



Institute for Software Research
University of California, Irvine

Scenario-based and State Machine-based Testing: An Evaluation of Automated Approaches



Leila Naslavsky
University of California, Irvine
awinblad@ics.uci.edu



Debra J. Richardson
University of California, Irvine
djr@ics.uci.edu



Hadar Ziv
University of California, Irvine
ziv@ics.uci.edu

August 2006

ISR Technical Report # UCI-ISR-06-13

Institute for Software Research
ICS2 110
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

www.isr.uci.edu/tech-reports.html

Scenario-based and State Machine-based Testing: An Evaluation of Automated Approaches

Leila Naslavsky, Debra J. Richardson, and Hadar Ziv
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
{lnaslvs, djr, ziv} @ics.uci.edu

ISR Technical Report # UCI-ISR-06-13

August 2006

Abstract: Software testing is regarded by practitioners as the central activity used for ensuring that a system behaves as expected. It consists of the following activities: test generation, execution, evaluation, coverage analysis and regression testing. Testing is often a neglected activity that has the potential and the need for benefiting from automation. In fact, testing solutions currently available automate some testing activities. However, the majority of the automated solutions are code-based. Code-based testing uses the implementation to derive test cases. It can be used to test the system. It cannot, however, be used to test that the system built is what was intended. Testing activities that compare the system's implementation to their expected behavior are more informative because they show the system satisfies its original requirements. This kind of activity is characteristic of specification-based testing.

In practice, specification-based testing has not yet achieved the level of adoption achieved by code-based testing. This can be explained in part by the lack of support for its activities. These activities require support for managing relationships between specifications and other artifacts. Relationships include mapping from specification to implementation, and traces from requirements to test results. The mapping supports generation and/or execution of test scripts, while the traces from requirements to test results supports requirements coverage analysis.

This survey studies automated specification-based testing from two perspectives. The first perspective evaluates the level of automation, which is done by evaluating the support for each testing activity. The second perspective aims at better understanding what kinds of artifacts and relationships are used by the approaches, and how they manage these relationships.

Scenario-based and State Machine-based Testing: An Evaluation of Automated Approaches

Leila Naslavsky, Debra J. Richardson, and Hadar Ziv
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
{lnaslvs, djr, ziv} @ics.uci.edu

ISR Technical Report # UCI-ISR-06-13

August 2006

Abstract

Software testing is regarded by practitioners as the central activity used for ensuring that a system behaves as expected. It consists of the following activities: test generation, execution, evaluation, coverage analysis and regression testing. Testing is often a neglected activity that has the potential and the need for benefiting from automation. In fact, testing solutions currently available automate some testing activities. However, the majority of the automated solutions are code-based. Code-based testing uses the implementation to derive test cases. It can be used to test the system. It cannot, however, be used to test that the system built is what was intended. Testing activities that compare the system's implementation to their expected behavior are more informative because they show the system satisfies its original requirements. This kind of activity is characteristic of specification-based testing.

In practice, specification-based testing has not yet achieved the level of adoption achieved by code-based testing. This can be explained in part by the lack of support for its activities. These activities require support for managing relationships between specifications and other artifacts. Relationships include mapping from specification to implementation, and traces from requirements to test results. The mapping supports generation and/or execution of test scripts, while the traces from requirements to test results supports requirements coverage analysis.

This survey studies automated specification-based testing from two perspectives. The first perspective evaluates the level of automation, which is done by evaluating the support for each testing activity. The second perspective aims at better understanding what kinds of artifacts and relationships are used by the approaches, and how they manage these relationships.

1	INTRODUCTION.....	4
1.1	SCOPE	5
1.2	STRUCTURE OF THE SURVEY	5
2	AUTOMATED SCENARIO-BASED AND STATE MACHINE-BASED TESTING.....	7
2.1	TEST GENERATION	7
2.2	EXECUTION AND EVALUATION	8
2.3	COVERAGE ANALYSIS	9
2.4	REGRESSION TESTING.....	9
3	BACKGROUND DISCUSSION.....	11
3.1	SCENARIOS AT DIFFERENT LEVELS OF ABSTRACTION.....	11
3.2	RELATIONSHIP MANAGEMENT AND SPECIFICATION-BASED TESTING.....	12
4	EVALUATION FRAMEWORK.....	14
4.1	TEST GENERATION	14
4.2	EXECUTION AND EVALUATION	15
4.3	COVERAGE ANALYSIS	15
4.4	REGRESSION TESTING.....	15
4.5	GENERAL ASPECTS	16
5	APPLYING THE EVALUATION FRAMEWORK TO SCENARIO-BASED AND STATE-BASED APPROACHES	17
5.1	SEQUENCE DIAGRAM TEST CENTER (SeDiTEC).....	17
5.1.1	<i>Application of the Evaluation Framework.....</i>	19
5.2	SCENTOR.....	21
5.2.1	<i>Application of the Evaluation Framework.....</i>	22
5.3	COWTEST PLUS UIT ENVIRONMENT (COW_SUITE).....	24
5.3.1	<i>Application of the Evaluation Framework.....</i>	26
5.4	AUTOMATED GENERATION AND EXECUTION OF TEST SUITES FOR DISTRIBUTED COMPONENT-BASED SOFTWARE (AGEDIS).....	29
5.4.1	<i>Application of the Evaluation Framework.....</i>	30
5.5	SCENARIO-BASED OBJECT-ORIENTED TESTING FRAMEWORK (SOOTF).....	33
5.5.1	<i>Application of the Evaluation Framework.....</i>	35
5.6	UMLTEST	38
5.6.1	<i>Application of the Evaluation Framework.....</i>	40
5.7	ABSTRACT STATE MACHINE LANGUAGE (ASML) TEST TOOL.....	42
5.7.1	<i>Application of the Evaluation Framework.....</i>	44
5.8	TESTING OBJECT-ORIENTED SYSTEMS (TOTEM)	46
5.8.1	<i>Application of the Evaluation Framework.....</i>	48
5.9	UMLAUT/SIMULATOR	50
5.9.1	<i>Application of the Evaluation Framework.....</i>	51
5.10	TEST SEQUENCE GENERATOR (TESTOR)	53
5.10.1	<i>Application of the Evaluation Framework.....</i>	54
5.11	UC-SCSYSTEM.....	56
5.11.1	<i>Application of the Evaluation Framework.....</i>	57
5.12	SUMMARY	59
5.12.1	<i>Test Generation.....</i>	59
5.12.2	<i>Test Execution and Evaluation</i>	62
5.12.3	<i>Coverage Analysis and Regression Testing.....</i>	65
6	OTHER RELATED WORK.....	66
6.1	SCENT-METHOD	66
6.2	ENGINEERED USE CASES.....	66
6.3	TAOS.....	66

6.4	MODEL-DRIVEN ARCHITECTURE AND TESTING	67
7	CONCLUSIONS AND RESEARCH RECOMMENDATIONS	69
8	GLOSSARY	71
	REFERENCES	73

LIST OF FIGURES

FIGURE 1 - AUTOMATED SCENARIO-BASED AND STATE MACHINE-BASED TESTING	7
FIGURE 2 - TECHNICAL SPECIFICATION-BASED TESTING ACTIVITIES SUPPORTED BY SeDiTeC	19
FIGURE 3 - TECHNICAL SPECIFICATION-BASED TESTING ACTIVITIES SUPPORTED BY SCENTOR	22
FIGURE 4 - TECHNICAL SPECIFICATION-BASED TESTING ACTIVITIES SUPPORTED BY COW_SUITE	26
FIGURE 5 - TECHNICAL SPECIFICATION-BASED TESTING ACTIVITIES SUPPORTED BY AGEDIS	30
FIGURE 6 - TECHNICAL SPECIFICATION-BASED TESTING ACTIVITIES SUPPORTED BY SOOTF	35
FIGURE 7 - TECHNICAL SPECIFICATION-BASED TESTING ACTIVITIES SUPPORTED BY UMLTest.	40
FIGURE 8 - TECHNICAL SPECIFICATION-BASED TESTING ACTIVITIES SUPPORTED BY ASML	44
FIGURE 9 - TECHNICAL SPECIFICATION-BASED TESTING ACTIVITIES SUPPORTED BY TOTEM	48
FIGURE 10 - TECHNICAL SPECIFICATION-BASED TESTING ACTIVITIES SUPPORTED BY UMLAUT/SIMULATOR	51
FIGURE 11 - TECHNICAL SPECIFICATION-BASED TESTING ACTIVITIES SUPPORTED BY TESTOR	54
FIGURE 12 - TECHNICAL SPECIFICATION-BASED TESTING ACTIVITIES SUPPORTED BY UC-SCSYSTEM	57

LIST OF TABLES

TABLE 1 - EVALUATION FRAMEWORK APPLIED TO SeDiTeC	20
TABLE 2 - EVALUATION FRAMEWORK APPLIED TO SCENTOR	23
TABLE 3 - EVALUATION FRAMEWORK APPLIED TO COW_SUITE.	28
TABLE 4 - EVALUATION FRAMEWORK APPLIED TO AGEDIS	32
TABLE 5 - EVALUATION FRAMEWORK APPLIED TO SOOTF	37
TABLE 6 - EVALUATION FRAMEWORK APPLIED TO UMLTest.	41
TABLE 7 - EVALUATION FRAMEWORK APPLIED TO ASML	45
TABLE 8 - EVALUATION FRAMEWORK APPLIED TO TOTEM	49
TABLE 9 - EVALUATION FRAMEWORK APPLIED TO UMLAUT/SIMULATOR	52
TABLE 10 - EVALUATION FRAMEWORK APPLIED TO TESTOR	55
TABLE 11 - EVALUATION FRAMEWORK APPLIED TO UC-SCSYSTEM	58
TABLE 12 – TEST GENERATION	59
TABLE 13 - TEST GENERATION	60
TABLE 14 – TEST GENERATION	61
TABLE 15 – TEST EXECUTION AND EVALUATION	62
TABLE 16 - TEST EXECUTION AND EVALUATION	63
TABLE 17 - TEST EXECUTION AND EVALUATION	64
TABLE 18 - COVERAGE ANALYSIS AND REGRESSION TESTING	65

1 Introduction

Software testing is still regarded by practitioners as the central activity used for ensuring that a system behaves as expected. It consists of the following activities: test case generation, execution, result evaluation, coverage analysis, and regression testing [61]. In traditional software development processes, testing activities are left to the end of the software life cycle, so schedule slippage, time-to-market pressures, and cost-constraints result in neglected testing. It is clear that testing has the potential and the need for benefiting from automation. Automation can reduce the amount of effort spent on technical testing activities. It can also increase the precision of activities like result evaluation, often performed by humans and thus more error-prone. There are in fact, solutions currently available in the industry [58] and in the academia [22, 77] that automate some testing activities and reduce testers efforts. However, the majority of automated solutions are code-based. Code-based testing uses the implementation to derive test cases. Thus, it can be used to test the system, but it cannot be used to test the original expectations about the system.

Testing activities that compare the system's implementation to their expected behavior are more informative because they show the system satisfies its original requirements. This can be achieved by deriving test cases from high-level specifications, with specification-based testing. Non-technical stakeholders are usually largely involved with creating these expectations. Therefore, testing approaches based on specifications that non-technical stakeholders find easier to manipulate could be even more informative.

In practice, however, specification-based testing has not yet reached the level of adoption achieved by code-based testing. This can be explained in part by the lack of support for its activities, which require management of relationships between specifications and other artifacts. Examples of such relationships are relationships between the specifications and the implementation (sometimes called mapping), or between the specifications and the test results (sometimes called traces). The first supports generation of concrete test scripts or execution of abstract test scripts, while the second supports coverage analysis based on the specifications. Relationship management is a topic currently addressed by research on software traceability. It aims at automation or semi-automation of creation, maintenance and storage of meaningful relationships across software artifacts [5, 7, 21, 46, 38, 69] as well as on describing and classifying those relationships [9, 69, 45].

Software traceability tools automate some software development activities by connecting all inter-relatable software artifacts and supporting relationship management. They exploit the fact that software artifacts have different forms, objectives and semantics; exist at different levels of abstractions and are usually closely inter-related through different kinds of relationships [56, 67, 66]. These *relationships* can be represented by explicit links, references, and similar name, among other representations [45]. Nevertheless, for those kinds of solutions to be practical, many challenges investigated by the traceability research need to be addressed first. Some challenges exist because stakeholders do not usually maintain (and sometimes not even establish) the relationship representation across the artifacts. As a result, even if the relationships exist, they might become obsolete [69, 25]. Establishing and maintaining relationships among software artifacts is important because relationships can be used in a number of different software engineering activities such as software change impact analysis and software validation, verification and testing [69]. Current traceability approaches that explore validation and verification concentrate on solving inconsistencies among requirements artifacts, design artifacts and code [23, 27, 43, 68]. The use of such approaches to improve automation of specification-based software testing is yet to be fully explored.

To summarize, one way to address current testing problems is by reducing effort spent on testing activities, while concentrating on using expectations about the systems to test them, as opposed to relying only on the code to derive the test cases. Therefore, since specification is a type of expectation, the use of automated specification-based testing should improve testing.

Unfortunately, these approaches are not fully automated and face challenges related to management of relationships across artifacts.

This survey studies approaches to automated specification-based testing and suggests future research directions to increase their automation and their likelihood of being adopted in practice. It explores these approaches from two perspectives. The first perspective evaluates their level of automation, which is done by evaluating the level of automation of each testing activity. The second perspective aims at better understanding what kinds of artifacts and relationships among them are produced and used by the approaches, and how the approaches manage these relationships.

1.1 Scope

This survey evaluates a representative subset of currently available automated scenario-based and state-based testing approaches. This section explains the reasons for including approaches based on these types of specifications.

Both scenario-based and state-based specification languages have emerged as important modeling perspectives. In particular with the adoption of UML tools by practitioners. Extending automated support for testing systems described with scenarios and state-based specifications can leverage the effort put into creating them. The availability of more automated testing solutions can stimulate the use of such specifications. Therefore, a reason for evaluating approaches based on these two types of specifications was to understand how they could be more automated.

Inline with the previous reason, approaches evaluated should automate the larger number of testing technical activities. A fair amount of academic research explores state-based specification testing. These approaches automate test case generation and execution with a different variety of tools. Tools such as AutoFocus accept as input specification described by proprietary languages. Other tools accept as input publicly available specification languages such as SDL¹ or LTS. TestComposer and AutoLink receive specifications in SDL, while Cooper, TGV and TorX receive specifications in LTS. Thereby, there is already an infrastructure in place that supports automation of state-based testing approaches (Finite State Machines [FSM] and its variations). State-based languages have often a high degree of formalism, which reduces their likelihood of adoption by practitioners. State Machines (one variation of state-based languages) are part of UML, which is the industry's de-facto standard modeling language. For this reason, state-based approaches included (namely AGEDIS, UMLAUT/Simulator, TESTOR) are UML-based.

As discussed in section 1, testing approaches based on specifications that non-technical stakeholders find easier to manipulate are more informative. Scenarios are examples of such specifications. Practitioners had adopted them as a means to express systems requirements and specifications [73]. For this reason, this survey includes scenario-based approaches.

1.2 Structure of the Survey

This section introduced some existing testing problems and suggested that they might be reduced with automated testing approaches that are based on specifications that are closer to expectations about the system-to-be. It also suggested that management of relationships among the artifacts used and produced by such approaches play an important role in their level of automation and adoption.

This survey concentrates on evaluating scenario-based and (UML) state machine-based automated testing approaches. Among other aspects, approaches were evaluated with respect to the level of automation of testing activities, to the kinds of artifacts used and produced, to the

¹ See glossary.

relationships required to exist among these artifacts, and to the support provided to manage those relationships. Section 2 explains the testing activities. To clarify the reason for evaluating scenario-based approaches, section 3 provides some background explaining the uses of scenarios at different levels of abstraction. It also explains how relationship management relates to specification-based testing. Then, Section 4 presents the evaluation framework, section 5 applies it to selected approaches, and section 6 discusses other relevant approaches not evaluated in detail. Section 7 concludes and makes research recommendations. In addition, section 8 describes a glossary of testing terms that might be useful for the reader that is not very familiar with the software testing vocabulary.

2 Automated Scenario-Based and State Machine-Based Testing

Specification-based testing [61, 63] is a way to test the software based on its specification. This means specification-based testing uses specifications to derive test scripts to test the system. Specifications describe how the software is expected to work. Therefore, the use of specifications renders the testing process more informative by comparing actual expectations about the system to the implementation. It indeed shows the system satisfies its original requirements. Automated specification-based testing is comprised by the following activities: test generation, execution, evaluation [61], coverage analysis and regression testing. Figure 1 depicts these activities, how they relate to each other, and some of the artifacts they use. Since these are automated activities, engines should support them. The following sub-sections explain each activity in more detail.

