeDiTeC [30] 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 [31] tool creates functional test drivers for e-business applications from SDs that have embedded test

Table 5 Comparison of modeling and testing features supported by different approaches for test generation and execution based on SDs

	SeDiTeC [30]	Javed et al. [32]	SCENTOR [31]	Test conductor [33]	UML checker (ours)
Interaction parameters	Partly[a]	Partly[a]	Partly[a]	Yes	Yes
Keyword-based UI testing	No	No	No	No	Partly[b]
Internal interaction checking	Yes[c]	Partly[d]	No	Partly[e]	Yes[f]
Loose conformance checking	Yes	No[g]	No	Partly[h]	Yes
Test stub injection	Yes[i]	No	No	Partly[j]	Yes[k]
Interaction operators	No[l]	No	No	Partly[m]	Yes
Complex value specifications	No	No	No	No	Yes[n]
Test code generation	No[o]	Yes[p]	Yes	Yes	Yes
Test results in the model	Partly[q]	No	No	Yes	Yes
Model coverage analysis	No	No	No	Yes	Yes

[a] Only message parameters are supported (call and return values)
[b] Currently only for console applications
[c] With code instrumentation. Actual capabilities are unclear
[d] Internal messages are traced but have to be manually checked
[e] Internal messages exchanged between objects in code generated by the tool are checked, but not internal object creation
[f] With AOP
[g] Since internal messages have to be manually checked, no automated conformance checking criteria is provided
[h] Message types mentioned in the SD are considered, all others are ignored
[i] With code generation
[j] Return values are injected, but not messages originated from stubs
[k] With AOP and reflection
[l] However, some operators may be simulated by combining SDs
[m] `alt`, `opt` and `loop` operands without guards are not supported, as well as some advanced operators
[n] Complex expressions for message parameters and return values, without side effects on participating objects, are supported
[o] Only run-time interpretation of SDs is supported
[p] Using Model-Driven Architecture (MDA) techniques
[q] It is produced an 'observed' SD, but the original SD is not marked

data (parameters and expected values of method calls), but does neither check internal interactions nor generate test stubs.

Some commercial tools support conformance testing based on SDs. To our knowledge, the IBM Rational Rhapsody TestConductor Add On [33] is one of the more advanced tools. Targeting real-time embedded applications, it supports many features in common with our approach (like internal interaction checking and visual feedback) 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 [23]. An approach for coordinating distributed test components (namely test drivers) was presented in [34].

Table 5 presents a comparison of the features supported by the approaches analyzed with similar scope as ours (i.e., supporting test generation and execution based on UML SDs).

9 Conclusions and future work

It were presented a set of techniques and a toolset for the automatic conformance testing of software applications against partial 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 extended Petri nets 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 approach can be applied for other technologies.

The ability to test internal interactions (method calls and object instantiation) in object-oriented systems in a simple and flexible way (based on UML 2 SDs) is one of the main benefits of the approach and toolset presented. The same kind of testing would require complex programming with unit testing frameworks, and does not have a similar level of support in other test automation approaches analyzed based on UML SDs. Testing internal interactions may help in fault localization, in assuring consistency between implementation and specification, and in test suite minimization.

In the context of development processes that emphasize the construction of lightweight design models, such as Agile Modeling [35], the toolset presented can be used together with existing code generation facilities from structural models to support a hybrid model-driven engineering approach, combining partial production code generation from structural models and test code generation from partial behavioral models (playing the role of executable test specifications). With such an approach, the time invested in constructing design models can be recovered in the short term, at least partially, and the consistency between implementation and specification can be better assured.

As future work, we plan to: develop front-ends for other modeling environments (reusing the runtime test library); extend the conformance checking engine to support duration and time constraints (with timed Petri nets); extend the execution tracing engine to support the testing of distributed systems; 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; conduct further experiments to measure the quality and productivity benefits that can be achieved with the model-based testing approach presented here, as compared to traditional approaches based on unit testing frameworks.

