Test Generation from UML Sequence Diagrams

João Pascoal Faria  Ana Paiva  Zhuanli Yang  
INESC TEC and  Department of Informatics Engineering,  Ningbo University of Technology  
Faculty of Engineering,  Faculty of Engineering,  Porto, Portugal  
University of Porto  University of Porto  yzhl0104@163.com  
jpf@fe.up.pt  apaiva@fe.up.pt  

(This technical report is an extended version of our submission to QUATIC’2012)

Abstract—Model-driven engineering approaches aim at avoiding productivity, model quality and model maintenance problems that arise when models are used for documentation only, by generating executable applications from models. However, in many cases, the level of detail of the models needed to generate complete applications is too much or only effective for specific domains. For those cases where it is not practical to build complete models and generate complete applications from them, we propose a lightweight approach, applicable at different levels (unit, integration and system testing), that combines partial application generation from structural models with test generation from partial behavioral models. To demonstrate the approach, we developed a plug-in that adds to the code generation capabilities of an existing UML modeling tool, the capability of generating executable tests from sequence diagrams acting also as parameterized test scenarios, including some novel features as compared to existing model-based testing tools.

Keywords – test generation; UML; sequence diagrams

I. INTRODUCTION

Models are a key means to handle complexity through abstraction in engineering [1]. Recognizing that building models of software intensive systems for documentation only potentiates productivity, model quality and model maintenance problems, model-driven development approaches, such as the OMG’s MDA initiative [2][3], aim at generating executable applications from models. However, in many cases, the level of detail from the models needed to generate complete applications is too much or only effective for specific domains. For those cases where it is not practical to build complete models and generate complete applications from them, we propose a lightweight approach that combines partial application generation from structural models with test generation from partial behavioral models. To demonstrate the approach, we developed a plug-in that adds to the code generation capabilities of an existing UML modeling tool, the capability of generating executable tests from sequence diagrams acting also as parameterized test scenarios. Although there are already some tools for generating test cases from UML sequence diagrams (see section V), the proposed tool contributes with some novel features, namely the ability to: check the completeness and consistency of the models used for test generation; check internal interactions inside the application, besides external interactions with clients or users; loose conformance checking allowing incomplete specifications and implementations.

The rest of the report is organized as follows: section II presents the overall approach; section III illustrates the approach with an example; section IV details the most important features of behavior modeling and testing; section V presents additional case studies; section V presents related work; finally, section VI draws some conclusions and points out future work.

II. PROCESS

Our tools were developed to support a software development process combining model-driven development (i.e., code generation from models), model-based testing (i.e., test generation from models) and test-driven development (TDD) features, as depicted in Fig. 1. Without any loss of generality, we automated the proposed process for the Java programming language and the Enterprise Architect modeling tool [6]. The process involves the iterative application of the following steps:

1) Modeling. The application structure and behavior are modeled with UML class diagrams and (test-ready) sequence diagrams [4]. The preference for sequence diagrams comes from the following reasons: they capture the essence of object behavior (message exchange), as well as interactions with users and other systems; they are useful for iterative use-case driven development (a use case at a time, crossing several classes, instead of a class at a time); they provide a simple specification completeness criteria (all classes and methods are exercised); they can serve also as test specifications. Overall, instead of developing full heavy-weight behavior specifications, our approach is to develop lightweight (partial and executable) behavior specifications, which serve also as test specifications.

2) Modeling. The application structure and behavior are modeled with UML class diagrams and (test-ready) sequence diagrams [4]. The preference for sequence diagrams comes from the following reasons: they capture the essence of object behavior (message exchange), as well as interactions with users and other systems; they are useful for iterative use-case driven development (a use case at a time, crossing several classes, instead of a class at a time); they provide a simple specification completeness criteria (all classes and methods are exercised); they can serve also as test specifications. Overall, instead of developing full heavy-weight behavior specifications, our approach is to develop lightweight (partial and executable) behavior specifications, which serve also as test specifications.
section IV. The visualization of problems detected is illustrated in section III.

3) Code generation. Production code in the target programming language (essentially class skeletons) can be generated from the structural model, using the code engineering features of the modeling tool. From the behavioral model (i.e., the sequence diagrams), executable test code in a unit testing framework [7] is generated automatically by our TestGenerator add-in for Enterprise Architect. The generated code checks not only if the application reacts as specified to external clients and users, but also if internal interactions in the application occur as specified. Aspect-oriented programming (AOP) is used to trace the internal interactions. To promote reuse and simplify the generated test code, we provide the needed AOP code in a reusable tracing library in AspectJ [8].