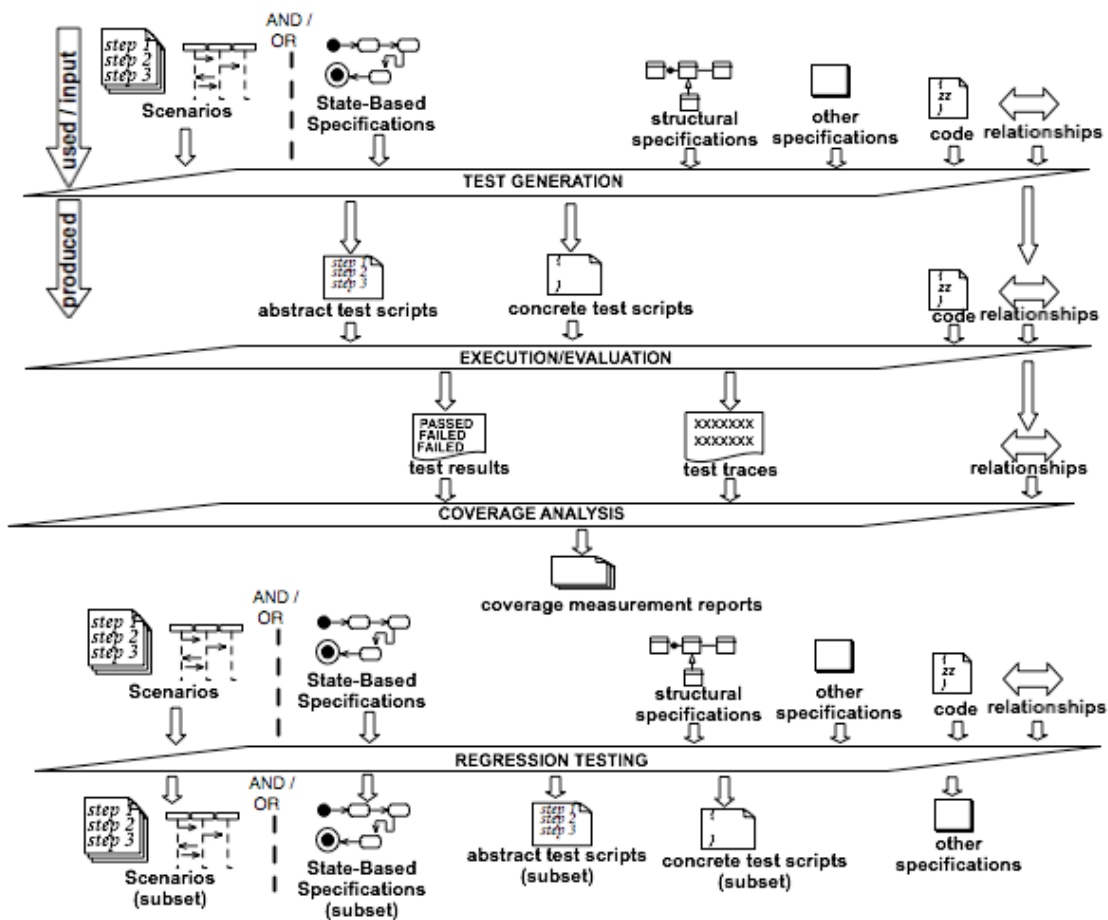


Figure 1 - Automated scenario-based and state machine-based testing

2.1 Test Generation

Test generation consists of the generator receiving a specification as input and creating *test scripts*. *Test scripts* (sometimes named *test sequences*) are a set of steps that should be followed when testing a program. Test values input and output should also be available with the test

scripts. Depending on the language and the level of abstraction at which they are written, test scripts can either be executed manually or automatically. Scripts for manual execution usually describe what steps the tester should follow when using the system, what inputs to provide and what outputs to expect. The evaluation of the result is also made manually. On the other hand, scripts for automatic execution consist of calls to system methods with what inputs to provide, and what outputs to expect. The evaluation is either made manually by the tester, or automatically by an oracle. Generation of such scripts from specifications requires establishment of relationships from the specification to the implementation.

Additionally, the test script can be at different levels of abstraction. At the *unit* level, it verifies if a unit of code is running as expected, this unit can be a module, a class, a procedure, and so forth. At the *integration* level, it tests that the units interact properly. It is thereby assumed that these individual units were already tested. Test scripts at the integration level not only compare the specification to the code. They can also exercise the specifications at different levels of abstraction comparing a lower level to a higher one. Thereby, generation of this type of script also requires knowledge of the lower level specifications, including implementation. Finally, at the *system* level, the test script evaluates the system as a whole against the requirements. This type of test is also called the Black-box test, since internal knowledge of the system is not necessary.

Figure 1 above shows that the test generation activity possibly receives as input different kinds of artifacts: scenarios at different levels of abstraction², state-based specifications (finite state machines and its variations), structural description of the system, code (sometimes named system under test - SUT), other specifications, and relationships among these artifacts. Depending on the information input, test scripts for manual or automatic execution can be generated. If the relationships between specification and the implementation are available (*relationships* in figure 1), test scripts that can be compiled and run can be generated. These are named *concrete test scripts* in figure 1. Otherwise, scripts for manual execution can be generated. These are named *abstract test scripts* in figure 1. In addition, since this activity creates new artifacts, it should also create new relationships among existing and new artifacts. This information is represented in figure 1 by the arrow leaving the test generation activity towards the *relationships* representation.

Usually, when scenarios are input, they are transformed into test scripts (concrete or abstract). These test scripts are later (see “execution and evaluation”) used to test the system by exercising the code (SUT). If state-based specifications are input, the tool traverses the specification, according to some pre-defined criteria. The criteria are used to guide path selection, as a way to reduce the number of paths generated. These paths are the test scripts generated. As it was observed with this survey, sometimes both scenarios and state-based specifications scenarios are provided as input to the test generation activity. In this case, the scenarios would be used to guide the test path selection. Test values in the generated scripts can either be created by the user or generated by the tool using existing techniques such as category partition [57].

2.2 Execution and Evaluation

Test execution and result evaluation are the next activities. Test execution runs test scripts. Result evaluation compares the results obtained against the expected results. Evaluation can be done manually or automatically. When done manually, it is labor intensive and error-prone. Thereby, evaluation by automated testing approaches demands automatic evaluation that is done by oracles.

² See explanation in section 3.1.

Test oracles³ consist of oracle procedure and oracle information. The procedure uses the information to verify that the execution behavior complies with the expected [62]. An oracle can therefore check that a final result obtained was the expected. It can also compare that the steps taken to reach that result (or the states through which the system passed to get to that result) are the same steps described in the specification.

Figure 1 above shows that execution and evaluation activities receive as input concrete or abstract test scripts, which were generated by the test generation engine. It also receives as input the code to be tested, among possibly other artifacts. Depending on the artifacts input, execution and evaluation can take place differently. If concrete test scripts are input, they are executed and the oracle evaluates the results obtained. The evaluation could compare the results obtained to the results expected, or the behavior obtained to the behavior expected.

If abstract test scripts are input, they need to be processed before automated execution can happen. In this case, the steps described in the abstract test scripts are used to guide the execution. The execution and evaluation activity needs to know how abstract test scripts relate to the code. This information is represented as *relationships* in figure 1. As shown in figure 1, the output of this activity could be test results and/or test traces. Test results are lists of the test scripts executed and evaluated, together with the results obtained for each of them. It informs if the test script passed or failed. Test traces are lists of the individual steps executed when the test script was run. For instance, these steps could be statements executed, or method calls executed. In fact, test traces inform the actual behavior of the system, which was obtained when the test script was run. In addition, since this activity creates new artifacts, it should also create new relationships among existing and new artifacts. This information is represented in figure 1 by the arrow that leaves the test generation activity towards the *relationships* representation.

It is important to note that sometimes coverage analysis (explained in the next section) is performed in parallel with the execution and evaluation. This happens in particular if the coverage is measured with respect to code (statement, method, branch).

2.3 Coverage Analysis

Tests results should be measured to inform stakeholders (testers, developers, managers) about the extent to which a set of test scripts covered the artifacts. The coverage can be measured in relation to the available software artifacts such as code, detailed specifications, or requirements. This information helps them make informed decisions regarding the testing process. It includes the number of test cases that passed, the number of test cases that failed, and how much coverage was achieved.

Figure 1 above shows that the coverage analysis activity receives as input test results or test traces, and relationships among test artifacts; it outputs a coverage analysis report. The coverage analysis report informs about coverage obtained with respect to different artifacts, or with respect to finer entities that compose the artifacts. Indeed, it depends on the granularity of the test results and test traces available. It also depends on the relationships connecting the test results to other artifacts based on which the coverage is measured. For instance, if relationships are available from individual requirements to test scripts and their results, requirements coverage can be measured.

2.4 Regression Testing

Software changes require further attention so as the quality of the final product will not be affected. When the software is modified, it needs to be retested to reduce the chances that new

³ See definition in section 9.

faults were inserted to the system. At the same time, test scripts that were already run and passed might not need to be re-run. Selective regression testing [31] deals with this issue by reducing the amount of test scripts that are run after a change is in place. Changes can happen to different software artifacts: directly to code, to the architecture, or to the specification, and selective regression testing can be performed for each kind of change [31, 47]. Ideally, automated specification-based testing approaches should support specification-based selective regression testing [78].

Figure 1 above shows that the regression testing activity can receive as input modified specifications (scenarios, state-based or structural), the relationships between specification, test scripts and other artifacts. It then outputs the selected test scripts to be re-run. It is important to note that in addition to selecting existing test scripts, new ones might need to be created. Thus, this technical activity could also output a subset of the specification (represented as ‘Scenarios subset’ or ‘State-based specifications subset’) to be used as basis for generating the new test scripts.

3 Background Discussion

This section discusses important concepts required for better understanding this survey. First, it explains the many uses of scenarios by the software engineering field and puts these uses in context with regards to testing. As detailed below, this discussion is important because scenarios are specifications close to user expectations that when expressed at lower level of abstractions are used as test scripts. It then discusses how relationship management relates to specification-based testing.

3.1 *Scenarios at Different Levels of Abstraction*

In software engineering, scenarios⁴ are used to express system requirements, components and objects interactions [72], and test scripts [25, 40]. They play an important role in software testing because concrete test scripts are code level scenarios annotated with further information to support comparison between obtained and expected results. Scenarios are used to describe expectations about the system. When scenarios are described at the right level of abstraction, they can be used and understood by the majority of stakeholders (users, customers, developers, testers, etc.). Therefore, testing approaches based on scenarios address the concerns of using of expectations to test the system.

Requirements engineering is one of the areas that most heavily explores the uses of scenarios [6, 8, 39, 73]. In this survey, scenarios at requirements level are scenarios used to describe requirements after a preliminary discovery phase (as distinguished from those used during the process of discovering the requirements). These scenarios are closer to stakeholders' understanding than scenarios used in later phases. They describe sequences of events that take place at the system-wide level of abstraction. These events occur in the real world (only) or are interactions between a user and the system-to-be. Information about the system's internal events is not pertinent to this kind of scenario. An example of this kind of scenario is seen in use cases, in which the individual traces (if present) are scenarios.

Scenarios at architecture level encompass scenarios that involve even more detail. In a manner analogous to architecture structure definitions, during this phase of scenario definition, decisions are made on which concepts will become components of the systems architecture. Thus, the scenarios are at the architecture level of abstraction, in which events are described as message exchanges among architectural components, or among components and connectors, depending on the architectural style chosen. An example of this kind of scenario is a UML sequence diagram, or a message sequence chart where the agents that exchange messages are architecture-level components and connectors.

Scenarios at design level are described in more detail than scenarios at architecture level. The concepts used in this case are object-parts of the architectural components. The design scenarios are thus scenarios at design abstraction level, much closer to code. While events are message exchanges among objects. A design scenario is an architecture-scenario that has been opened and elaborated to show the internals of the components. An example of this kind of scenario is seen in UML sequence diagrams, which are largely used by testing approaches denominated as scenario-based.

Scenarios at code level describe the flow of events at a system implementation level. They are in fact runtime trace information corresponding to statements executed/to be executed or method invocations, depending on the level of abstraction at which they appear. Thus requirements, analysis, architecture, or design scenarios, when executed, result in some code scenario. In this case, events are method invocations that describe object interactions. Scenarios at

⁴ See glossary for scenario definition.

design level can be related to scenarios at code level through the concepts used to describe them. Events from scenarios at design level are message exchanges among objects, and message exchanges become, in turn, method calls. Objects from design scenarios also appear as implementation concepts at the code level, and thus both concepts are related.

The scenarios explained above for each of the software life cycle phases can in fact be used as test scripts. They appear in each phase of the development life cycle, and are used to test system implementations or other system artifacts. Scenarios at requirements level are the closest to the users' comprehension, and are indeed used as guidance for Black-box testing. Scenarios at architecture, and design levels have internal information about the system, and thus can be used to simulate its behavior and guide white-box testing. Scenarios at each level of abstraction can be used to test the behavior of a lower level scenario conforms to the expected behavior described by the higher-level one. Since one higher-level scenario can lead to multiple lower level scenarios, there is a need to test their conformity. For instance, suppose there are two related scenarios, one at the design level and the other at the code level. A message (and related objects) in the design level scenario (UML sequence diagram) is related to several method calls (and object implementation) in the code level scenario. Simulation and correlation of lower level events to higher-level events check for conformity across those two scenarios. In fact, some automated approaches evaluated in this survey support this activity by using mapping from one level to the other to check the conformance.

3.2 Relationship Management and Specification-Based Testing

As discussed in the introduction, relationships among testing artifacts play an important role on the automation of testing activities. Common relationships required for software testing are relationships between specification and implementation, and between requirements artifacts and test scripts or test results. For instance, assume the requirements of system are somehow related to the test scripts written to test these requirements. Also assume that there are relationships from the test scripts to the test results obtained from running these test scripts. With this information, a tester can identify which requirements were already covered by a test run by simply following the relationships from requirements to test cases, and from test cases to the results. Given the importance of these relationships to testing activities, support for managing⁵ their life cycle within automated testing solutions should exist. It includes creation, persistence, maintenance, and destruction of these relationships (automatically, semi-automatically or manually⁶).

The artifacts at the end points of the relationships characterize the relationships. Therefore, changes to these artifacts might imply the relationship itself does not hold anymore. Thus, to fully achieve automation of specification-based testing activities, there is a need to automate management of testing artifacts and the relationships among them. Relationship management is a theme explored by the research on software traceability.

In their empirical study about the use of traceability, Balasubramaniam and Matthias [9] observe that “the semantics or the meaning attributed to linkage is guided by the reasoning that the user will be performing with the linkage”, so one can define different links⁷, or different attributes to the same link to relate artifacts one to the other with a specific use in mind. For instance, consider UML class and state machine diagrams. A link between a class and a state machine means that state machine is a model of that class' particular behavior. This link is useful for programmers to understand the behavior of the class they should implement. As another

⁵ Relationship management here means creating, persisting, consulting, maintaining and destroying relationships.

⁶ This classification was in [66].

⁷Note that a ‘link’ is the representation of a ‘relationship’.

example, consider links between methods from a class and scenarios at the code level. They can mean that the scenario will execute or executed that method. This information is important for performing regression testing. That way, given a change the method, the existence of this link (with that information) will reduce the effort spent on locating the test scripts that might be obsolete.

The commonality between specification-based testing and traceability lays on the need to manage relationships between artifacts. This means traceability infrastructures have the potential to be used with the specific aim of improving testing. This would leverage automation of some specification-based testing activities by supporting management of test-related artifacts and their relationships.

4 Evaluation Framework

Automation of specification-based testing approaches is one way to reduce effort spent on this activity. Consequently, it is a way to reduce testing problems. This survey evaluates some of these approaches with respect to the support provided to required technical activities: test generation, execution of results, evaluation of results, coverage analysis [61], and regression testing.

Since support for management of the relationships among the testing artifacts can contribute for automating specification-based testing activities, the evaluation of those approaches also aims to understand which artifacts are used by each activity, which relationships (if any) are established among these artifacts to support a particular technical activity, and how the approaches support management of these relationships.

Additionally, these approaches were evaluated with respect to other generally important aspects. These aspects include the level of formality of the specification language used by the approach (what affects the chances of adoption because of the learning curve), and the support for performing testing across different levels of abstraction (what adds value to the test because problems can be identified and solved before implementation phase is reached improving the software quality).

The evaluation framework is defined in the format of questions, where each question is related to one required technical activity, or to general aspects. The following four sections (4.1, 4.2, 4.3, 4.4) list and comment the questions. Section 4.5 lists and comments the questions concerned with the general aspects, and section 5 evaluates the approaches according to these questions.

4.1 Test Generation

As explained in section 2.1 and shown in figure 1, test generation activity generates test scripts from some specification input. The following questions were derived based on this activity, and will be used to evaluate selected approaches.