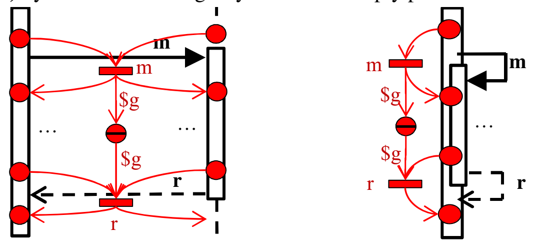
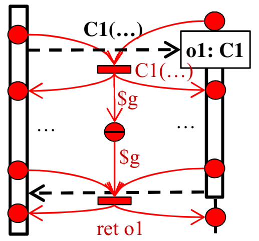
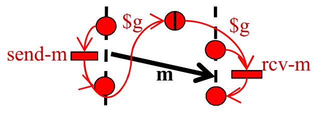
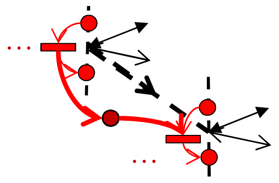
Acknowledgments This work is financed by the ERDF European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT—Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-020554.

Appendix: Transition generation rules

The following tables present the transition generation rules for several types of interactions' features. The generated elements are superimposed in red.

See Tables 6, 7, 8, 9 and 10.

Table 6 Translation of messages and ordering constraints

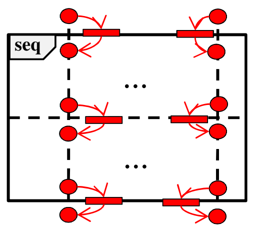
(a) synchronous message: synchCall and reply pair  m m $g $g r r ... ... m m $g $g r r ...
(b) synchronous message: createMessage and reply pair  C1(...) o1: C1 C1(...) $g $g ... ... ret o1
(c) synchronous message: asynchCall or asynchSignal 
(d) general ordering constraint 

Notes:

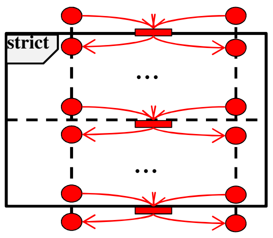
- a,b) An auxiliary LIFO place is used to store the group identifier of pairs of call/replay occurrences, to make sure that reply occurrences corresponding to ignored call occurrences are also ignored. Explicit or implicit reply messages are always assumed at the end of each execution bar.
- c) An auxiliary FIFO place is used to store the identifiers of messages sent, to make sure that receive occurrences corresponding to ignored send occurrences are also ignored, and that, in the presence of multiple pending messages inside a loop, the ordering of messages is preserved.
- a,b,c) The rules also apply for messages that cross the boundaries of combined fragments, via explicit or implicit gates (a means used in UML to let messages cross combined fragments' boundaries).
- d) In our interpretation, a before 'event' must have occurred for each 'after' event to occur. A different interpretation would be to use an inhibitor arc linked to the 'before' event, in which case the 'after' event could occur also if the 'before' event did not occur.