4) Test execution (before production code completion). As advocated in test-driven development [9], the added test cases should be executed and observed to fail, before writing the corresponding production code, to assure their usefulness.

5) Production code completion. Starting from the newly generated production code and any previously existing code base, the developer should write the needed production code (usually method bodies) in order to pass in the tests.

<table>
<thead>
<tr>
<th>Manual activities</th>
<th>Automated activities</th>
<th>Artifacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Model application structure &amp; behavior</td>
<td></td>
<td>Code generation</td>
</tr>
<tr>
<td>2. Check model consistency &amp; completeness (UMLChecker)</td>
<td></td>
<td>Test code</td>
</tr>
<tr>
<td>3. Generate code from structural model</td>
<td></td>
<td>Production code</td>
</tr>
<tr>
<td>4. Execute tests &amp; see them failing</td>
<td></td>
<td>Test code (JUnit3)</td>
</tr>
<tr>
<td>5. Complete production code (method bodies)</td>
<td></td>
<td>Usable libraries</td>
</tr>
<tr>
<td>6. Execute tests &amp; see them passing</td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

![Figure 1](image1.png)

Figure 1 Overview of the proposed process, depicted as a UML activity diagram. Contributions of this report are highlighted. The technologies shown are those currently supported by the developed tools.

6) Test execution (after production code completion). All test cases are executed again, this time with the expectation of getting a green bar, and any problems found are fixed in the production code or in previous steps.

III. EXAMPLE

We illustrate the most distinctive steps of the approach presented for an example application – a spreadsheet engine.

1a) Structural Modeling. Fig. 2 presents the class structure of a simple spreadsheet engine, supporting the creation of spreadsheets (Spreadsheet), creation of named cells (Cell) inside spreadsheets, assignment of values (setValue) or formulas (setFormula) to cells, and querying cell values (getValue), possibly calculated as a function of other cells. A parser (Parser) allows converting textual expressions to an in-memory tree representation (Formula) following the Composite design pattern. Expressions can only contain numerical constants (Const), cell references (CellRef) and binary operators (BinOp). Besides the API, the spreadsheet engine also provides a simple command line interface (SpreadsheetCLI). An exception class (CircularReferenceException) is also defined, to be instantiated on any attempt to calculate a cell as a function of itself. The Parser class is currently marked with the «stub» stereotype, meaning that it is not yet implemented.

![Figure 2](image2.png)

Figure 2 Class diagram of the spreadsheet engine. Features not covered by the sequence diagrams are automatically marked with a stereotype.

1b) Behavioral Modeling. The intended behavior of the spreadsheet engine, namely internal interactions among objects

...
in the application and external interactions with client applications or users, is partially specified in the sequence diagrams of Fig. 3 and Fig. 4. Fig. 3 illustrates how the spreadsheet engine API can be used by a client application to define and query calculated cells. It also illustrates the internal workings, namely the conversion of textual expressions to in-memory tree representation, and the choice for lazy evaluation (on demand) of calculated cells, instead of eager evaluation (on update). Combined fragments with the par operator [4] indicate that the implementation has the freedom to process the left-hand side (lhs) and right-hand side (rhs) of binary expressions in parallel (or by any order), both during the parsing and evaluation phases. For increased generality, the sequence diagram is defined in terms of a set of parameters, whose types and example values are defined in a stereotyped note by the tester. The sequence diagram in Fig. 4 illustrates how the spreadsheet engine can be used by a user through the command line interface, and how the class that handles user interaction (SpreadsheetCLI) interacts with the rest of the application (omitting further details of internal working already described in other diagram). Simple keywords (start, enter, display) are used o model user interaction.

2) Model verification. The execution of our UMLChecker plug-in reveals and marks some inconsistencies and incompleteness in the model, as shown by the messages in red in Fig. 3 and Fig. 4, and the stereotype «NotCovered» in Fig. 2, respectively. Further information is provided in notes on the affected elements (not visible in diagrams). In order to achieve 100% coverage (of classes and methods), besides fixing inconsistencies, one should add new scenarios or extend the existing ones to illustrate circular references and the conversion from the in-memory to the textual expression representation.