- A. Does the approach generate concrete test scripts? *Some approaches generate abstract test scripts to be manually executed (or to be provided as input to the execution and evaluation activity). An approach that automates test generation should generate concrete test scripts (test scripts that can be compiled and executed). This question highlights the level of automation of the approach with regard to test script generation.*
- B. When generating the test scripts, what is the coverage criteria used? *Approaches are more effective if the test scripts generated test important aspects of the system. Thus, some criteria should be followed during generation to avoid trying to cover all aspects when testing, which is near to impossible. Well-known examples of code-based coverage criteria are statement coverage, branch coverage, and path coverage. This question highlights what is the strategy used by the approaches to reduce the test suite.*
- C. What kinds of relationships are needed across artifacts to support test generation? *The answer to this question should clarify the following.*
 - (1) *If the approach considers the relationships across artifacts **explicitly** or **implicitly**⁸ when performing this particular activity.*
 - (2) *If relationships are **fine-grained** or **coarse-grained**. This is defined by the artifacts used by each approach and on the endpoints connected by the relationships*
 - (3) *If the relationship is across models at the same level of abstraction (**horizontal**) or across different levels of abstraction (**vertical**). Vertical relationship is often called*

⁸ See Glossary.

- refinement, reification or mapping (when the relationship is from a specification to the code).*
- D. Does the approach provide support for management of these relationships? *If relationships are used across artifacts to perform this activity, it is important to know how they are created and if there is support (manual or automatic) for maintaining them. Maintenance is important because artifacts can change, and relationships can become invalid.*

4.2 Execution and Evaluation

As explained in section 2.2 and shown in figure 1, execution and evaluation consists of running the test scripts and ensuring compliance between the obtained and expected results and/or behavior. The following questions were derived based on this activity, and will be used to evaluate the selected approaches.

- E. Does the approach allow the specification to be used as oracles? *The use of specification-based testing approaches implies creation of specifications. Specifications are expected behavior of a system and should be used as input to oracles.*
- F. Is the comparison between obtained behavior and/or result and expected behavior and/or result automatic? *This question highlights the level of automation of the approach with regard to evaluation of results.*
- G. What kinds of relationships are needed across artifacts to support execution and evaluation? *See section 4.1, question C.*
- H. Does the approach provide support for management of these relationships? *See section 4.1, question D.*

4.3 Coverage Analysis

As explained in section 2.3 and shown in figure 1, coverage analysis consists of generating coverage measurement report that informs the user about the coverage achieved by the test scripts run with respect to different artifacts or with respect to finer-grained entities that compose the artifacts. The following questions were derived based on this activity, and will be used to evaluate the selected approaches.

- I. Does the approach support coverage analysis? *Coverage analysis is important to inform stakeholders (testers, developers, managers) about the extent to which a set of test scripts covered the artifacts. It should, therefore, be supported by automated approaches.*
- J. What kinds of relationships needed across artifacts to support coverage analysis? *See section 4.1, question C.*
- K. Does the approach provide support for management of these relationships? *See section 4.1, question D. It is also important to understand if the approaches even create the relationships between results obtained and specification originating them with the intent of measuring coverage.*

4.4 Regression Testing

As explained in section 2.4 and shown in figure 1, regression testing consists of trying to reduce the chances that new faults were inserted when a system is modified. While selective regression testing consists of doing the same but avoiding re-run all test cases. Thus, automated specification-based testing approaches should support specification-based selective regression testing.

- L. Does the approach support selective regression testing? *This question was included because regression testing is an important testing technical activity and ideally should be supported by automated testing approaches.*

Since we are evaluating specification-based testing approaches, we do not expect them to support other kinds of regression testing than specification-based ones.

4.5 General Aspects

Some aspects about the approaches allow us to learn about its usefulness and the chances of being adopted.

- M. What specification language is used?
- N. What is its level of formality? *High, Medium, Low. This affects the chances of adoption because of the learning curve. The answer to this question also explains whether it is possible to have approaches that automate technical activities while maintain a low/medium level of formality.*
- O. Does the approach support performing testing across levels of abstraction? *As explained before in section 2, such support adds value to the testing activity.*
- P. What is the level of abstraction of the specification input? *They can be at Requirements, architecture, design, or code levels. (See figure 1).*
- Q. What is the level of automation of the approach? *Considering the number of the technical activities are automated, (1 activity – Low, 2 activities – Medium, 3 activities - High)*

5 Applying the Evaluation Framework to Scenario-Based and State-Based Approaches

This section evaluates relevant automated approaches to specification-based testing, within the scope explained in section 1.1.

5.1 Sequence Diagram Test Center (*SeDiTeC*)

Sequence Diagram Test Center (SeDiTeC) [29, 28] implements one approach that executes UML sequence diagrams to test java applications. The use of UML sequence diagram is motivated by the wide adoption of use case driven approaches to software development (e.g. RUP). Sequence diagrams describe use case realizations, which are described through message exchange among objects. They characterize a relationship between use case and sequence diagram. Relationships are not explicitly shown nor used by SeDiTeC's users. They are, however, used to discover which objects implement the use cases' functionalities.

The tool supports the execution of sequence diagrams, so it uses the sequence diagrams to test the integration among objects. Sequence diagrams are extended with actual parameters of the methods in the sequence diagrams and with expected results. This approach names the information that complements the sequence diagram as *test case data set*. Test case data set can optionally contain information about exceptions that can be thrown. The sequence diagrams complemented with the test case data set is called the *test specification*. It is worth noting that the approach also imposes some requirements for the sequence diagrams to be testable. Whereby the sequence diagrams should describe sequences of method calls that can be executed in the final implemented system, so their level of abstraction is low.

Two major features of the tool are: support for combination of sequence diagrams, and support for stub generation. Stub generation mechanism allows for simulation of the behavior of classes not yet implemented. The tool generates stubs for those classes, and the behavior of the stub can be changed dynamically according to the sequence diagram description. These mechanisms rely on the fact that the objects in the sequence diagrams are instances of distinct classes from the class diagram. The support for combination of sequence diagrams should be used to combine diagrams described separately into more complex sequence diagrams with conditional branches.

In general, the automated approach works as follows. The user creates UML diagrams using a UML case tool. The user then exports these diagrams as XML format to be imported by SeDiTeC. On the first step, the user annotates the sequence diagrams with test case data set; this means the user is creating the test cases to be run. Next, the user indicates how the sequence diagrams should be combined, so he/she creates *combined sequence diagrams*. In particular, the authors classify the sequence diagrams into two types: one that is meant to set up the initial state of the system and the other actually contains methods to be tested. Thus, these two types of sequence diagrams should be associated. The association is done when objects with same name are used in different sequences diagrams. Next, the user will select the test cases describing the *combined test cases* (these test cases had been already defined in the first step). For classes not yet implemented, whose objects are part of the sequence diagrams to be tested, the user should invoke the stub generator functionality. Of course, it does not make sense to run the test specifications against the implementation of stub-only classes, but once the classes are implemented, test will be performed against actual implementation.

To evaluate the results, the source code is instrumented to log information about the method calls, parameters and results obtained from each method. The result of the execution is an observed sequence diagram that is compared to the expected sequence diagram to determine if test passed or failed. It is not always possible to automatically reach an answer of whether the

observed and expected sequence diagrams match. This happens because the observed sequence can be a refinement of the expected one. By the authors' definition, a sequence diagram "A" refines a sequence diagram "B" if the information contained in "B" is a subset of the information contained in "A", meaning that "B" was not detailed enough. In this case, it is up to the tester to decide if they will pass or fail diagram refinements.

5.1.1 Application of the Evaluation Framework

Figure 2 below depicts the specification-based testing activities. It shows the activities supported and the artifacts required by this approach (in black). Because approaches evaluated do not support all activities and because they use a variety of artifacts, this figure helps on placing the approach in the context of full automation.

This approach supports test generation, execution and evaluation. The specifications input for the test generation are scenarios at the design level (UML sequence diagrams). Other artifacts include use case (represented as ‘other specifications’) and class diagrams (structural specifications), relationships among those artifacts, and code (SUT). Test generation outputs concrete test scripts, which are executed and evaluated. It then produces test results. Table 1 answers the questions from the evaluation framework. Questions that are not applicable to this approach were answered with N/A.

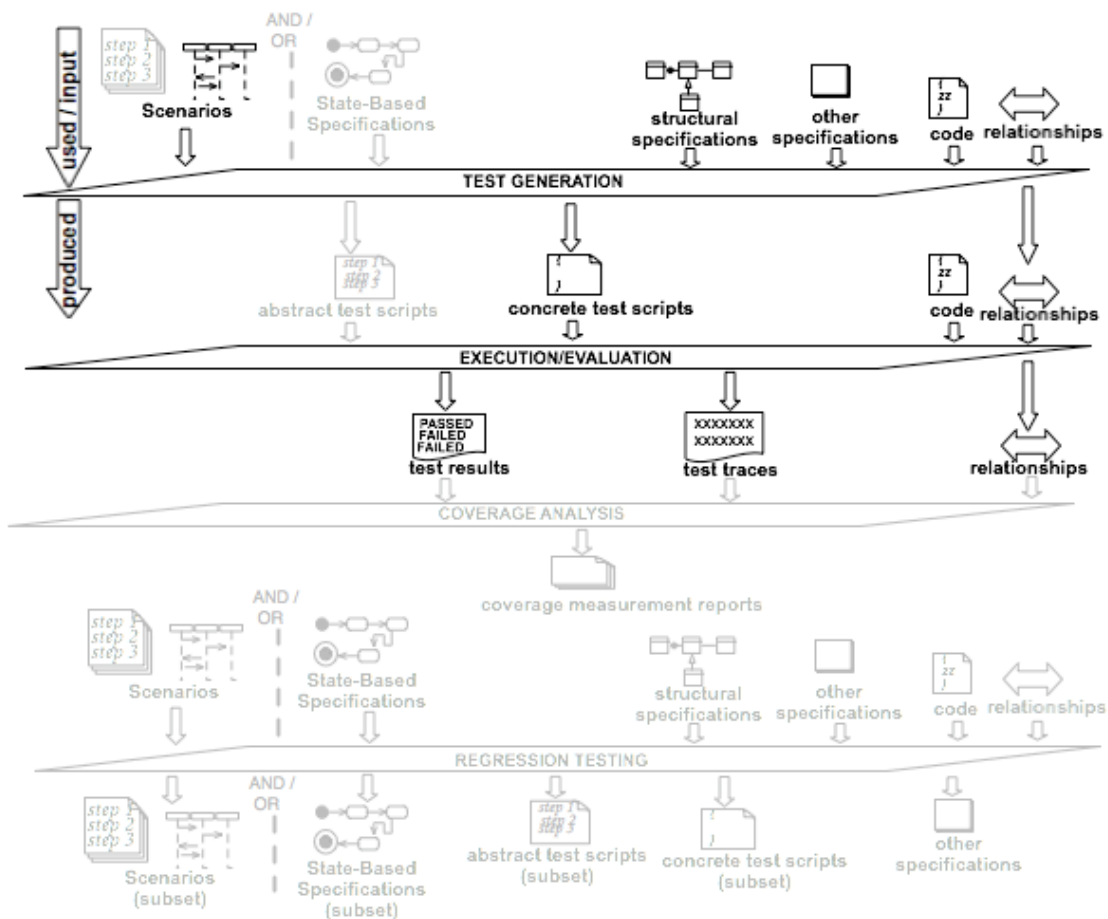


Figure 2 - Technical specification-based testing activities supported by SeDiTeC

Test Generation	Generates concrete test scripts?	<i>Yes, they are generated with the annotated sequence diagrams that the user inputs.</i>
	Coverage Criteria?	<i>No, the tester decides which and how many sequence diagrams will be executed, and inputs the test data.</i>
	Kinds of relationships needed to support test generation?	Implicit, Horizontal, and Fine-grained. They are obtained from UML model imported by the tool. They exist across UML models at the same level of abstraction (e.g. objects in the sequence diagrams are related to the classes they belong to in the class diagram). They relate every method call in the sequence diagram to a declared method in the target class in the class diagram. Implicit, Vertical and Coarse-grained. They relate classes in the class diagram to their associated source code. Implicit, Vertical and Fine-grained. They relate method in the target class in the class diagram to a method in the source code.
	Support for management of these relationships?	<i>No.</i>
Test Execution and Evaluation	Specification used as oracle?	<i>Yes, it uses the annotated sequence diagrams to exercise the code, and the output trace is matched to these annotated sequence diagrams.</i>
	Automated evaluation?	<i>Yes, the concrete test scripts are used to exercise the code (implemented classed and stubs), which was previously instrumented. This exercise produces observed sequence diagrams (test traces). The approach matches expected sequence diagrams to observed sequence diagrams by comparing the following information: order of the method calls, identity of method calls, identity of objects that initiate the method calls, parameters and return values, and thrown exception. Resulting execution of the instrumented code produces an xml file that includes all information necessary to create a sequence diagram. This information is enough to compare the obtained sequences to the expected ones. If ALL aspects are equal, the test passes otherwise it fails.</i>
	Kinds of relationships needed to support execution and evaluation?	Implicit, Vertical and Fine-grained. They are the comparison between the obtained and the expected behavior to support evaluation, which relate methods in the sequence diagrams to methods in code. Implicit, Horizontal and Fine-grained. They are used to execute the test scripts. They are relationships from the concrete test scripts to the code it tests.
	Support for management of these relationships?	<i>No.</i>
Coverage Analysis	Support for coverage analysis?	<i>No, the tool generates a report, but does not measure coverage achieved.</i>
	Kinds of relationships needed to support coverage analysis?	<i>N/A. However, it maintains implicit relationships from sequence diagrams to their respective test specifications.</i>
	Support for management of these relationships?	<i>N/A, relationships are not explicitly created nor maintained.</i>
Regression Testing	Support for regression testing?	<i>No.</i>
General Aspects	Specification Language?	<i>UML use case, sequence and class diagrams.</i>
	Level of formality?	<i>Low.</i>
	Testing across levels of abstraction?	<i>Yes, it compares the expected sequence diagram to the observed sequence, but does not do it across sequence diagrams at many levels of abstraction.</i>
	What is the level of abstraction of the specification input?	<i>Scenarios at the design level.</i>
	What is the level of automation of the approach?	<i>High (automates two or more activities)</i>

Table 1 - Evaluation framework applied to SeDiTeC

5.2 SCENTOR

SCENTOR [76, 75] is an approach that supports generation of scenario-based testing using JUnit⁹ [30] as basis. Its domain is e-business-specific, and the applications tested should be based on Enterprise Java Beans (EJB). It uses UML sequence diagrams as inputs, which are scenarios at the design level of abstraction.

Similarly to SeDiTeC, it assumes that the user creates the sequence diagram with some case tool. The user exports the diagram to the common XML representation and loads it to SCENTOR. After the sequence diagrams are loaded into SCENTOR, for each method in the sequence diagram, the user specifies the methods' concrete parameter values and expected results. These are written in Java, which is the language used by the JUnit framework. The sequence diagram loaded to SCENTOR and annotated with such information is called *test specification*. The approach assumes existence of direct match between the objects and methods from the sequence diagram and the objects and methods from the implementation. After the test specification is created, SCENTOR can generate the test scripts by extending JUnit framework classes. The resulting test scripts are similar to those written manually, which are compiled and run by SCENTOR (with Java's compiler and virtual machine.)

Other features of SCENTOR are the generation of global setup test code and global tear down test code. These can be shared by multiple test cases. The advantage of supporting global setup and teardown is that they are sometimes time consuming and sharing them can reduce this effort.

⁹ See glossary.

5.2.1 Application of the Evaluation Framework

Figure 3 below shows the approach supports test generation, execution and evaluation. The specifications input for the test generation are scenarios at the design level (UML sequence diagrams.) The annotations created by the user in Java are represented by ‘other specifications’. This activity outputs JUnit-based concrete test scripts, which can be compiled, run and evaluated. Table 2 answers the questions from the evaluation framework. Questions that are not applicable to this approach were answered with N/A.