Table 7 Translation of sequencing operators

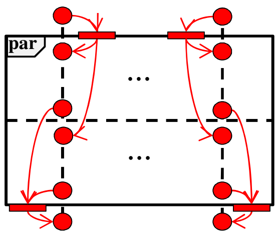
(e) weak sequencing



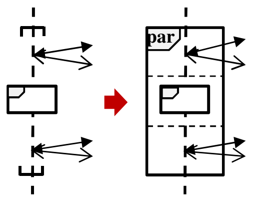
(f) strict sequencing



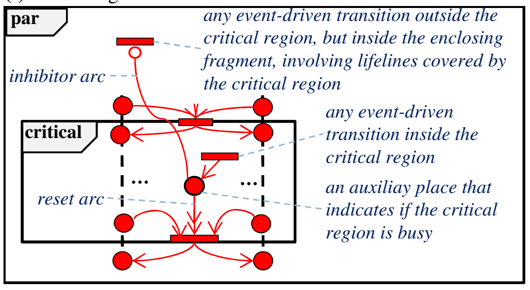
(g) parallel



(h) co-region



(i) critical region

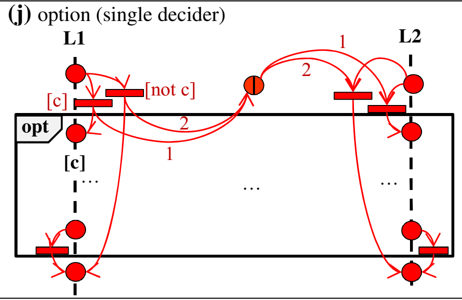

par
any event-driven transition outside the critical region, but inside the enclosing fragment, involving lifelines covered by the critical region
inhibitor arc
critical
any event-driven transition inside the critical region
reset arc
an auxiliay place that indicates if the critical region is busy

Notes:

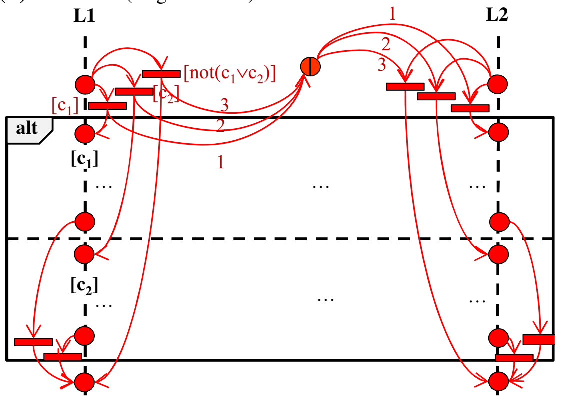
- e,f,g) These rules extend trivially to more operands and/or lifelines.
- h) According to the UML spec ([1], pg. 486), a coregion area is just a notational shorthand for a `par` fragment covering a single lifeline having as operands the enclosed fragments.
- i) According to the UML spec ([1], pg. 484), "a critical region means that the traces of the region cannot be interleaved by other OccurrenceSpecifications (on the Lifelines covered)."

Table 8 Translation of option and alternative

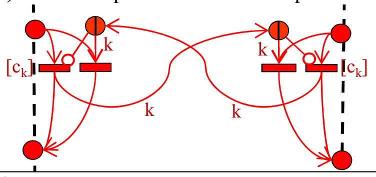
(j) option (single decider)



(k) alternative (single decider)



(l) translation pattern for a decision point with multiple "deciders"

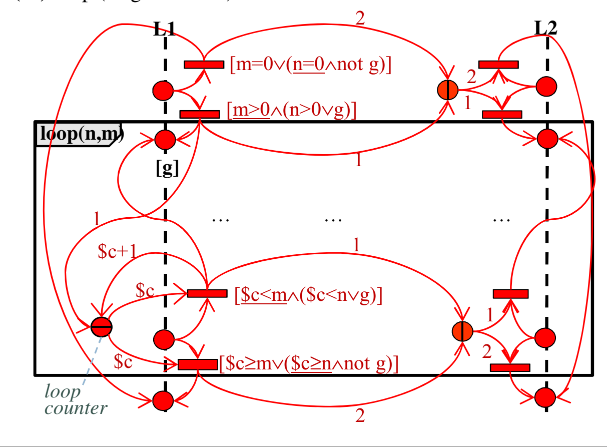


Notes:

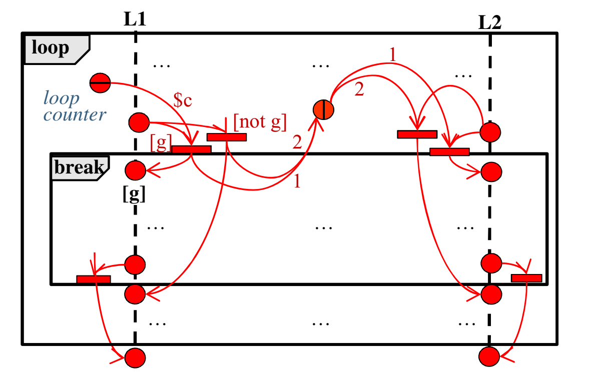
- j,k) In order to comply with the UML spec, lifelines are not forced to synchronize at decision points. Instead, decisions are coordinated asynchronously between lifelines as follows: the first lifeline to take a decision in a decision point (involving the evaluation of a guard condition or a free choice), sends the decision (a number identifying the selected continuation) to the remaining lifelines in a way similar to an asynchronous message. When another lifeline reaches the same decision point it just follows the decision taken by the first lifeline.
- j, k) In the absence of guard in the SD, all guards are also omitted in the generated transitions on L1.
- k) In case the guard is `else` or is omitted in one of the operands, the option to skip the combined fragment doesn't exist.
- l) This translation pattern is applied in case there isn't a single "decider" lifeline (owning the first event after any of the possible outcomes of the decision). The pattern is repeated for each of the possible outcomes of the decision point. Decision points exist with `opt`, `alt`, `loop`, `break` or `neg`. E.g., the `opt` operator has one decision point at the begin of the combined fragment, with two possible outcomes: enter (numbered 1 in the figure) or skip (numbered 2 in the figure).

Table 9 Translation of loop, break and interaction use

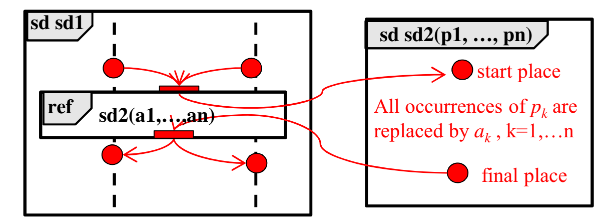
(m) loop (single decider)



(n) break (single decider)



(s) interaction use

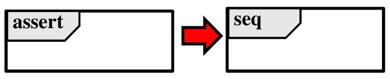


Notes:

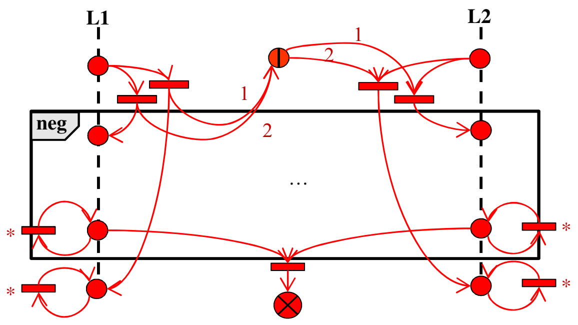
- m) This rule follows the same pattern introduced for the `opt` operator, with the difference that there are two decision points, at begin and end of the loop. In the absence of a guard condition g, the transition guards are reduced to the expressions underlined. An auxiliary LIFO place is used to store the loop counter, which is initialized when the loop is entered, is incremented when the loop continues to the next iteration, and is emptied when the loop is exited. The loop counter is not needed when min (`n`) and max (`m`) limits are not defined.
- n) A `break` fragment is similar to an `opt` fragment, except that at the end of the fragment execution skips to the place immediately after the end of the enclosing fragment. In case the enclosing fragment is a loop, the loop counter is emptied. In the absence of a guard condition g, the choice is made non-deterministically.