3b) Test code generation. The execution of our TestGenerator plug-in produces the test class partially shown in Fig. 5, which has to be linked with our trace library (traceutils.jar) and enabled for AspectJ before test execution with JUnit3. The generated class extends InteracTestCase, which is based on the TestCase class from JUnit3. A test method was generated for each sequence diagram. In the case of the parameterized sequence diagram, additional test methods were generated for the given parameter values. For the first sequence diagram, the interactions with the client application are encoded straightforwardly, using assertEquals to check return values. Code generated to check internal interactions is shown only in one case, for space limitation reasons. In general, this code follows the steps: declare special object handlers for objects to be created internally, if needed; declare the expected call tree; perform the intended action; perform a final conformance check between the actual and expected call trees. For the second sequence diagram, the interactions with the user are encoded straightforwardly based on an auxiliary test class (Console), using assertEquals to check values displayed to the user.

6) Test execution. Finally, Fig. 6 shows the result of test execution for an implementation with a bug in the divide operator. The failure is detected in an internal interaction inside the system, hence helping fault localization and bug fixing.

Figure 3 Sequence diagram illustrating the behavior of the spreadsheet engine when accessed by a client application through the API. Inconsistencies with the class diagrams are automatically marked red.
public class SpreadsheetTest extends InteracTestCase {
    public void testSpreadSheetAPI(double xVal, char op, double yVal, double zVal, String yExpr) { 
        Spreadsheet s = new Spreadsheet("s");
        Cell x = s.addCell("x");
        Cell y = s.addCell("y");
        x.setValue(xVal);
        ObjHandler<CellRef> r = new ObjHandler<CellRef>();
        ObjHandler<BinOp> f = new ObjHandler<BinOp>();
        Trace.expect( 
            new Call("setFormula", "x", x, null, y, yExpr, null, f, new CombiPar( 
                new CombStrict{ 
                    new Call("getValByRef", s, "x", x, r), 
                    new Constr("Const", c, "Const", cVal), 
                    new Constr("Funct", f, args(op, r, c))) 
            ), yVal, y.getValue());
    }
    public void testSpreadSheetAPI_0() { 
        testSpreadSheetAPI(1.0, '+', 2.0, 3.0, "x + 2");
    } 
    public void testSpreadSheetCLI() { 
        SpreadsheetCLI s = SpreadsheetCLI.class, null); 
        Console.enter("x = 1");
        assertEq(2.0, Console.check());
        Console.enter("x");
        Console.stop();
    } 
}

Figure 4 Sequence diagram illustrating the behavior of the spreadsheet engine when accessed by a user through the command line interface. Inconsistencies with the class diagrams are automatically marked red.

// package and import declarations omitted

Figure 5 Excerpt of generated test class for the spreadsheet example. Some abbreviations were used and package prefixes were omitted to save space.

IV. BEHAVIOR MODELING AND TESTING FEATURES

A. Checking model consistency and completeness

One of our goals was to have a precise (and automatically measurable) but lightweight criteria for behavioral model completeness, and consequently (in our case), test coverage. A simple criterion that is currently checked by our UMLChecker plug-in is that all methods and constructors defined in the structural model are exercised in the behavioral model (including overloaded and overridden methods whenever distinguishable). The tool provides visual feedback of the features not covered (see Fig. 3). The tool also checks and depicts visually (see Fig. 4 and Fig. 5) inconsistencies in the behavioral model with respect to the structural (unknown methods, wrong number of parameters, wrong parameter types, wrong return types, accessibility violations, etc.).

B. Parameterized scenarios

Our test-ready sequence diagrams are standard UML sequence diagrams, with simple restrictions and conventions, namely regarding parameterization and actor specification.

Parameterization is of paramount importance for building more generic scenarios. Currently, we support the definition of a set of parameters (with name and type) in each sequence diagram, with example values that can be used for data-driven testing as illustrated in the example. Parameters have the scope of the sequence diagram and can be used anywhere.

Mandatory actor lifelines are used to represent external users or client applications that initiate interactions with the application under test, and become test drivers in the test harness. Currently, we distinguish between the two types of actors based on name conventions.