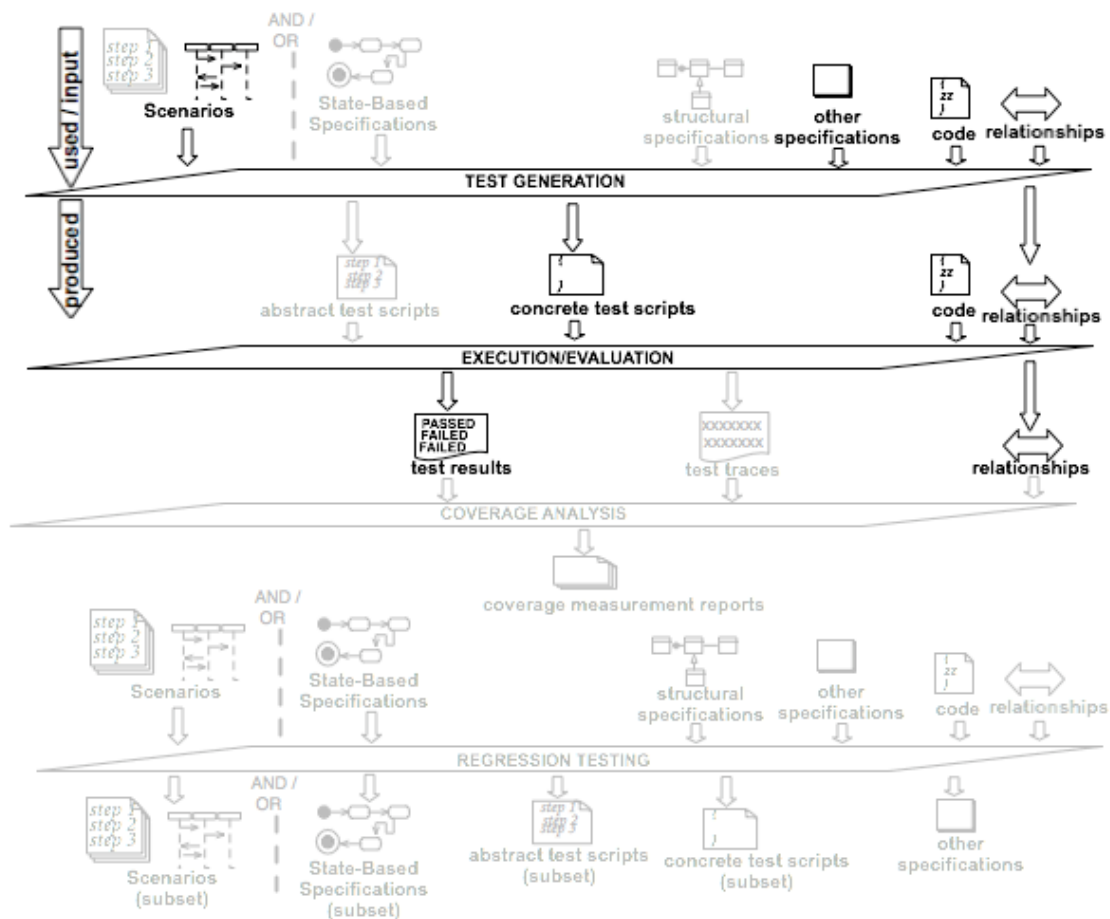


Figure 3 - Technical specification-based testing activities supported by SCENTOR

Test Generation	Generates concrete test scripts?	<i>Yes, they are generated with the annotated sequence diagrams that the user inputs.</i>
	Coverage Criteria?	<i>N/A, the tester decides which sequence diagrams will be executed, and also inputs the test data.</i>
	Kinds of relationships needed to support test generation?	Implicit, Vertical and Fine-grained. They are defined by matching the names of the methods in the sequence diagrams and the names of the methods in the implementation.
	Support for management of these relationships?	<i>No.</i>
Test Execution and Evaluation	Specification used as oracle?	<i>Yes, it uses annotations about the results on the sequence diagrams to compare them to the obtained results. The behavior, however, is not compared step by step.</i>
	Automated evaluation?	<i>Yes, only the results are compared. This is a feature from the JUnit framework that is used as basis for this approach.</i>
	Kinds of relationships needed to support execution and evaluation?	Implicit, Horizontal and Fine-grained. They are used to execute the test scripts. They are relationships from the concrete test scripts to the code it tests. The evaluation compares the expected result for each method with the obtained result. It is based on the expected results written in Java by the user and available as part of the executable test script.
	Support for management of these relationships?	<i>No.</i>
Coverage Analysis	Support for coverage analysis?	<i>No, the tool generates test results, but does not measure coverage achieved.</i>
	Kinds of relationships needed to support coverage analysis?	<i>N/A.</i>
	Support for management of these relationships?	<i>N/A.</i>
Regression Testing	Support for regression testing?	<i>No.</i>
General Aspects	Specification Language?	<i>UML sequence diagrams.</i>
	Level of formality?	<i>Medium, it annotates the sequence diagrams with programming language.</i>
	Testing across levels of abstraction?	<i>No.</i>
	What is the level of abstraction of the specification input?	<i>Scenarios at the design level.</i>
	What is the level of automation of the approach?	<i>Medium.</i>

Table 2 - Evaluation framework applied to SCENTOR

5.3 COWtest pluS UIT Environment (COW_SUITE)

COWtest pluS UIT Environment (COW_SUITE) [12, 13, 14, 15] is a tool that provides systematic and automated support for test planning and test case generation, based on UML diagrams. It addresses test case prioritization and test case generation problems. It is composed of two other tools: Cow_test (COst Weighted TEst STrategy) and the UIT (Use Interaction Testing). The first one provides support for test case prioritization, and the second provides support for test case generation. This survey concentrates on the UIT tool.

UIT supports test case plan generation directly from UML diagrams (use case, sequence, and class diagrams), without requiring the user to provide further formalism. In particular, UIT is based on sequence diagrams that describe use case realizations through message exchange among objects. For each use case, it generates a use case test suite, which is a set of test cases built for the same use case. The approach is based on the idea that use cases can be organized hierarchically. Thus, the use case diagram can lead the test construction because it defines incremental stages of test integration from the lower level abstraction to the higher ones. This section explains the artifacts used by this approach and how UIT works to generate their test artifacts.

The specification used by COW_SUITE is imported from Rational Rose tool. From the project imported, it constructs a *Main graph* and a *Design graph*. The *Main graph* is a high level representation of the system. It consists of actors, use cases and sequence diagrams, which are the nodes of the graph. The relationships among those elements (realizes, extends, uses, traces) are the edges of the graph. The use cases represent the functionalities and sub-functionalities of the system, while the sequence diagrams describe how the use cases are realized by the interactions between objects and actors. From the *Main graph*, COW_SUITE generates *Main trees*. A *Main tree* is a hierarchical functional decomposition of the system. It is represented by use cases and by the sequence diagrams that realize them. At each level of the tree, there can be use cases or sequence diagrams. Use cases in a deeper level represent sub-functionalities of use cases in a higher level. Each level of the tree describes a different degree of detail of the system's functionalities, so it represents a specific level of integration. The *Design graph* consists of packages, developed components and dependencies among them.

The *Main tree* can be seen from two different perspectives: *Use Case view* and *Logical view*. The *Use Case view* shows the actors, use cases and sequence diagrams. They represent significant system's functionality and interactions between the system and the external world. This view shows how the expected system's functionalities are realized by the use cases. A path from a high-level use case to the leaf in this view shows the refinement of the associated functionality. Therefore, tests derived from sequence diagrams in this view are at a higher level of abstraction. They are used for testing system integration and sub-system integration. The *Logical view* shows the allocation of system's functionalities to the static structure of the system. It also shows the dynamic collaboration among objects. The *Logical view* has use case realization information that traces how the use case evolved into the design model. Sub-flows of use cases are detailed with sequence diagrams. In addition, the logical view specifies classes participating in the realization of the use case. It shows a list of packages that implement the use case (this is an explicit link between use cases and the packages that implement them).

The UIT method is the part of COW_SUITE that supports derivation of test procedures from UML diagrams. The test artifacts used by UIT are explained bellow.

Test Suite is the set of all *Test Procedures*. For each use case, UIT generates a corresponding test suite.

Test Procedure is a sequence of messages and their associated parameters (it can be thought as being an abstract test script). *Test Procedures* are automatically generated from the test specifications.

Test Specifications are created considering all values choices, which are input by the user

interacting with the tool. For each category associated with the test case, all possible values are considered.

Test Case consists of the messages in the *Messages Sequence*, *Interaction Categories* and *Setting Categories*.

Interaction Categories are messages entering an object in the sequence diagram; it can be identified by observing the messages received by that object in its temporal (vertical) line in the sequence diagram.

Setting Categories are values or data structures that can influence the execution of the messages sequence. They are defined by considering the input parameters to messages involved, by analyzing the class to which the message belong, and by examining the attributes and data structures that can affect the observed interaction.

Messages Sequences are a set of messages used to realize a single functionality. They consist of a list of messages composed of: (1) A message or messages entering the main object used to realize a single functionality (this list is obtained by following the temporal order of that object in the sequence diagram and by listing the messages), (2) A message or messages exchanged among other objects with the objective of realizing that same functionality. Thus, several categories (interaction and/or setting) can correspond to a message sequence, because messages in a message sequence identify *Interaction Categories*, while the attributes that affect these messages identify *Setting Categories*.

UIT is an incremental test methodology that supports multiple levels of abstractions. The expressiveness of the test procedures generated depends on the level of detail available with the sequence diagrams (if more detail is provided with the diagram, a more expressive test procedure is generated). It assumes a UML project is provided with use case, sequence and class diagrams. It also assumes the Main Tree is generated.

For each sequence diagram selected as basis for the test case derivation, the following steps are taken. For each object in the sequence diagram, their Interaction Categories are identified. Then, the next step is to define the Messages Sequences. For each Message Sequence, Setting Categories are defined. Each Message Sequence defines a test case, and the messages in the test case are analyzed to see if there are feasibility conditions and if the sequence needs to be divided into sub-cases of different choices. Therefore, the partial result is that each sequence diagram is associated with test cases and their sub-cases. For each test case, its test specification is defined. In this step, the user interacts with the tool to insert the choices values, and add constraints for the case of contradictory combinations (this is determined by the category partition methodology). Finally, the test procedures are automatically generated from the test specifications. It is worth noting that these test procedures are not yet an executable test script, and there is still a need for a test driver to execute and log the results, and this tool does not provide such support. The output of this tool is a text file describing the test suite.

5.3.1 Application of the Evaluation Framework

Figure 4 below shows the approach supports test generation. The specifications input for the test generation are scenarios at the design level (UML sequence diagrams); other artifacts include use case diagrams that are represented as ‘other specifications’, class diagrams (represented as ‘structural specifications’), and relationships among those artifacts. This activity outputs abstract test scripts, which require further implementation to be run. Table 3 answers the questions from the evaluation framework. Questions that are not applicable to this approach were answered with N/A.

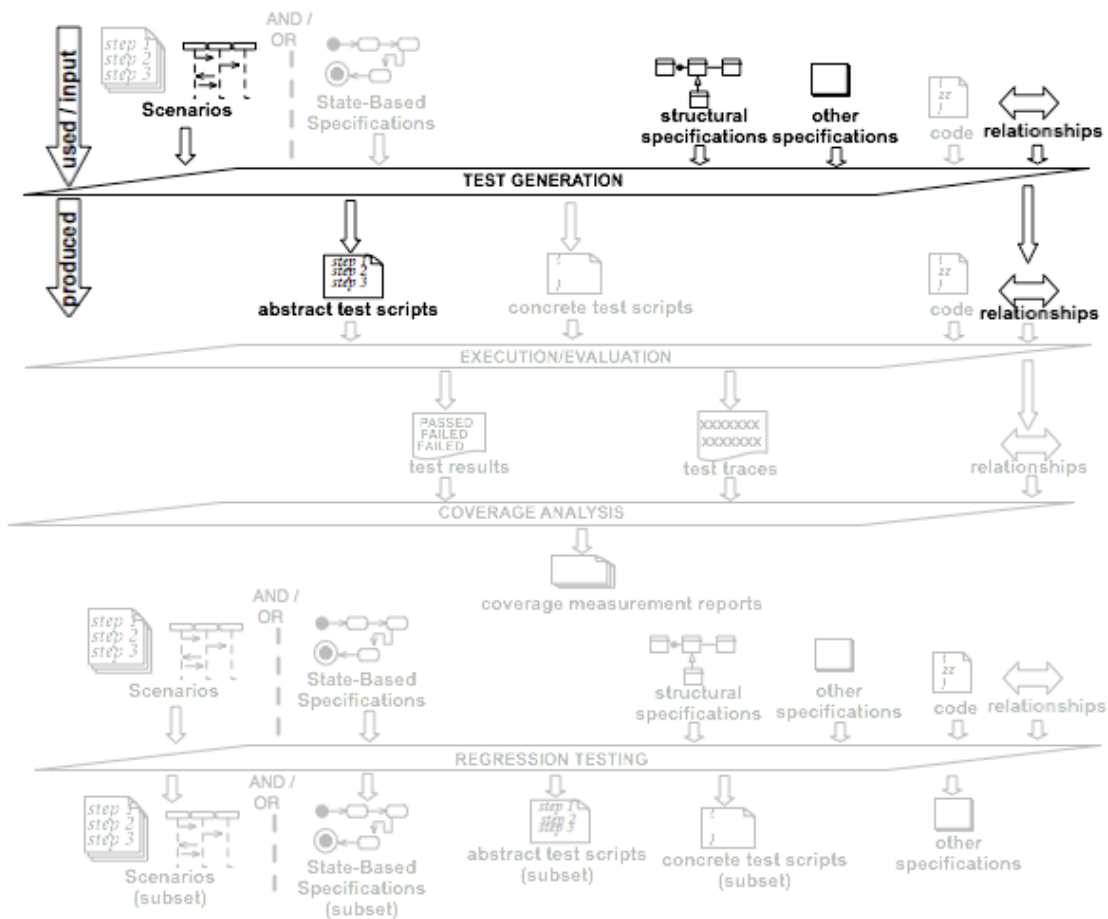


Figure 4 - Technical specification-based testing activities supported by COW_SUITE

Test Generation	Generates concrete test scripts?	<i>No, it generates abstract test scripts (test sequences).</i>
	Coverage Criteria?	<i>Fixed number of test procedures or functionality coverage.</i>
	Kinds of relationships needed to support test generation?	<p>Explicit: relationships are implicit in the UML model imported by the tool. When the tool generates the hierarchical trees, the relationships across diagrams become explicit. In the hierarchical trees, nodes are diagrams and other elements from the model (e.g. packages). The edges in the trees represent the relationships. Relationships across trees are also made explicit.</p> <p>Explicit, Horizontal, and Fine-grained. They are defined across UML models at the same level of abstraction. They relate objects in the sequence diagrams to the classes they belong to in the class diagram. Every method call in the sequence diagram is associated with a declared method in the target class in the class diagram.</p> <p>Explicit, Vertical, and Fine-grained. They are defined across UML models at different levels of abstraction. Additionally, They are defined between class diagrams and source code. They relate method in the class diagram is to methods in the code.</p> <p>Explicit, Vertical, and Coarse-grained. They are defined across UML models at different levels of abstraction. These are some of the relationships available from the Main Tree. Each level of the main tree describes a different degree of detail of the system's functionalities, and thus a vertical refinement relationship. Among other artifacts, these relationships relate use cases and sequence diagrams (a sequence diagram realizes a use case), and use cases and the packages that implement it.</p>
	Support for management of these relationships?	<i>No, relationships are made explicit by the tool, not created with it. But changes made to the UML model (that might also affect the relationships among models) resulting from the test generation process, such as changes to parameters or operations, can be saved to a new UML model and reloaded.</i>
Test Execution and Evaluation	Specification used as oracle?	<i>N/A.</i>
	Automated evaluation?	<i>No.</i>
	Kinds of relationships needed to support execution and evaluation?	<i>N/A.</i>
	Support for management of these relationships?	<i>N/A.</i>
Coverage Analysis	Support for coverage analysis?	<i>No, however, some relationships could support manual coverage analysis.</i>
	Kinds of relationships needed to support coverage analysis?	<p><i>The tool does not support test execution, so it does not generate test results. The coverage analysis (if done) is manual. When creating the test sequences, however, the tool creates relationships that could be used to partially support this activity. Explicit, Horizontal and Coarse-grained. They relate a sequence diagram to a set of test cases and their sub-cases, test specifications to their originating test cases, test procedures to their originating test specifications, and a test suite to its set of test procedures.</i></p>
	Support for management of these relationships?	<i>No.</i>
Regression Testing	Support for regression testing?	<i>No.</i>

General Aspects	Specification Language?	<i>UML use case, sequence and class diagrams.</i>
	Level of formality?	<i>Low.</i>
	Testing across levels of abstraction?	<i>No.</i>
	What is the level of abstraction of the specification input?	<i>Scenarios at the design level, as supported by the UML tool where the specifications are created.</i>
	What is the level of automation of the approach?	<i>Low.</i>

Table 3 - Evaluation framework applied to COW_SUITE.

5.4 Automated Generation and Execution of test suites for DIstributed component-based Software (AGEDIS)

The Automated Generation and Execution of test suites for DIstributed component-based Software (AGEDIS) [37, 36] project provides an integrated environment for modeling, model-based test generation, execution and other test related activities. The approach requires the user to provide the behavioral model of the system under test. Differently from the other approaches evaluated in section 5, AGEDIS does not use scenarios as the main modeling artifact. Instead, it uses state-based specifications.

AGEDIS accepts as input models described with the AGEDIS Modeling Language (AML). AML is an extension of UML that uses a subset of the UML diagrams types: class diagrams, object diagrams, and state diagrams. The purpose of the language is to describe existing systems and what they are able to do (as opposed to how the system is used). Therefore, the language does not support UML use case diagrams. AML extends UML by defining some stereotypes, tagged values, and a target language named Intermediate Format (IF) [20]. The target language is used to annotate the models with details about operations, actions and data types, which allows the models to be compiled. The tool compiles and simulates the model, based on which the generator creates the possible test sequences.