Table 10 Translation of conformance related operators

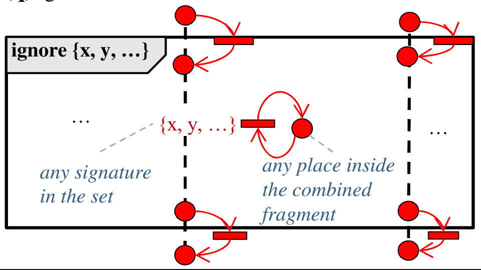
(o) assert



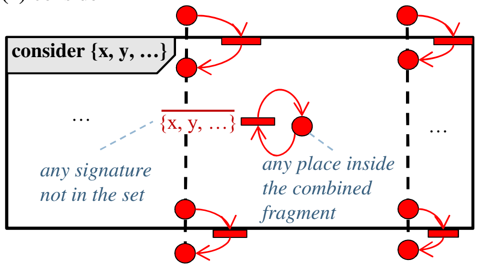
(p) negative (one decider)



(q) ignore



(r) consider


consider {x, y, …}
{x, y, …}
any signature not in the set
any place inside the combined fragment

Notes:

- o) The `assert` operator is basically ignored (i.e., it is handled as the neutral `seq` operator), because it is already assumed implicitly in our concept of test-ready SDs. In fact, traces that differ from the specified sequences are assumed invalid (and not inconclusive). In other words, the responses specified in the SD to the inputs from actors are implicitly interpreted as assertions of valid behavior.
- p) According to the UML standard ([1], pg. 484) , "the interactionOperator neg designates that the CombinedFragment represents traces that are defined to be invalid. (...) All InteractionFragments that are different from Negative are considered positive meaning that they describe traces that are valid and should be possible." Hence, we treat a negation as a non-deterministic alternative between the contents of the `neg` fragment (specifying an invalid trace prefix) and any trace (involving any lifelines), with precedence for the former. The precedence is enforced by our disambiguation policy (see sec. 5.3.3). The self-loops are redundant in case of loose conformance mode.
- q, r) The transitions indicated inside the combined fragment are redundant in the case of loose conformance, i.e., the `consider` and `ignore` fragments are basically ignored in loose conformance mode. In case of nested `consider` and `ignore` fragment, we assume that only the nearest enclosing fragment is relevant for each lifeline location.