C. Loose conformance checking of internal interactions

Besides checking external interactions with the user or client application (simulated by the test driver), the test harness also checks if the interactions inside the application occur as specified in the sequence diagram. To that end, the tracing library monitors method and constructor calls and return values through before/after advice in AspectJ, and incrementally checks them against the expectations specified in the sequence diagram.
Diagram (and encoded in the generated test code), looking for possible violations (actual parameter values or return values different from expected, missing calls, unexpected calls, etc.). One of our goals is to allow the construction of lightweight behavior models, in which only the internal interactions considered of interest by the modeler are specified. To that end, we put forward a notion of loose conformance as illustrated in Fig. 7. It means that an implementation is according to a specification if the specification tree is a sub tree of the implementation.

![Figure 7 Illustration of the loose conformance checking concept.](image)

Formally, in strict conformance, the conformance predicate would be:

\[
s\text{ConformsTo}(\text{actualCallTree}: \text{Tree}<\text{Call}>, \text{expectedCallTree}: \text{Tree}<\text{Call}>) =
\begin{align*}
& (\text{isEmpty}(\text{actualCallTree}) \land \text{isEmpty}(\text{expectedCallTree})) \\
& \lor \\
& (\neg \text{isEmpty}(\text{actualCallTree}) \land \neg \text{isEmpty}(\text{expectedCallTree}) \\
& \land \text{matches}(\text{root}(\text{actualCallTree}), \text{root}(\text{expectedCallTree})) \\
& \land s\text{ConformsTo}(\text{child}(\text{actualCallTree}), \text{child}(\text{expectedCallTree}))
\end{align*}
\]

which uses the auxiliary predicate:

\[
s\text{ConformsTo}(\text{actualChild}: \text{Seq}<\text{Tree}<\text{Call}>>, \text{expectedChild}: \text{Seq}<\text{Tree}<\text{Call}>>) =
\begin{align*}
& (\text{isEmpty}(\text{actualChild}) \land \text{isEmpty}(\text{expectedChild})) \\
& \lor \\
& (\neg \text{isEmpty}(\text{actualChild}) \land \neg \text{isEmpty}(\text{expectedChild}) \\
& \land \text{matches}(\text{root}(\text{actualChild}), \text{root}(\text{expectedChild})) \\
& \land s\text{ConformsTo}(\text{child}(\text{actualChild}), \text{child}(\text{expectedChild})) \\
& \land s\text{ConformsTo}(\text{tail}(\text{actualChild}), \text{tail}(\text{expectedChild}))
\end{align*}
\]

By contrast, in loose conformance we have:

\[
l\text{ConformsTo}(\text{actualCallTree}: \text{Tree}<\text{Call}>, \text{expectedCallTree}: \text{Tree}<\text{Call}>) =
\begin{align*}
& (\neg \text{isEmpty}(\text{expectedCallTree}) \Rightarrow \\
& (\neg \text{isEmpty}(\text{actualCallTree}) \\
& \land \text{matches}(\text{root}(\text{actualCallTree}), \text{root}(\text{expectedCallTree})) \\
& \land l\text{ConformsTo}(\text{child}(\text{actualCallTree}), \text{child}(\text{expectedCallTree})))
\end{align*}
\]

which uses the auxiliary predicate:

\[
l\text{ConformsTo}(\text{actualChild}: \text{Seq}<\text{Tree}<\text{Call}>>, \text{expectedChild}: \text{Seq}<\text{Tree}<\text{Call}>>) =
\begin{align*}
& (\neg \text{isEmpty}(\text{expectedChild}) \Rightarrow \\
& (\neg \text{isEmpty}(\text{actualChild}) \\
& \land \text{matches}(\text{root}(\text{actualChild}), \text{root}(\text{expectedChild})) \\
& \land l\text{ConformsTo}(\text{child}(\text{actualChild}), \text{child}(\text{expectedCallTree})))
\end{align*}
\]

To allow a more efficient implementation and because of the need to handle stubs (see subsection F), we take an irrevocable matching approach (\(\neg\text{matches}\) clause above).

Unknown values in method parameters, denoted “?,” are also allowed. They are taken into account in the “matches” predicate.

Regarding objects created internally, whenever the same object (as identified by its lifetime and name) appears multiple times in a sequence diagram, it is checked if the same happens in the implementation. This is implemented through typed object handlers (see Fig. 5), that capture the object reference on the first occurrence (constructor call) and use that reference in subsequent checks.