In addition to the AML model, the users create *test generation directives*. Test generation directives comprise information about the strategies used to generate the *abstract test suite*. Test generation directives are described as a UML state machine annotated with regular expressions, and they are created using a UML case tool. A compiler interprets the AML model and the directives. It converts the interpreted information into the intermediate format (IF) [20]. The result can be used by the model simulator to support model debug, and the by test generator to create *abstract test suites*. *Abstract test suite* is a set of *abstract test cases*, which is a sequence of input stimuli and the expected system's behavior in response to those stimuli (abstract test script).

Abstract test cases cannot be compiled and executed. They are used to guide the execution of the code. The users create *test execution directives* to support test execution. The *test execution directives* comprise information about where and how to execute the abstract test suite. Particularly, it contains relationships from the system's model to its implementation (data types, classes, methods, and objects). It also contains information about the environment where the tests are executed. The execution engine uses this information to execute the abstract test cases. Execution is done with the following steps: (1) generating the stimuli, (2) observing the response, (3) comparing the response to the expected response (it plays the role of an oracle), and (4) logging the execution trace.

AGEDIS also supports coverage analysis, defect analysis and feedback. A functional coverage tool named FoCuS [2] is integrated to AGEDIS. It allows the user to describe a functional coverage model in terms of methods and attributes of the system's implementation. It reports on combinations of data values not covered, and sequences of methods not invoked. Integration to AGEDIS is as follows. FoCuS receives as input either the test suites or the test traces, so the user does not need to create a functional coverage model. As a result, FoCuS itself can generate coverage analysis reports. Through a feedback interface, it creates the test purposes (test purposes is one part of the test generation directives) that direct the test generator to create test cases that cover the sequences of methods and data values not yet covered. The defect analysis and feedback tool clusters test cases according to the identified defect similarities and to steps in the test case prior to the observation of the defect.

From the description of AGEDIS integrated environment, we can observe that the major activities described in the evaluation framework are addressed. However, not all activities are automated. This approach requires extended knowledge by the user about the particularities of the IF language, which is used to annotate the models and support their execution.

5.4.1 Application of the Evaluation Framework

Figure 5 below shows the approach supports test generation, execution and evaluation, and coverage analysis. The specification input for the test generation is state-based specification (more specifically, an extension of UML, named AML); class diagrams (represented as ‘structural specifications’); other artifacts such as object diagrams that are represented by ‘other specifications’; and relationships among some of these artifacts. This activity outputs abstract test scripts, which require relationships from abstract concepts to code concepts to be executed.

An integrated tool supports coverage analysis. Differently from other approaches, coverage analysis can be performed either in relation to abstract test scripts or in relation to resulting test traces. However, this information is not depicted in figure 5. Table 4 answers the questions from the evaluation framework. Questions that are not applicable to this approach were answered with N/A.

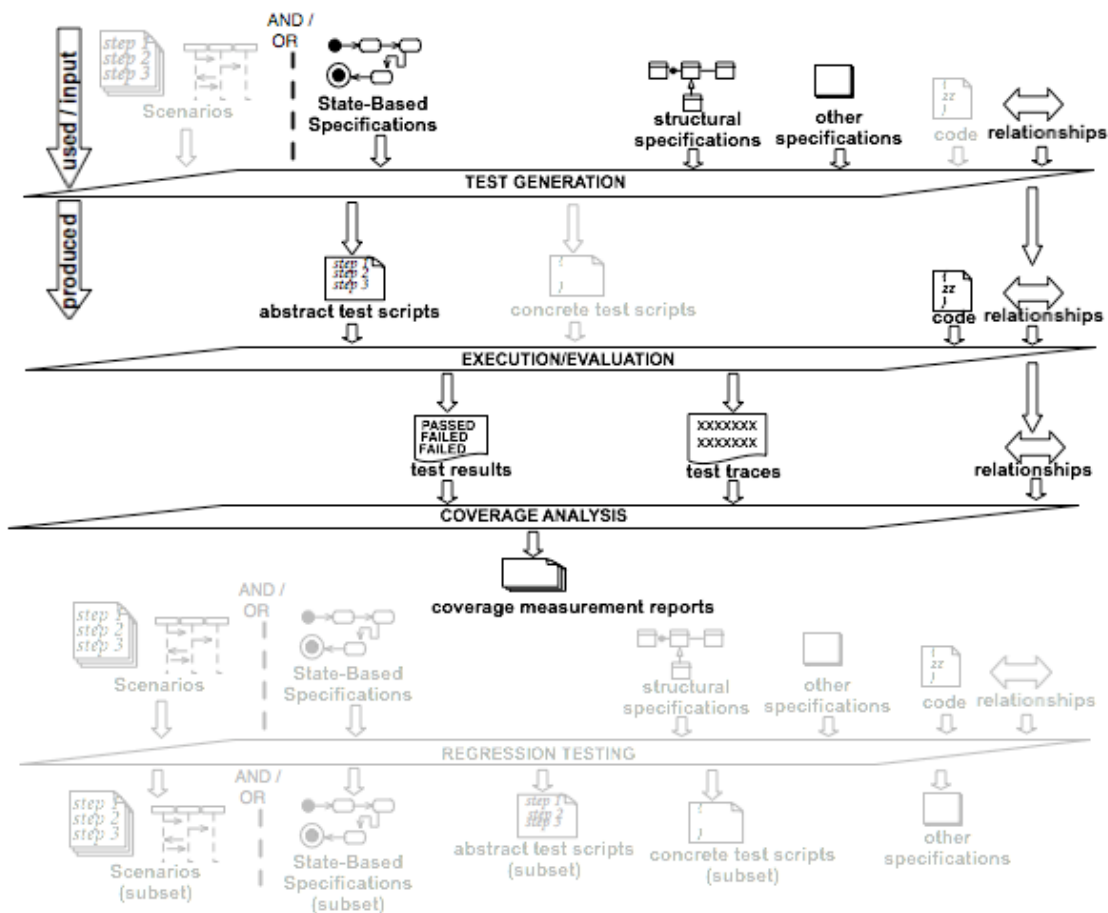


Figure 5 - Technical specification-based testing activities supported by AGEDIS

Test Generation	Generates concrete test scripts?	<i>No, it generates abstract sequences from state machines</i>
	Coverage Criteria?	<i>Customizable</i>
	Kinds of relationships needed to support test generation?	Implicit, Vertical and Fine-grained. Since the approach transforms AML specifications into IF, it assumes relationships between models expressed in these languages ¹⁰ . Implicit, Horizontal and Coarse-grained. Test case generation is based on TGV and GOTCHA [41, 26] test generators. It explores the state space (with user selected test generation strategies) and outputs an XML file describing the set of test scripts (test suite). The test scripts generated are abstract. They are described at the modeling level. They are derived from AML models that must have relationships among them. For instance, relationships from state machines that describe the behavior of objects to a class in a class diagram.
	Support for management of these relationships?	<i>No.</i>
Test Execution and Evaluation	Specification used as oracle?	<i>Yes, the abstract test case contains information about the results expected from applying the stimulus to the implementation.</i>
	Automated evaluation?	<i>Yes, an execution engine uses test execution directives to execute the abstract test cases by generating the stimuli, observing the response and comparing it with the expected (it plays the role of an oracle), and logging the execution trace. The user can also use the model simulator that supports observation and debug of the behavioral model, which is a kind of manual evaluation.</i>
	Kinds of relationships needed to support execution and evaluation?	Explicit, Vertical, and Fine-grained. To execute and evaluate the test scripts, the abstract test cases must be related to the code. The abstract stimuli are mapped to method invocations and abstract observations are mapped to value checking. This, in addition to other information guides test execution. Together they are called the test execution directives. The user creates them.
	Support for management of these relationships?	<i>Yes, support is provided for the user to manually create and modify the test execution directives (the relationships).</i>
Coverage Analysis	Support for coverage analysis?	<i>Yes, with the integrated coverage analyzer tool named tool named FoCuS [2]. FoCuS supports coverage analysis of methods (code), and attributes of the objects (this kind of coverage is based on models provided by the user, which is a specification of objects' attributes' values). As adapted for AGEDIS, the coverage analyzer reads either the test suite (what will be tested) or the test traces, and outputs test purposes aiming to cover method sequences and values not yet covered. Additionally, it supports visualization of test traces and the abstract test suite that generated them, allowing for other manual coverage analysis.</i>
	Kinds of relationships needed to support coverage analysis?	Explicit, Vertical and Coarse-grained. abstract test suites are associated with the test traces resulting from their execution. Implicit, Horizontal, and Fine-grained. They relate test traces to the code, because each method in the resulting trace was a call to a method in the code.
	Support for management of these relationships?	<i>Yes, partially. The association between the abstract test suites and the test traces are created and persisted when tests are executed.</i>
Regression Testing	Support for regression testing?	<i>No.</i>

¹⁰ This is further discussed in section 6.4.

General Aspects	Specification Language?	<i>AML - extension of UML (UML class, object, and state machine diagrams), and their proprietary Intermediary Format (IF) Language.</i>
	Level of formality?	<i>High, they use IF to describe in detail data types and operations.</i>
	Testing across levels of abstraction?	<i>Yes, since it uses the abstract test cases to test the systems, the comparison is between abstract test cases and implementation, but not across many levels of abstraction.</i>
	What is the level of abstraction of the specification input?	<i>Design level: state-based specifications, class and object diagrams</i>
	What is the level of automation of the approach?	<i>High.</i>

Table 4 - Evaluation framework applied to AGEDIS.

5.5 Scenario-based Object-Oriented Testing Framework (SOOTF)

The Scenario-based Object-Oriented Testing Framework (SOOTF) [70, 71] is a framework that supports for scenario-based test specification creation, test execution, scenario-based regression testing and results evaluation. It is based on the *End-to-End* (E2E) integration process proposed by the department of defense aiming to address the problem of testing large integrated systems. E2E's idea was implemented by a tool (the E2E testing tool - by the same authors) and incorporated to SOOTF's front-end.

E2E testing tool uses a semi-formal specification of scenarios, named *test scenario specification*. Test scenario specifications are scenarios described in natural language annotated with additional information. They include pre and postconditions, incoming data, expected output, and message sequences (method calls). It can be considered that this approach uses scenarios at different levels of abstraction. Once the test scenario specifications are described, the system supports preparation of test script for execution, execution and evaluation of results.

In the E2E testing tool, scenarios are organized hierarchically into a tree structure (scenario tree, which is a functional decomposition of the system under test) that classifies them as *atomic scenarios*¹¹, *sub-scenarios*, and *complex scenarios*. An *atomic scenario* represents the basic system's function. It is a scenario, with its input data and output result, but describing only **one** atomic function of the system under test. *Complex scenarios* are described by composing the atomic scenarios with some control operators such as sequencing, conditioned, concurrent, and iterative [70]. Additionally, scenarios can relate to each other with the following relationships: independent, trigger/triggered-by, mutually exclusive, concurrent dependency, and related. These relationships guide the way sub-scenarios can be composed into complex scenarios (e.g. a mutually exclusive relationship between two scenarios means they can not be composed into a complex one using the concurrent control operator).

The framework should be used as follows. To create a test scenario specification, the tester first captures the system's behaviors from end user's point of view in the form of a scenario. He or She then transforms the scenarios into test scenario specifications iteratively by identifying inputs and outputs, pre and post conditions, messaging sequence, and dependencies among scenarios. This information should be consistent with the design document (UML use case, class, sequence, state diagrams) and with the implementation. Consistency is achieved by using matching names (when annotating the test scenario specifications, the users use names that match the design documents and implementation). Therefore, relationships are established between high-level test specifications, design documents, and implementation. They are required for script template generation and for test execution. Other relationships required describe dependencies between scenarios. The framework predefines some types of dependencies (functional, input, output, input/output, persistent data, execution, condition, control). Those dependencies are used for performing regression testing.

The framework is object-oriented, and extends the JUnit framework [30] to implement scenario-based testing (as oppose to unit testing). It provides test execution classes templates that support several functions for test scripts generation and execution. It is expected that a one to one map exists from the scenario tree to the testing execution class hierarchical structure. The root of the scenario trees is mapped to the root class in the corresponding class hierarchy, and the sub-groups in the scenario tree are also mapped to the sub-classes in the test execution class hierarchy. These test execution classes are in charge of loading test case data information from a database (previously populated by the user when entering scenario information). The execution classes are also in charge running the test scenarios, and of saving and reporting the results (e.g.

¹¹ Atomic scenario is also referred to as thin threads in other papers by same authors, thin thread is a term defined by the DoD End-to-End Integration Testing project.

number of test cases that failed in the test scenarios). It is worth noting that the framework provides the template for the classes and methods, but the tester should actually implement them, in a similar way to when they use JUnit.

In addition to test scenario specification and management, the framework supports scenario-based regression testing. It uses the relationships created by the user to support this activity. SOOTF implements a scenario-slice algorithm. It receives as input the element that changed in a particular scenario. It searches for other scenarios that might be impacted by that change because they were related to the changed element. The tool supports the user with generating new test scripts, and with identifying the affected scenarios. Since the tool supports the user with manually creating relationships among artifacts (test scenario specifications, requirement items, class/methods, and input/output), all this information is stored in a database for later analysis. Thus, whenever a scenario changes, possibly impacted artifacts are identified.

5.5.1 Application of the Evaluation Framework

Figure 6 below shows the approach supports test generation, execution and evaluation, coverage analysis, and regression testing. Specifications input for the test generation are scenarios at requirements and design levels (more specifically, this approach uses a proprietary semi-formal scenario specification language). This activity (with a fair amount of users' effort) outputs executable test scripts. Execution and evaluation are supported and results are saved for later analysis. Coverage analysis is supported by an integrated tool, which performs the analysis over the obtained test results and relationships between artifacts, as shown in figure 6. The figure also depicts the support provided for regression testing, which uses the relationships among artifacts and the modified specifications (in this case, the modified test scenario specifications). This approach's identifies the affected scenarios, which are represented in the figure by 'Scenarios (subset)'. Table 4 answers the questions from the evaluation framework. Questions that are not applicable to this approach were answered with N/A.