References

1. OMG Unified Modeling Language^{TM} (OMG UML): Superstructure, Version 2.4.1. Object Management Group (OMG) (2011)
2. Mellor, S.J., Clark, A.N., Futagami, T.: Model-driven development. IEEE Softw. Mag. **20**(5), 1418 (2003)
3. Uttin, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, Burlington (2007)
4. Faria, J.P., Paiva, A.C.R., Yang, Z.: Test generation from UML sequence diagrams. In: 8th International Conference on the Quality of Information and Communications Technology, pp. 245–250 (2012)
5. JUnit testing framework: http://www.junit.org (2014)
6. Faria, J.P., Castro, M.V., Paiva, A.C.R.: Techniques and toolset for conformance testing against UML sequence diagrams. In: 25th IFIP International Conference on Testing Software and Systems (ICTSS'13), LNCS vol. 8254, pp. 180–195. Springer, Berlin, Heidelberg, New York (2013). http://link.springer.com/chapter/10.1007%2F978-3-642-41707-8_12#
7. Stotts, P.D., Pugh, W.: Parallel finite automata for modeling concurrent software systems. J. Syst. Softw. **27**, 27–43 (1994)
8. Micskei, Z., Waeselynck, H.: The many meanings of UML 2 sequence diagrams: a survey. J. Softw. Syst. Model. **10**, 489–514 (2011)
9. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. Int. J. Softw. Tools Technol. Trans. **9**, 213–254 (2007). (Springer)
10. Jorgensen, P.C.: Event-driven Petri nets. In: Modeling Software Behavior: A Craftsman's Approach, pp. 137–153. Auerbach Publications, Boca Raton (2009)
11. Faria, J.P., Paiva, A.C.R.: UML Checker: Formal Specification in VDM++ of the Petri Net-based Conformance Checking Engine, TR-SDBT-2013-04, FEUP (2013). https://blogs.fe.up.pt/sdbt/files/2013/04/TR-2013-04.pdf
12. Enterprise Architect. http://www.sparxsystems.com.au (2014)
13. AspectJ. http://www.eclipse.org/aspectj (2014)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Pearson, London (1994)
15. Hallal, H., Boroday, S., Petrenko, A., Ulrich, A.: A formal approach to property testing in causally consistent distributed traces. Formal Aspects Comput. **18**(1), 63–83 (2006)
16. Castro, M.V.: Automating Scenario Based Testing with UML and AOP. MSc thesis, FEUP (2013) https://blogs.fe.up.pt/sdbt/files/2013/04/Castro.pdf (in Portuguese)
17. JaCoCo Java Code Coverage Library. http://www.eclemma.org/jacoco/ (2014)
18. Google's Guava libraries. https://code.google.com/p/guava-libraries/ (2014)
19. Harel, D., Maoz, S.: Assert and negate revisited: modal semantics for UML sequence diagrams. J. Softw. Syst. Model. **7**(2), 237–253 (2008). (Springer)
20. Knapp, A., Wuttke, J.: Model checking of UML 2.0 interactions. Models Softw. Eng. LNCS **4364**, 42–51 (2007). (Springer)
21. Bouabana-Tebibel, T., Rubin, S.H.: An interleaving semantics for UML 2 interactions using Petri nets. Inform. Sci. **232**, 276–293 (2013)
22. Fernandes, J.M., Tjell, S., Jørgensen, J.B., Ribeiro, O.: Designing tool support for translating use cases and UML 2.0 sequence diagrams into coloured Petri net. In: SCESM'07, IEEE CS (2007)
23. Briand, L., Labiche, Y., Leduc, J.: Towards the reverse engineering of UML sequence diagrams for distributed java software. IEEE Trans. Soft. Eng. **32**(9), 642–663 (2006)
24. Kansomkeat, S., Offutt, J., Abdurazik, A., Baldini, A.: A comparative evaluation of tests generated from different UML diagrams. SNPD **2008**, 867–872 (2008)
25. Philip, S., Joseph, A.T.: Test sequence generation from UML sequence diagrams. SNPD **2008**, 879–887 (2008)
26. Samuel, P., Mall, R.: A novelt test case design technique using dynamic slicing of UML sequence diagrams. e-Informatica **2**(1), 71–92 (2008)
27. Nayak, A., Samanta, D.: Automatic test data synthesis using UML sequence diagrams. J. Object Technol. **9**(2), 115–144 (2010)
28. Benattou, M., Bruel, J., Hameurlain, N.: Generating test data from OCL specification. In: ECOOP Workshop Integration and Transformation of UML Models (2002)
29. Engels, G., Gldali, B., Lohmann, M.: Towards model-driven unit testing. In: Khne, T. (ed.) MoDELS 2006 Workshops, LNCS, vol. 4364, pp. 182–192 (2007)
30. Fraikin, F., Leonhardt, T.: SeDiTeC-testing based on sequence diagrams. In: Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002), IEEE (2002)
31. Wittevrongel, J., Maurer, F.: SCENTOR: scenario-based testing of E-business applications. In: 2nd International Workshop on Automation of Software Test (AST) (2007)
32. Javed, A., Strooper, P., Watson, G.: Automated generation of test cases using model-driven architecture. In: 2nd International Workshop on Automation of Software Test (AST) (2007)
33. IBM Rational Rhapsody: IBM Rational Rhapsody Automatic Test Conductor Add on User Guide, v2.5.2 (2013)
34. Boroday, S., Petrenko, A., Ulrich, A.: Implementing MSC tests with quiescence observation. In: TESTCOM/FATES 2009. LNCS, vol. 5826, pp. 49–65 (2009)
35. Ambler, S.: Agile Modeling (AM) Home Page—Effective Practices for Modeling and Documentation. http://www.agilemodeling.com (2014)