D. User interaction modeling and testing

The UML does not prescribe a unique and precise means for modeling user interaction, so we adopted a set of keywords similar to the ones used in user interface testing frameworks such as FIT [10]. Currently, we have only automated the testing of user interaction through the console, so the keywords supported are (see also the examples in Fig. 4): start – the user starts the (main method of the) designated class; enter – the user enters the value specified through the standard input; display – the application displays the value specified to the standard output.

The test library automatically starts the designated class in a thread separate from the test thread, and uses around advices in AspectJ to intercept any input/output calls issued by the application (on System.in and System.out) and replace them by buffered communication with the test thread, which feeds input values as specified in enter messages and checks output values against the specified display messages.

E. Combined fragments

Combined fragments in sequence diagrams (see [4] and example in Fig. 3) are very useful to specify more generic scenarios involving control flow variants (with the loop, opt, alt, par operators), and to promote reuse (with the ref operator). Besides the standard UML operators, we also support other operators found in ConcurTaskTrees notation [11], namely the orderIndependent combinator (which we name perm, for permutation). One of ours goals is to allow partial behavior specifications, reason why the iteration or decision expressions in the combined fragments may be omitted or underspecified.
During test execution and conformance checking, combined fragments are handled differently depending on the source lifeline of the initiating message: if the source lifeline is an actor (user or client application, which becomes a test driver in test execution) or a stub (control perspective), any under-specification is processed with randomization; otherwise (observation perspective), under-specification is processed accordingly by our loose conformance checking engine (i.e., any actual execution trace compatible with the specification is allowed).

F. Testing incomplete implementations (stub in the middle)

Classes with methods or constructors not yet implemented (i.e., with empty or dummy bodies), or which implementation one does not want to use, are marked with the «stub» stereotype, as illustrated for class Parser in the example. The test library automatically generates the behavior specified in the sequence diagrams (and encoded in the generated test code) in response to method or constructor calls on that class. The calls are intercepted with an around advice in AspectJ, which executes any specified outgoing calls through reflection (including private calls and calls to other stubs) and returns the value specified (if any), instead of executing the called method (but constructors still have to be executed). This feature allows implementing and testing complex scenarios iteratively.

V. ADDITIONAL CASE STUDIES

A. Student’s t-distribution

In this example we consider a program for calculating the probability of the Student’s t-distribution. Through a command line interface, the program asks the value of t and degrees of freedom of the distribution, and prints the corresponding total probability between 0 and t.

Fig. 8 shows the class structure of the program. A numerics package provides numerical integration by the Simpson method (SimpsonNumericalIntegration) of any function (FunctionObject). A statistics package provides definitions of continuous probability distributions (ProbabilityDistribution, ProbabilityDensityFunction) and Student’s t-distribution (TStudentDistribution). User interface through the command line is handled by a separate class (TStudentDistributionCLI).

Fig. 9 illustrates the intended program behavior. It illustrated not only user interaction, but also internal interactions between objects in the program.

Figure 8 Class diagram of the Student’s t-distribution program.

Figure 9 Sequence diagram illustrating the behavior of the Student’s t-distribution program.
This example illustrates some modeling and test generation features not illustrated by the spreadsheet engine example:

- Usage of the loop combined fragment;
- Usage of the "..." symbol to denote unknown method parameter values;
- Usage of the "*" symbol to denote message repetition;
- Usage of the strict conformance option (instead of the default loose conformance) — this means that method calls not shown in the diagram are not expected;
- Usage of an alternative user interface modeling style.

B. Observer pattern

This example illustrates the application of the Observer design pattern (also known as Publish-Subscribe).

![Class diagram of the application of the Observer pattern.](image)

![Sequence diagram illustrating the behavior of the Observer pattern.](image)

Figure 10 Class diagram of the application of the Observer pattern.

Figure 11 Sequence diagram illustrating the behavior of the Observer pattern.

VI. RELATED WORK

UML is a de facto widespread modeling notation in industry. The high benefit of using such models is when they are effectively integrated in the software development process, avoiding the “just for documenting purposes”. There are several research works that try to use UML models for generating code or for generating tests to check if the software application is as expected. Considering test case generation, some important features of such approaches are the generation of test sequences, test data and test code.

The most common UML diagrams used for test case generation are statecharts [12][13][14]. However, there are also several works that extract test cases from behavioral models such as sequence diagrams, activity diagrams or from different sets of UML diagrams, such as use cases and sequence diagrams, or sequence and activity diagrams, or sequence diagrams as state diagrams [15][16][17][18][19][20].