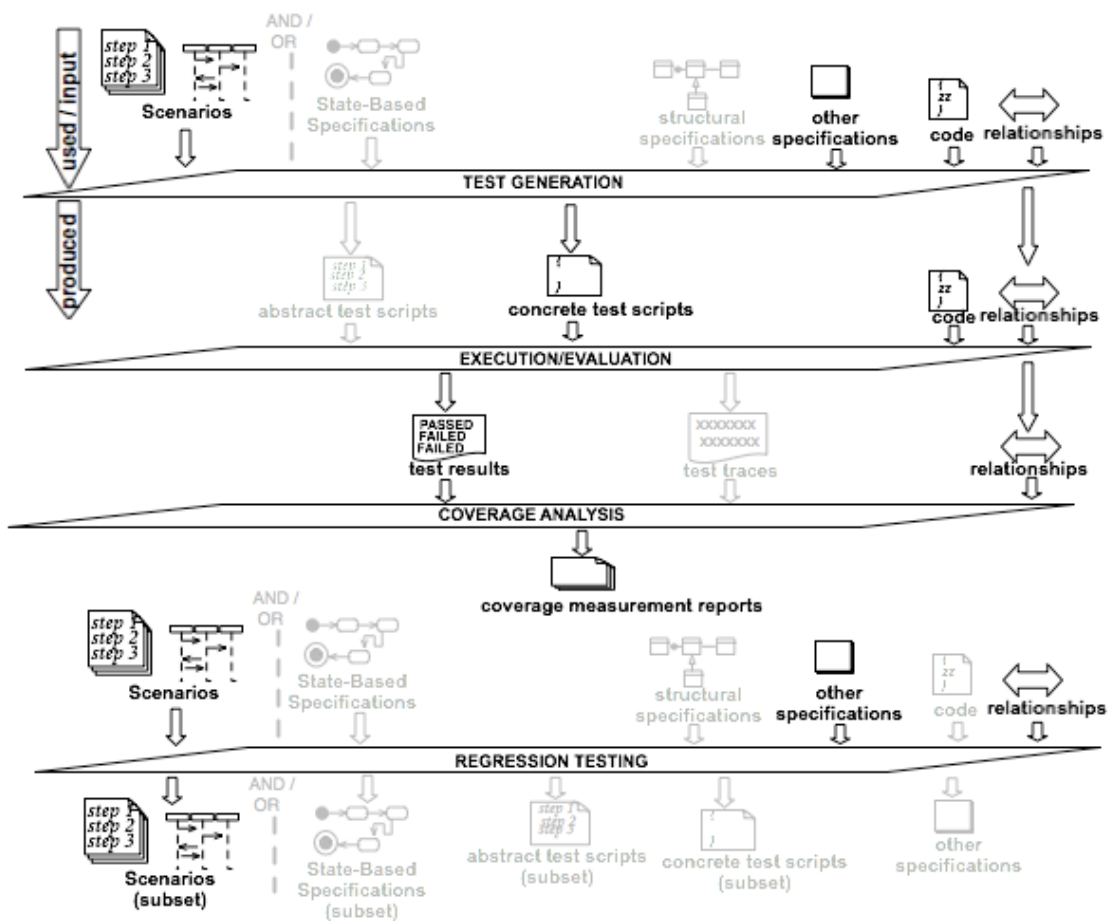


Figure 6 - Technical specification-based testing activities supported by SOOTF

Test Generation	Generates concrete test scripts?	<i>Yes, it provides test execution classes templates that support several functions for test scripts generation and scenario-based testing execution. The user however needs to implement some parts of the script.</i>
	Coverage Criteria?	<i>N/A.</i>
	Kinds of relationships needed to support test generation?	Explicit, Horizontal and Fine-grained. To create a test scenario, the user inputs detailed information that enables the interactive transformation of a scenario into a test scenario. To generate test scripts, a relationship exists between the individual messages in the messaging sequence (in the scenario test specification) and the methods from the generated test script. The relationship is inferred by name matching. It is however made explicit when test scripts templates are generated. Explicit, Horizontal, and Coarse-grained. Also to generate test scripts, the scenarios are organized hierarchically and the test classes' (used by the test scripts) generated also form a hierarchy that corresponds to the scenarios' hierarchy.
	Support for management of these relationships?	<i>Yes, the support is provided for manual management. The user who is in charge of creating the test scenarios specifications, is also in charge of managing some of these relationships, with the exception to the relationships inferred by name.</i>
Test Execution and Evaluation	Specification used as oracle?	<i>Yes, the test execution classes implemented for each scenario at the scenario tree, has methods to check postconditions and expected outputs. The behavior, however, is not compared step by step.</i>
	Automated evaluation?	<i>Yes, it is automatic, but the evaluations are: comparison between expected results and obtained results, and post condition check. The actual behavior is not considered.</i>
	Kinds of relationships needed to support execution and evaluation?	Explicit, Horizontal and Coarse-grained relationships. When creating the test scenario specification, the user describes post-condition for the whole scenario, which describes the expected state of the whole system after the execution of the scenario. The test script uses the postconditions to evaluate the results of the test run. Implicit, Horizontal and Fine-grained. They are used to execute the test scripts. They are relationships from the concrete test scripts to the code it tests.
	Support for management of these relationships?	<i>Yes, manually, by the tool's user who is in charge of creating the test scenarios specifications.</i>
Coverage Analysis	Support for coverage analysis?	<i>Yes, in addition to the coverage analysis that outputs reports regarding the coverage achieved by the tests execution in relation to the requirements items, to the scenarios, to the components and so forth, the framework is also integrated with a statistical model tool that evaluates test performance (Assurance-Based Testing - ABT) and provides information on the level of confidence achieved by the test scripts run.</i>
	Kinds of relationships needed to support coverage analysis?	Explicit, Coarse-grained, Vertical. The information on the test scenarios is also related to the design document (UML use case, class, sequence, state diagrams) and to the implementation using their names.
	Support for management of these relationships?	<i>Yes, partially. As explained, the relationships across artifacts can be manually changed by the users, while the history results is persisted and available for analysis.</i>

Regression Testing	Support for regression testing?	<i>Yes, they support scenario-based regression testing using scenario-slice algorithm to identify other entities (e.g. other scenarios.) that might have been impacted by a change to a scenario. This information is also obtained from the test scenario specification, which describes dependency relationships between scenarios.</i>
	Specification Language?	<i>Scenario - they use their proprietary scenario template, described with a semi-formal specification language.</i>
	Level of formality?	<i>Medium.</i>
	Testing across levels of abstraction?	<i>No.</i>
	What is the level of abstraction of the specification input?	<i>Requirements and design: scenarios at requirements level scenarios that are refined and annotated with design-level information, and UML use cases, sequence diagrams, object and class diagrams.</i>
	What is the level of automation of the approach?	<i>High, although it requires a large amount of input from the user.</i>

Table 5 - Evaluation framework applied to SOOTF.

5.6 UMLTest

UMLTest is proof-of-concept test data generation tool that is integrated with Rational Rose case tool. This tool extends another tool named SpecTest [52, 53, 54, 55]. SpecTest provides support for generating *test requirements*¹² based on appropriate criteria. It also provides support for generating actual test values, but does not support test scripts generation. Thereby, although it provides some automation with respect to generation of test specification, its main contribution is the definition of test criteria at the specification level (criteria to reach specification coverage as opposed to code coverage). Differently from the other approaches evaluated by this survey that concentrate mainly on trying to automate test activities (and generate scripts as well as execute and evaluate the results).

SpecTest accepts as input Software Cost Reduction (SCR) specifications, while UMLTest accepts UML state diagrams. Both kinds of specifications should be previously created using specific tools (SCRTTool by the Naval Research Laboratory and Rational Rose tool by IBM, respectively). The tools are capable of parsing such specifications and transforming them into a general specification graph. This graph is used as the basis for generation of the test requirements and test values. This specification graph is a state-based directed graph where nodes represent states and edges represent the possible transitions among states. The user is given four options of coverage criteria he or she wants to use. The tool uses the selected criteria to output test requirements for testing. Test requirements are truth assignments to the predicates used as guards to the transitions in the graph.

The criteria supported are the following: transition coverage, full predicate coverage, transition-pair coverage, complete sequence coverage. Transition coverage states that the test set must include tests that cause every transition on the specification graph to be taken. Full predicate coverage states that the test set must include tests that cause each clause in a predicate to determine the value of the whole predicate, for each predicate on each transition on the specification graph. Transition-pair coverage states that the test set must include tests that cause every pair of adjacent transition to be traversed in sequence (e.g. if in the graph there are transitions A – B and B – C, there is a test that traverses both transitions in sequence). Complete sequence coverage states that the test set must include tests that traverse meaningful sequences of transitions, based on the test engineers' experience. This last criterion is not automated or measurable because in most of the cases the number of possible sequences is way too large to be covered.

The next step on the use of the tool is to actually generate the test specifications for each unique test requirement. The test specifications generated consist of prefix and test case values, verification conditions, exit conditions and expected outputs¹³. The following step would be to construct one test script for each generated specification. This construction is not automated by this approach, and requires knowledge about the implementation.

Another related work by the same authors is an approach to generate runtime information from design level UML collaboration diagrams. UML Collaboration diagrams are a variation of UML sequence diagrams (they can both be named interaction diagrams). They are also used at different levels of abstraction. If used to realize use cases, they are considered as a variation of

¹² *Test requirements* are things that must be satisfied (or covered), e.g. to satisfy the statement coverage criteria, the requirement is that the statements should be reached.

¹³ Definitions: Prefix values are inputs necessary to reach the pre-state and to give the triggering events variables their before-values; Test case values can be commands, user inputs or software function and values for its parameters; Verify conditions are inputs that might be necessary to show the results; Exit conditions are commands that may be needed to terminate execution, and expected outputs are created from the after-values of the triggering events and any post-conditions associated with the transition.

scenario at the requirements level. If used describe object interactions, they are considered as variations of scenarios at the design level. In this related work, the authors describe an approach that supports static checking and dynamic testing. Static checking checks that some pre-identified constraints are not violated. Dynamic testing means that the tool derives message sequence paths from the collaboration diagram, based on which it instruments the implementation to support the user on keeping track of runtime interactions.

5.6.1 Application of the Evaluation Framework

Figure 7 below shows that this approach partially supports test generation. The specification input is state-based specification (more specifically, UML state machine diagrams). Table 6 answers the questions from the evaluation framework. Questions that are not applicable to this approach were answered with N/A.

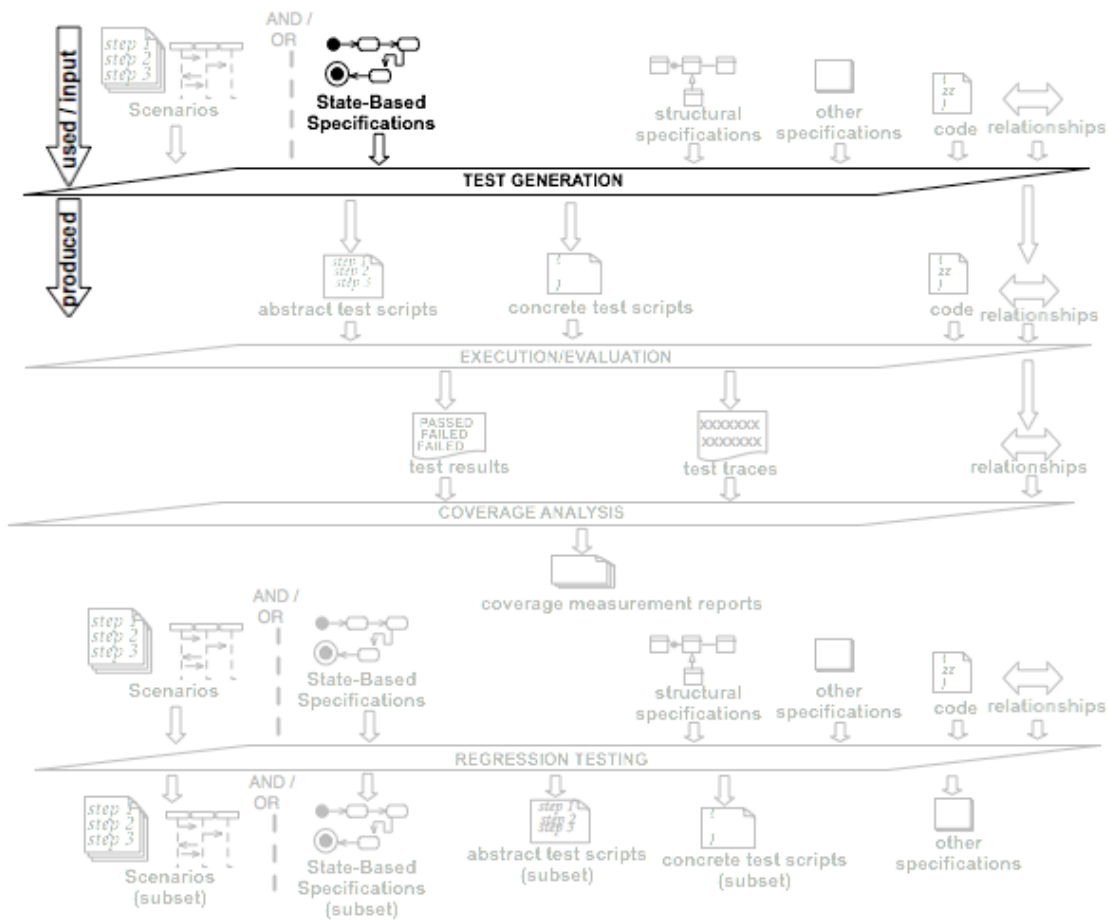


Figure 7 - Technical specification-based testing activities supported by UMLTest.

Test Generation	Generates executable test script?	No.
	Coverage Criteria?	<i>There are four coverage criteria options based on the specification, this is the main contribution of the approach and was explained in the previous section.</i>
	Kinds of relationships needed to support test generation?	<i>N/A. The tool generates test values by solving algebraic equations, those test values are part of the test specification output by the tool, but this generation does not require existence of relationships among artifacts. Once the test specifications had been generated, each one is used to generate one test script; this requires a mapping from the specification variables to the program variables and had not been automated by this approach.</i>
	Support for management of these relationships?	<i>N/A.</i>
Test Execution and Evaluation	Specification used as oracle?	<i>No, because the test specification generated by the approach has information about expected output, but the comparison itself is not automated.</i>
	Automated evaluation?	<i>No.</i>
	Kinds of relationships needed to support execution and evaluation?	<i>N/A.</i>
	Support for management of these relationships?	<i>N/A.</i>
Coverage Analysis	Support for coverage analysis?	<i>No.</i>
	Kinds of relationships needed to support coverage analysis?	<i>N/A.</i>
	Support for management of these relationships?	<i>N/A.</i>
Regression Testing	Support for regression testing?	<i>No.</i>
General Aspects	Specification Language?	<i>UML state machine diagrams.</i>
	Level of formality?	<i>High, they use state machines with OCL.</i>
	Testing across levels of abstraction?	<i>No.</i>
	What is the level of abstraction of the specification input?	<i>Design level: state-based specifications.</i>
	What is the level of automation of the approach?	<i>Low.</i>

Table 6 - Evaluation framework applied to UMLTest.

5.7 Abstract state machine Language (AsmL) Test Tool

Abstract State Machine Language (AsmL) [1] is an executable specification language with foundations on the theory of Abstract State Machines¹⁴ [35]. It is integrated into the .NET framework, and also into Microsoft development tools. The test tool realizes a semi-automatic approach that mainly support test generation, execution and evaluation. The test generation activity is composed of two main features: parameter generation and sequence selection. Parameter generation selects the set of input values for the test scripts. Sequence selection selects the steps of the test script. Test execution and evaluation exercises the code guided by the generated test scripts and compares the execution of the implementation execution to the execution of the specification.

AsmL test tool receives as input AsmL models annotated with information for test generation, which is named *model program*. An AsmL *model program* is either an XML and/or Microsoft Word document. The model program is the basis for test case generation. It is also used as the oracle.

As part of the test generation feature, AsmL uses the model program to generate finite state machines (FSM)¹⁵. This is done because of the existent support for FSM, which includes a well-established automata theory, efficient graph algorithms, and test generation tools. AsmL test tool integrates with these algorithms and tools. FSM are not used directly because they are not always the most natural modeling method to describe an implementation one needs to test [33]. The following paragraph describes the major features supported by the tool [4] and how they should be used.

To generate the FSM from the model program [32], the generation algorithm processes it following some criteria expressed as *control items*. The control items are created by the user as part of a configuration of the model program [33]. The main purpose of these control items is to reduce the number of states included in the generated FSM. Therefore, only states relevant from a certain testing standpoint (this testing standpoint can be for example a situation the user wants to test) are included. The control items are the following.

Variables are state variables that constitute the ASM states. The user will not want to include variables that are not pertinent to the states he wants included in the final FSM (e.g. some variables are used for logging purposes and should not be included in this selection);

Actions are methods in the model program that appear as state transition in the FSM. Some actions are events, which in this context means “an action that cannot be invoked by the tool, only observed”. The user will not want to include actions that are not pertinent to the test scenarios of interest;

Properties are expressions that aim to define equivalence relation between states of the model program. A collection of equivalent states is named hyperstates. The generator uses the properties to limit the number of states included in the FSM (by for example, including only states the define a new hyperstate)¹⁶.

Filters are Boolean expressions also used to limit the number of states by filtering out undesired states. It is recommended to include some filter that ensures that state exploration terminates.

¹⁴ See glossary.

¹⁵ See glossary.

¹⁶As described in 21 “The ASM may have too many, often infinitely many, states. To this end, we group ASM states into finitely many hyperstates. This gives rise to a finite directed graph or finite state machine whose nodes are the generated hyperstates”.

The sequence selection feature uses the generated FSM to derive the test scripts. The FSM is viewed as a graph where edges are marked as required or optional, and are also annotated with the cost of executing the action. The test generation algorithm (that is a combination of existing graph exploration algorithms) makes sure that every state is reachable by following required links only. Thus, the objective is to generate test scripts (sequences of actions that start from the initial state) that provide coverage of the required links with the minimal total cost. These generated test scripts are represented and persisted as XML. They are comprised of sequences of actions, their input parameters values and expected outputs (created with the parameter generator feature¹⁷).

The execution and evaluation feature uses a configuration that describes the relationship between the AsmL model and the implementation. This configuration is named *conformance binding*. The tool supports the user to interactively determine associations between items in the model and items in the implementation. It uses this information to determine which method in the implementation to execute when an action (method) from the model program is executed (the model and the implementation are executed in parallel). The tool automates the creation of associations between methods (in the implementation) and actions (in the model) whose signatures match. It also automates the creation of associations between global constants whose names match. This is only possible if the model program's namespace is different from the implementation's namespace. If the model program uses classes, the user needs to associate classes in the model to classes in the implementation.

The comparison between execution of the implementation and execution of the AsmL model uses another configuration named *conformance relation*. It is used to compare the states of the model to the states of the implementation. Conformance relation is defined as a Boolean expression that relates their variables. The expression is evaluated after each action execution. Methods return values are compared for equality, and thus, do not require such configuration.

The former paragraphs explained how AsmL test tool works. This same approach [11, 34] was extended to support scenario-oriented models (as well as use cases¹⁸) described using AsmL. In other words, that approach allows the use of the sequence notation of AsmL to programmatically describe scenarios. It supports the same features described above, which are test generation (parameter generation and sequence selection), test execution and evaluation. There are, however, some differences in the configurations the users create. The user needs to map the (AsmL) use case concept into an ASM, because ASM are the kind of model accepted by this tool.

¹⁷ Note that parameter and sequence generation can happen be executed at the same time as the FSM generation, but for the sake of clarity they are explained separately.

¹⁸ A use case is a collection of scenarios and the approach supports the codification of a set of scenarios into a use case, where parameterized scenarios are defined as non-parameterized scenarios capable of choosing the parameter from a specific domain. Their representation of use cases does not explicitly attach actors and roles to the events, and is basically a sequence of interactions: actors and actions performed by these actors with some variables used to bind parameters to actions.

5.7.1 Application of the Evaluation Framework

Figure 8 below shows that this approach supports test generation, and execution and evaluation. The specification input for the test generation is either a state-based specification, or scenarios at the requirement level (and use cases) annotated with further programmatic information. More specifically, this approach uses sequence notation of AsmL to programmatically describe scenarios. Additional information needed for sequence and parameter generation is represented as ‘other specifications’ in figure 8. Test generation activity generates abstract test scripts. They are input to the evaluation and execution activity, which requires mapping relationships for executing the test scripts. It also requires other relationships for evaluating the results. Table 7 answers the questions from the evaluation framework. Questions that are not applicable to this approach were answered with N/A.

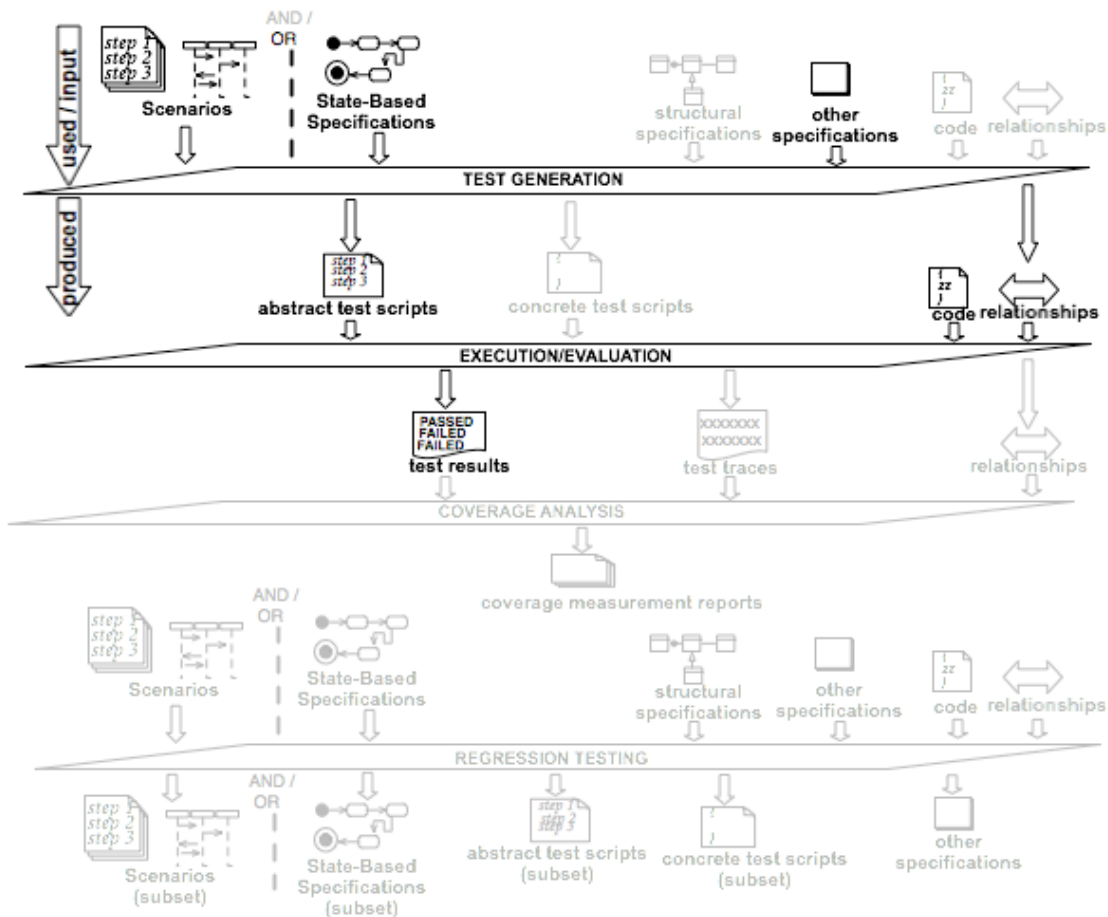


Figure 8 - Technical specification-based testing activities supported by AsmL

Test Generation	Generates concrete test scripts?	<i>No, the approach generates test sequences based on the model program (or on the use case). These can be simulated, but not directly used to exercise the system.</i>
	Coverage Criteria?	<i>The user is supposed to configure the following: a filter to filter-out undesired states, equivalence of the states (to be grouped into hyperstates). The user should also configure the domains to avoid many combinations of parameters.</i>
	Kinds of relationships needed to support test generation?	<i>N/A. The test generation feature traverses the FSM derived from the AsmL model. It outputs test sequences (sequences of actions, their input parameters values and expected outputs). Since they are not at the code level, relationship between model and implementation is not required for this particular activity.</i>
	Support for management of these relationships?	<i>N/A.</i>
Test Execution and Evaluation	Specification used as oracle?	<i>Yes, the model is used as oracle during conformance testing.</i>
	Automated evaluation?	<i>Yes, it is automatic, the model and the implementation are executed in parallel and their states are compared.</i>
	Kinds of relationships needed to support execution and evaluation?	Explicit, Vertical and Fine-grained. They are defined in the “conformance binding” to support the parallel execution of the model program and the implementation. They relate methods and actions whose signatures match. They also relate global constants in the model and in the implementation. Additionally, they are defined in the “conformance relationship” to support comparison of states of the model to the states of the implementation. These are Boolean expressions that relate variables in the model and in the implementation.
	Support for management of these relationships?	<i>Yes, manually, by the tool’s user who is in charge of creating the project (with the relationship information). The whole project is persisted; so are the relationships, but in the event of a change to the endpoints of a relationship (e.g. change to the name of a class in the model), the conformance binding should be changed to reflect it (the user would have to rebind the class in the model to the class in the implementation).</i>
Coverage Analysis	Support for coverage analysis?	<i>No, although the results of each test sequence run is displayed to the user, no reference is made on the documentation about persisting them, maintaining them or keeping track of a collection of use cases and which test sequences runs relates to which test case.</i>
	Kinds of relationships needed to support coverage analysis?	<i>N/A.</i>
	Support for management of these relationships?	<i>N/A</i>
Regression Testing	Support for regression testing?	<i>No.</i>
General Aspects	Specification Language?	<i>The proprietary language named AsmL (Abstract State Machines Language) it supports descriptions of programs in many levels of abstraction. In this approach it can be used for both annotated requirement-level scenarios and state-based models.</i>
	Level of formality?	<i>High (Its a programming language).</i>
	Testing across levels of abstraction?	<i>Yes, at least across the model and the implementation, with the conformance testing.</i>
	What is the level of abstraction of the specification input?	<i>Requirements and design: requirements-level scenarios (or their definition of use case) are annotated with design-level information; design-level state-based models are also input.</i>
	What is the level of automation of the approach?	<i>Medium.</i>

Table 7 - Evaluation framework applied to AsmL.

5.8 Testing Object-oriented systems (TOTEM)

Testing Object-oriented systems with the unified Modeling language (TOTEM) [16] is a methodology and tool that supports system testing based on UML models (Use Case, Sequence, Activity and Class diagrams). In particular, this approach supports test generation, while an extension of this work supports regression testing also based on UML diagrams. Similar to other UML-based testing approaches, the creation of the diagrams is done using a UML case tool. The diagrams are exported to a common representation and imported by TOTEM.

The approach assumes that each use case is related to one sequence diagram, which represents the use case realization through object interactions. It also assumes that OCL is used to describe classes' invariants and methods' preconditions and postconditions. The approach relies on system test requirements to generate test sequences. These requirements are created based on the available UML diagram.

The first test requirement describes use cases sequences (order in which use cases should be tested). UML activity diagrams describe the dependencies between use cases. The diagram is traversed using a depth-first search, which results in use case sequences (or paths in the activity diagram). Data dependencies are also considered. They are determined based on symbolic values from the activity diagram. This approach uses the concept of parameterized use cases. This concept allows the use case sequences to be instantiated several times with different symbolic values for the purpose of testing. The amount of instantiation is determined by configurations such as the number of users expected to the system provided by the tester. The formal parameters of the corresponding use cases are later analyzed to generate actual values. The sequences generated are combined (by interleaving) to generate complete sequences to be tested. This step is needed because use case path derivation from activity diagrams does not account for synchronization, so two different derived paths can occur concurrently.

The second test requirement identifies use case scenarios. The sequence diagrams are transformed into regular expressions that use as alphabet the public methods (also called operations) from the sequence diagrams. This transformation aims at expressing the diagram in a compact and analyzable way. The regular expressions are expressed as a composition of terms, where each term represents a single use case scenario or a set of use case scenarios. Since the regular expression can contain iteration symbols, they are further analyzed to identify test conditions under which their terms can be executed, and to identify the precise sequence of operations to be executed for each term.

The test scripts generated also have oracles. They are derived using the postconditions (described as OCL expressions) of operations. The conjunction of post-conditions is used to determine the test oracles of the operations sequences. This solution works for simple cases where the post-conditions clauses do not interfere one in the other. The tool does not address these more complex cases (it is left for future work). Once this information is available for one use case, their initial state, the operation sequences, and the oracles are formalized into a decision table used as the formal set of test requirements. Finally, these decision tables created for each use case are used in conjunction to derive the sequences of operations that test an entire use case sequence (as originally derived from the activity diagrams).

The former paragraphs describe how this approach generates test scripts. As part of the same project, the authors created the Regression Test Selection tool (RTSTool) [19, 17, 18]. This tool implements an approach for regression testing based on UML diagrams and relationships among these diagrams. The tool receives as input UML use case, class and sequence diagrams. These diagrams are also expressed in the common format and imported by RTSTool. This activity is supported by relationships between flows in a sequence diagram and the test scripts previously generated. The approach considers that particular changes are possible to class diagrams, use case diagrams and sequence diagrams. Depending on these changes, the test scripts traced from these diagrams are grouped into obsolete (cannot be executed on the new version of

the system), re-testable (is still valid, but need to be re-run) and reusable (still valid and do not need to be rerun).

5.8.1 Application of the Evaluation Framework

Figure 9 below shows that this approach supports test generation, and regression testing. The specifications input for the test generation are scenarios at the design level (sequence diagrams), class diagrams (represented in the figure by ‘structural representation’), use case diagrams, and activity diagrams (the last two represented in the figure by ‘other specifications’). Additional information needed from the user regarding test configuration is also represented in the figure by ‘other specifications’. This activity outputs abstract test scripts. The same artifacts are input to the regression testing activity. Regression testing also requires relationships from the design models to the test cases (represented as the arrow in the figure). Table 8 answers the questions from the evaluation framework. Questions that are not applicable to this approach were answered with N/A.

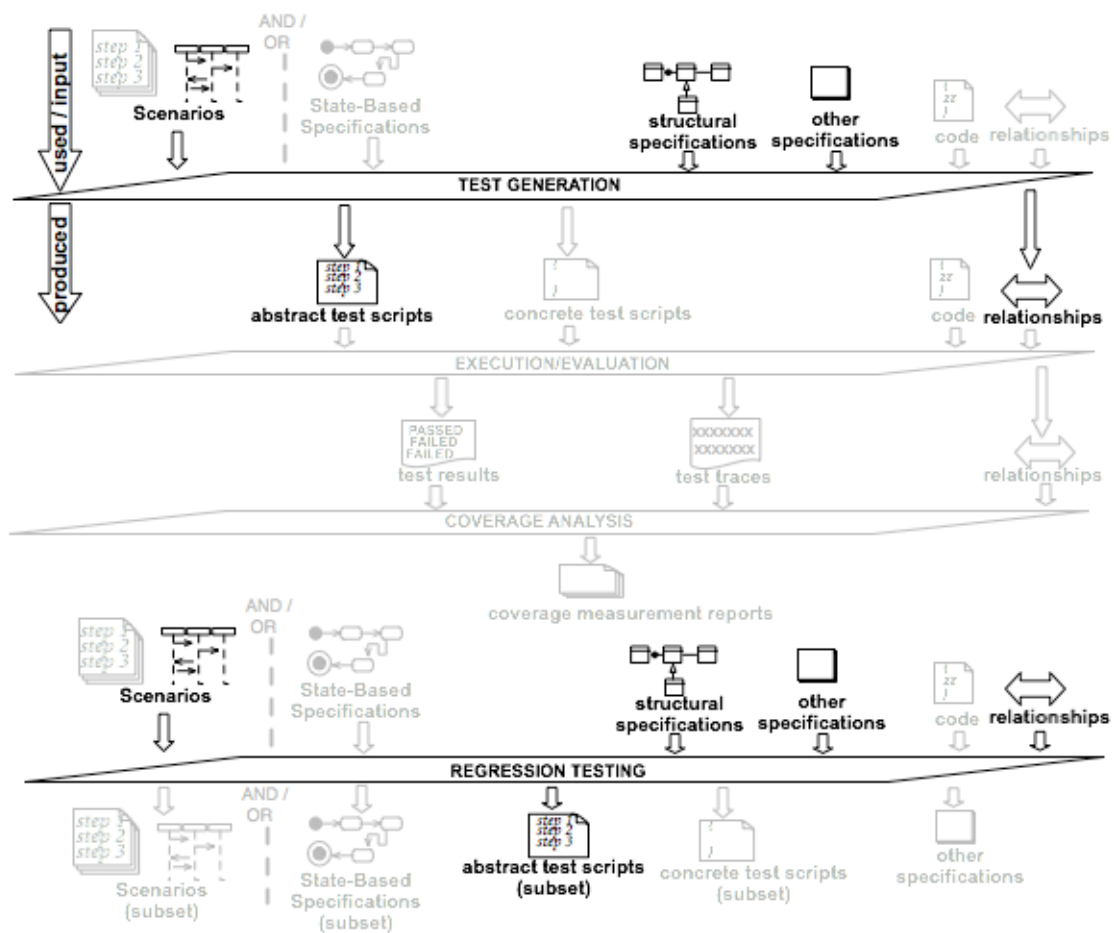


Figure 9 - Technical specification-based testing activities supported by TOTEM

Test Generation	Generates executable test script?	<i>No, the approach generates test sequences based on the on the use cases and sequence diagrams.</i>
	Coverage Criteria?	<i>Yes, to limit the amount of to instantiated the test sequences, it uses activity diagrams, sequence diagrams, class diagrams with OCL expressions, and information provided by the tester regarding expected input values.</i>
	Kinds of relationships needed to support test generation?	Implicit, Horizontal and Coarse-grained. They are obtained from the imported UML model. They relate diagrams at the same level of abstraction (use cases in the use case diagram are related to use cases in the activity diagram). Implicit, Vertical and Coarse-grained. They relate diagrams at the different levels of abstraction (each use case is realized by one sequence diagram).
	Support for management of these relationships?	<i>No.</i>
Test Execution and Evaluation	Specification used as oracle?	<i>Yes, the model is used to derive oracles from post-conditions of operations written with OCL.</i>
	Automated evaluation?	<i>No.</i>
	Kinds of relationships needed to support execution and evaluation?	<i>N/A.</i>
	Support for management of these relationships?	<i>N/A.</i>
Coverage Analysis	Support for coverage analysis?	<i>No.</i>
	Kinds of relationships needed to support coverage analysis?	<i>N/A.</i>
	Support for management of these relationships?	<i>N/A</i>
Regression Testing	Support for regression testing?	<i>Yes, the approach supports regression testing based on use case, class, sequence diagrams and relationships among those diagrams (obtained from UML model imported by the tool), and between those diagrams and generated test scripts.</i> <i>Relationships are Implicit, Fine-grained and Horizontal (an object in a sequence diagram is related to its class in the class diagram, every method call in that object is related to a declared method in its class in the class diagram, a sequence diagram is related to the test scripts it generated). Relationships are also Implicit, Coarse-grained and Vertical (each use case in the use case diagram is realized by one sequence diagram, each use case is also related to the test scripts it generated.).</i>
General Aspects	Specification Language?	<i>UML (sequence diagrams, activity diagrams) and OCL.</i>
	Level of formality?	<i>Medium.</i>
	Testing across levels of abstraction?	<i>No.</i>
	What is the level of abstraction of the specification input?	<i>Requirements and design: use case diagrams, design-level sequence diagrams.</i>
	What is the level of automation of the approach?	<i>Medium.</i>

Table 8 - Evaluation framework applied to TOTEM

5.9 UMLAUT/Simulator

UMLAUT [60, 44] is a case tool for manipulation and transformation of UML models. It was extended with the incorporation of Test Generation with Verification (TGV) tool, which is a tool for test generation of conformance test suites from specifications of reactive systems. TGV receives as input the specification of the system under test (SUT) and the *test purposes*. It outputs *abstract test cases* that aim at covering the test purposes. An *abstract test case* describes the interactions between the tester and the SUT. These interactions describe input and output to and from the system, which leads to verdicts about the test (pass, fail, inconclusive). TGV supports different (Lotos, SDL, UML, IF) specification and test purpose languages by connecting to their simulation APIs. Particularly for languages with semantics in terms of LTS or IOLTS, if a compiler produces a simulation API, an interface between this API and the TGV API can be built easily.

The support for UML is provided by UMLAUT. It is assumed that the specification of the system includes the class diagrams, state machine diagrams for each class, and also a description of its initial state in the form of UML object diagram. This model is used by UMLAUT to derive a Label Transition System (LTS) representing the operational semantics of the specification of the entire system.