Sequence diagrams are a kind of interaction diagram consisting of a set of objects and their relationships, including the messages that may be dispatched among them. Sequence diagrams address the dynamic view of a system emphasizing the time-ordering of messages [21]. As such, test cases generated from sequence diagrams may be adequate to find errors concerning the sequence of executed messages and check the returned values of such messages [22].

With the novel features of UML 2.0, sequence diagrams may combine interaction fragments which allow the description of a number of traces in a compact and concise manner. Examples of such combinations are alt, loop, opt, etc. Because of that, there are several examples in the literature that use an intermediate notation to represent the set of executions within a sequence diagram and afterwards, test cases are generated from this representation according to coverage criteria. Some examples of such representations are sequence dependency graphs [18], message dependency graph (MDG) [19], and structured composite graph (SCG) [17].

Besides generating test sequences, there are some approaches that also generate automatically test data. Nayak et al. [17] enrich sequence diagrams with attribute and constraint information derived from class diagrams and OCL (Object Constraint Language).
Constraint Language) constraints and use a constraint solving system to generate test data to cover paths along scenarios. Samuel et al. [19] create dynamic slice according to conditional predicates associated with messages in a sequence diagram and generate test data satisfying each slice. Benauttou et al. [27] generate test data automating partition analysis of individual method’s class by reducing the mathematical expressions which defines a method (operation) into Disjunctive Normal Form (DNF).

Another important feature is the generation of test code at the end. However, just some approaches are capable of doing so. The work of [23] and the SeDiTeC tool [24] are some of such examples.

Javed et al. [23] generate JUnit and SUnit. Test code is generated by reading XML data files. Although the test code generation is automatic, it must be performed for every XML data file because data is hardcoded within the generated code. To check if the returning values of messages are correct, they call additional methods to construct objects using parameters passed through the messages and afterwards compare the results obtained from the implementation with such previously constructed objects. This approach may not work when the implementation of the objects’ construction has additional behavior (for instance, side effects) that is not considered by the additional methods. The execution trace information gathering is not integrated into the approach and they do not automate their verification. It has to be performed by hand. In addition, they do not deal with the novel features of UML 2.0.

SeDiTeC [24] is another example of an approach generating test code. One advantage of such approach is the generation of stubs for parts of the system that are not implemented hence allowing starting testing activities earlier. However, they limit the kind of sequence diagrams that can be used as input for the approach describing the characteristics that they should have in order to be testable. As far as we know, they do not deal directly with the novel features of UML 2.0. However, they combine different sequence diagrams which can be used as a way to represent, for instance, alternative blocks of messages. Test data is kept separated with parameters and return values for the method calls.

Our approach is capable of generating test sequences and test code and solves some of the problems of the existing approaches. It imposes minimal restrictions to the input UML sequence diagrams. The generated test code is parameterized which does not require to generate test code for different test data; it only requires calling the test procedure with different parameters. The execution is integrated, which allows checking automatically the sequence of method calls. Besides this, our approach also builds implementation stubs which allow starting the testing activity earlier and has an additional functionality to verify if the sequence diagram covers all the methods defined within the related class diagram.

VII. CONCLUSIONS AND FUTURE WORK

We presented an approach with tool support for lightweight partial behavior specification through parameterized UML sequence diagrams that act also as executable test specifications. The approach was successfully experimented on a set of case studies. The process is explained and illustrated through an example.

Among other benefits, the approach allows checking the completeness and consistency of the models used for test generation (section IV, A); checking internal interactions inside the application, besides external interactions with clients or users, which improves fault localization; loose conformance checking allowing incomplete specifications and implementations (section IV, C).

As future work, we plan to: present test results in the model itself (by coloring faulty messages), and allow running the tests directly from the modeling tool (hiding test generation details); extend user interaction modeling and testing features for graphical user interfaces (GUI), taking advantage of our previous work on GUI test automation [25]; automatically generate test data (i.e., actual values for scenario parameters) through constraint satisfaction; conduct an experiment, using the PSP [26] measurement framework, to assess the productivity, quality and predictability of with our approach compared to others; extend the test engine support to extend the test execution engine to support the testing of time constrained, concurrent and distributed systems, particularly in the scope of integration testing.

REFERENCE


[27] Benattou, M., J.-M. Bruel, and N. Hameurlain, Generating Test Data from OCL Specification, 2002