The test purposes are expressed as scenarios, and they are described using the scenario-based language named O-Tela, which is based on UML sequence diagrams. It makes use of UML activity and class diagrams. In fact, the initial model of the test purposes can be created using some UML case tool. It can be saved to the common format (XMI), and then it can be imported by UMLAUT. The test purposes are also formalized as Label Transition System (LTS). It is important to note that two kinds of scenarios are provided as part of the test purposes: accept and reject scenarios. Accept scenarios describe scenarios relevant to the tester that should be considered for test case generation. While reject scenarios describe scenarios the tester wants to avoid. Therefore, the test generation process is driven by the expected scenarios, which are indeed used to reduce the space explored during test generation by eliminating calls known to be superfluous for the purposes of the test. The system specification and the test purposes both represented as LTS are then used by TGV to produce abstract test cases in the form of IOLTS¹⁹. These abstract test cases are then transformed into scenarios, described with another scenario-based language named TeLa, which is also based on UML sequence diagrams. UMLAUT also supports the transformation of abstract test cases in TeLa to XMI. This allows the scenarios to be visualized with any case tool that understands XMI.

¹⁹ See definition in section 9.

5.9.1 Application of the Evaluation Framework

Figure 10 below shows that this approach supports test generation. The specification input for the test generation is design-level scenarios (sequence diagrams), and UML class, state machine and activity diagrams. The design level scenarios are in fact described with this approach's particular language called O-Tela, so any additional information needed from the user regarding test configuration is represented in the figure by 'other specifications'. The two-way arrow represents relationships across UML models. This activity outputs abstract test scripts. Table 9 answers the questions from the evaluation framework. Questions that are not applicable to this approach were answered with N/A.

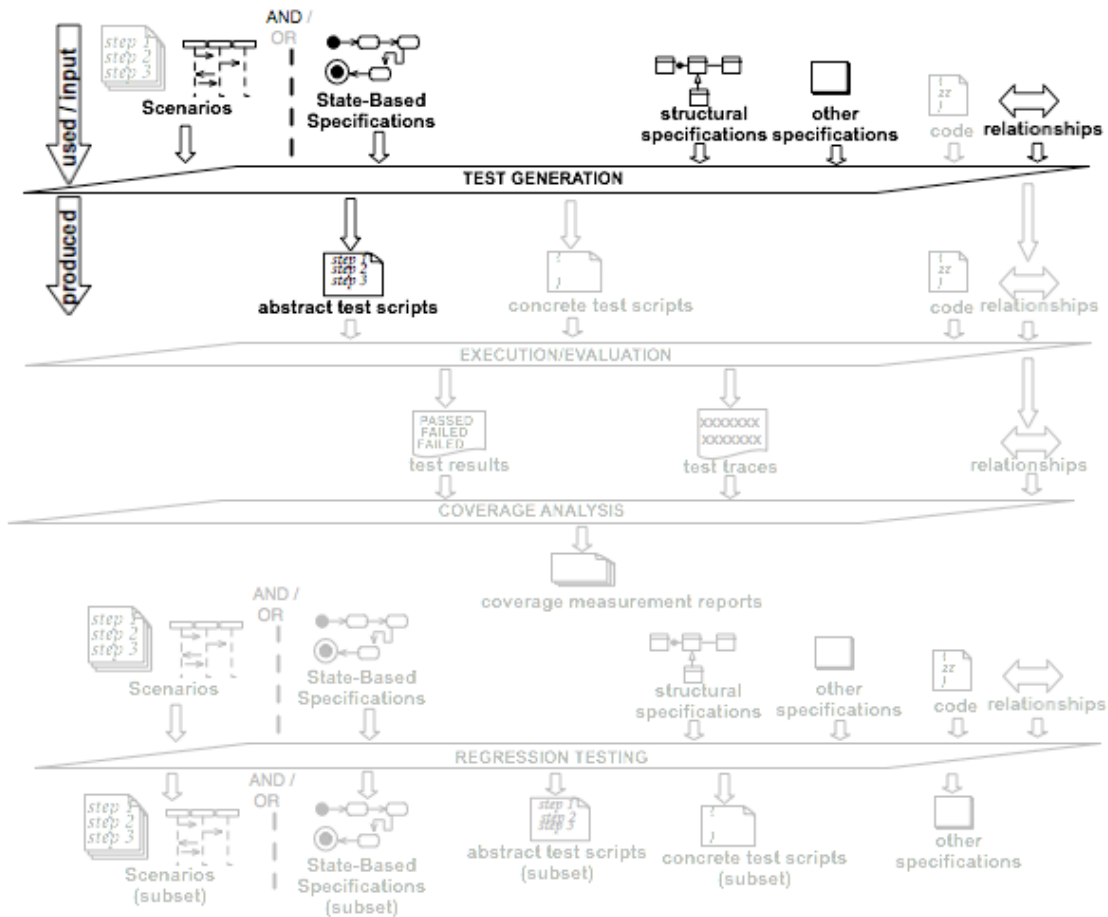


Figure 10 - Technical specification-based testing activities supported by UMLAUT/Simulator

Test Generation	Generates concrete test scripts?	<i>No, the approach generates test sequences based on the on the sequence diagrams and UML models.</i>
	Coverage Criteria?	<i>Yes, the approach generates test cases based on predefined test purposes (written with O-Tela, in the form of scenarios) and UML models. The test purposes can be considered coverage criteria because they actually reduce the size of test suites to those described as accept/reject scenarios.</i>
	Kinds of relationships needed to support test generation?	<i>Implicit, Horizontal, and Coarse-grained.</i> Relationships are obtained from the UML model imported by the tool (each class in a class diagram is related to a state machine diagram). <i>Implicit, Horizontal and Fine-grained.</i> Since the approach transforms UML specifications into LTS, it assumes relationships between state machines (from the UML specification) and LTS. It also transforms test objectives in (sequence diagrams O-Tela) into LTS, so it also assumes relationships between sequence diagrams O-Tela) and LTS. The language allows sequence diagrams to refer to class diagrams from the UML specification of the system.
	Support for management of these relationships?	<i>No.</i>
Test Execution and Evaluation	Specification used as oracle?	<i>N/A.</i>
	Automated evaluation?	<i>No.</i>
	Kinds of relationships needed to support execution and evaluation?	<i>N/A.</i>
	Support for management of these relationships?	<i>N/A.</i>
Coverage Analysis	Support for coverage analysis?	<i>No.</i>
	Kinds of relationships needed to support coverage analysis?	<i>N/A.</i>
	Support for management of these relationships?	<i>N/A.</i>
Regression Testing	Support for regression testing?	<i>No.</i>
General Aspects	Specification Language?	<i>UML, O-Tela, and Tela.</i>
	Level of formality?	<i>High.</i>
	Testing across levels of abstraction?	<i>No.</i>
	Specification Language?	<i>UML, O-Tela, and Tela.</i>
	What is the level of automation of the approach?	<i>Low.</i>

Table 9 - Evaluation framework applied to UMLAUT/Simulator

5.10 Test Sequence GeneraTOR (TESTOR)

TEst Sequence GeneraTOR (TESTOR) [59] is a test sequence generator algorithm that extracts test sequences from both state machines and scenario diagrams. It is different from the other approaches evaluated that concentrate on either one or the other. This approach focuses on model-based testing of component-based systems at the integration level. It assumes it receives as input the structural and behavioral specification of the components. Test sequences extracted could then be used for execution and evaluation. Therefore, the sequences can be used as oracles.

The algorithm was developed to address industry needs not addressed by other approaches. The addressed needs are as follows. Approaches should be usable, meaning that no further formalism other than the available specification is required. They should also be timeliness, meaning that even incomplete models can be used to outline testing plans. Additionally, approaches should have tool support. As discussed throughout this paper, tool support is fundamental to reduce test costs.

Using both sequence and state diagrams, addresses these three needs. The state diagrams are used to recover missing information from the sequence diagrams that are possibly incomplete. The main idea is that they use both kinds of specification to output a more informative scenario. The algorithm receives as input state machines and scenarios, both at the architectural level of abstraction. The sequence diagrams are used to guide the selection of paths in the state machine (*test purposes*, or *test generation directives*). The state machines are individually simulated so no composition is required in this algorithm, which avoids state explosion.

The algorithm works as follows. For each message in a sequence diagram, the algorithm searches the state machines to find the paths starting from the current state that include this message. Once the paths for that message are found, the algorithm performs the same search for the next not 'visited' message. Note that in the beginning the current state of the state machines is the initial state. This procedure is done recursively until all the messages in a sequence diagram have been visited. The result is a set of message traces. The algorithm then merges these sequences checking if the components can synchronize. Test sequences in the form of scenarios at the architecture level are outputted.

TESTOR was implemented as a plug-in to a tool named CHARMY. It supports creation of software architecture structure and behavior with state machines. Additionally, it supports the creation of sequence diagrams as components interactions.

5.10.1 Application of the Evaluation Framework

Figure 10 below shows that this approach supports test generation. The specification input for the test generation are scenarios at the architecture level, and state diagrams also at the architecture level. This activity outputs non-executable test scripts at the level of architecture. Table 9 answers the questions from the evaluation framework. Questions that are not applicable to this approach were answered with N/A.

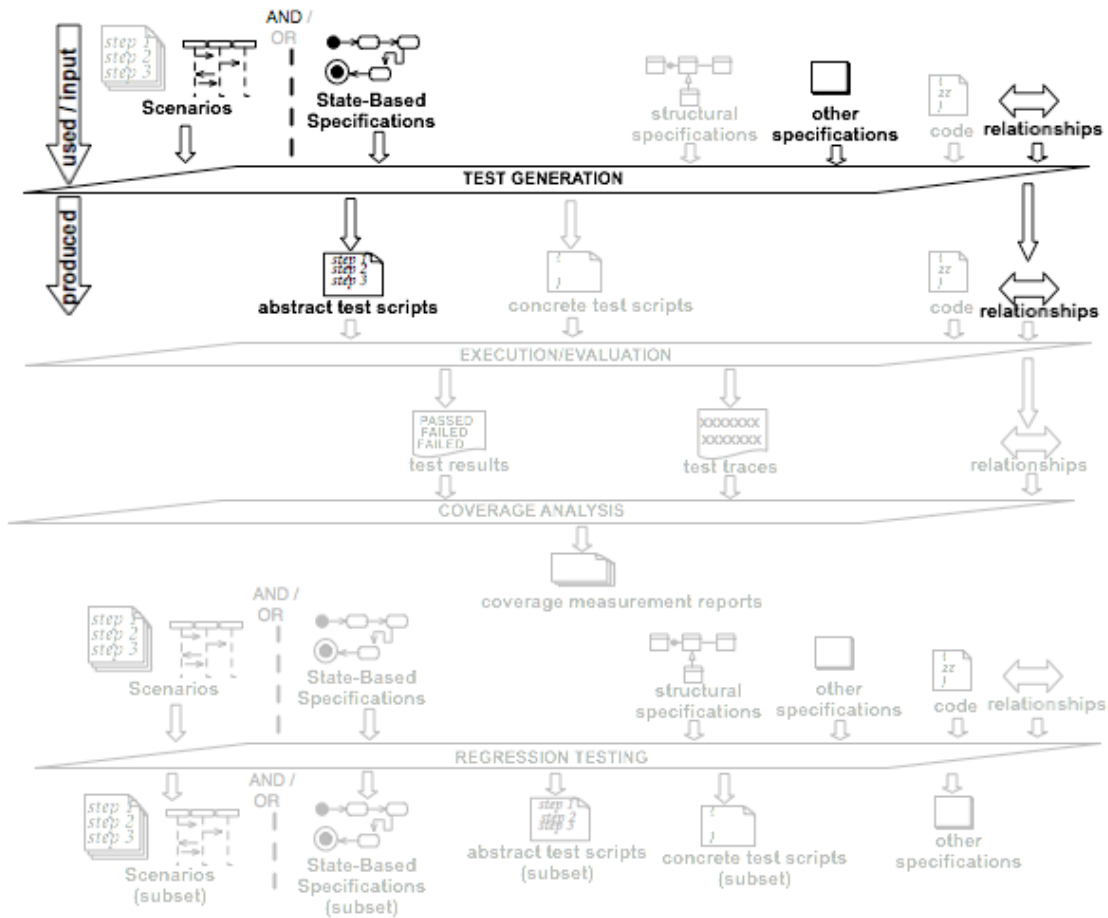


Figure 11 - Technical specification-based testing activities supported by TESTOR

Test Generation	Generates concrete test scripts?	<i>No, the approach generates abstract test sequences that describe component interactions.</i>
	Coverage Criteria?	<i>Yes, the approach uses sequence diagrams to guide the selection of paths in the state machine diagrams</i>
	Kinds of relationships needed to support test generation?	<i>Implicit, Horizontal, and Coarse-grained.</i> They are obtained from the models created with CHARMY tool. They relate components participating at a sequence diagram and the same component's state machine <i>Implicit, Horizontal and Fine-grained.</i> They relate messages in the sequence diagrams to events in the state machines.
	Support for management of these relationships?	<i>No.</i>
Test Execution and Evaluation	Specification used as oracle?	<i>Yes, the test sequences generated have the expected response to the stimuli in the sequences.</i>
	Automated evaluation?	<i>No.</i>
	Kinds of relationships needed to support execution and evaluation?	<i>N/A.</i>
	Support for management of these relationships?	<i>N/A.</i>
Coverage Analysis	Support for coverage analysis?	<i>No.</i>
	Kinds of relationships needed to support coverage analysis?	<i>N/A.</i>
	Support for management of these relationships?	<i>N/A.</i>
Regression Testing	Support for regression testing?	<i>No.</i>
General Aspects	Specification Language?	<i>UML (at the level of architecture).</i>
	Level of formality?	<i>High.</i>
	Testing across levels of abstraction?	<i>No.</i>
	What is the level of abstraction of the specification input?	<i>Architecture-level sequence and state machine diagrams.</i>
	What is the level of automation of the approach?	<i>Low.</i>

Table 10 - Evaluation framework applied to TESTOR

5.11 UC-SCSystem

The UC-SCSystem [49] is a prototype tool that supports test case generation from use cases. It was developed as part of the Triskell project, which is research team composed of researchers from some universities in France (CNRS, Université Rennes 1, INRIA INSA). It is based on the idea of using UML use cases enhanced with contracts (pre and post conditions). It relies on other tools to automate the approach. In this survey, however, we are considering the combination of tools that describe this approach. It works in two main steps, one generates test objectives from use cases enhanced with contracts, and the other generates the test scenarios from the test objectives and the use case scenarios.

The first main step receives as input use cases enhanced with executable contracts (pre and post conditions), which are written using a contract language that is proposed as part of the approach. The language is based on Boolean logic, so the use case contracts are first-order logical expression defined with respect to predicates. The predicates, in turn, describe facts in the system. These facts are described with respect to the use cases' parameters, which can be either actors or main concepts of the use case. These (contractualized) use cases are created with the support of an editor that manages the predicates and guides the design of contracts.

The contracts in the use cases are used to define a dependency order among use cases, which is an idea analogous to the graphical notations such as UML activity diagrams or dependency charts. To obtain this order, the use cases are instantiated with actual parameters, and then they are exhaustively simulated. The simulation generates a use case transition system (UCTS), which is a graph of nodes that describe the states of the system (a state is a set of predicates), and edges (transitions) between these states. The edges are labeled with actions that describe instantiated use cases. Therefore, a path in this graph is a valid sequence of instantiated use cases. In fact, the use case transition system (UCTS) is a representation of all possible ordering of the instantiated use cases. It is used to generate the test objectives (sequence of instantiated use cases), according to some selection criteria.

The second main step generates the test scenarios from the test objective and use case scenarios. Test scenarios are the sequential composition of use case scenarios (sequence diagrams). To transform the test objectives into a valid sequence of calls and expected outputs, each instantiated use case of the test objective is replaced by one use case scenario. When a scenario replaces an instantiated use case, the formal parameters of this scenario are also replaced by the actual parameters of the instantiated use cases. The last step in this process is the creation of java code from the test scenarios. This is done using an extension of the JUnit framework. The OCL contracts available with the sequence diagrams are transformed into JUnit assertions, while messages are transformed into method calls. One test method in the JUnit test class is considered a test case. Thus, in one test case from the JUnit framework is a sequence of calls to methods that correspond to an implementation of the sequence diagram. The approach relies on the JUnit infrastructure to run the test cases and evaluate the results.

5.11.1 Application of the Evaluation Framework

Figure 12 below shows that this approach supports test generation. The specifications input for the test generation are scenarios at the requirements level (use cases), and scenarios at the design level (sequence diagrams). The use cases are enhanced with contracts (pre and post conditions), represented by ‘other specifications’ in the figure. This activity outputs concrete test scripts, which are then executed. The results are evaluated, producing test results (as shown in figure 12). Table 11 answers the questions from the evaluation framework. Questions that are not applicable to this approach were answered with N/A.

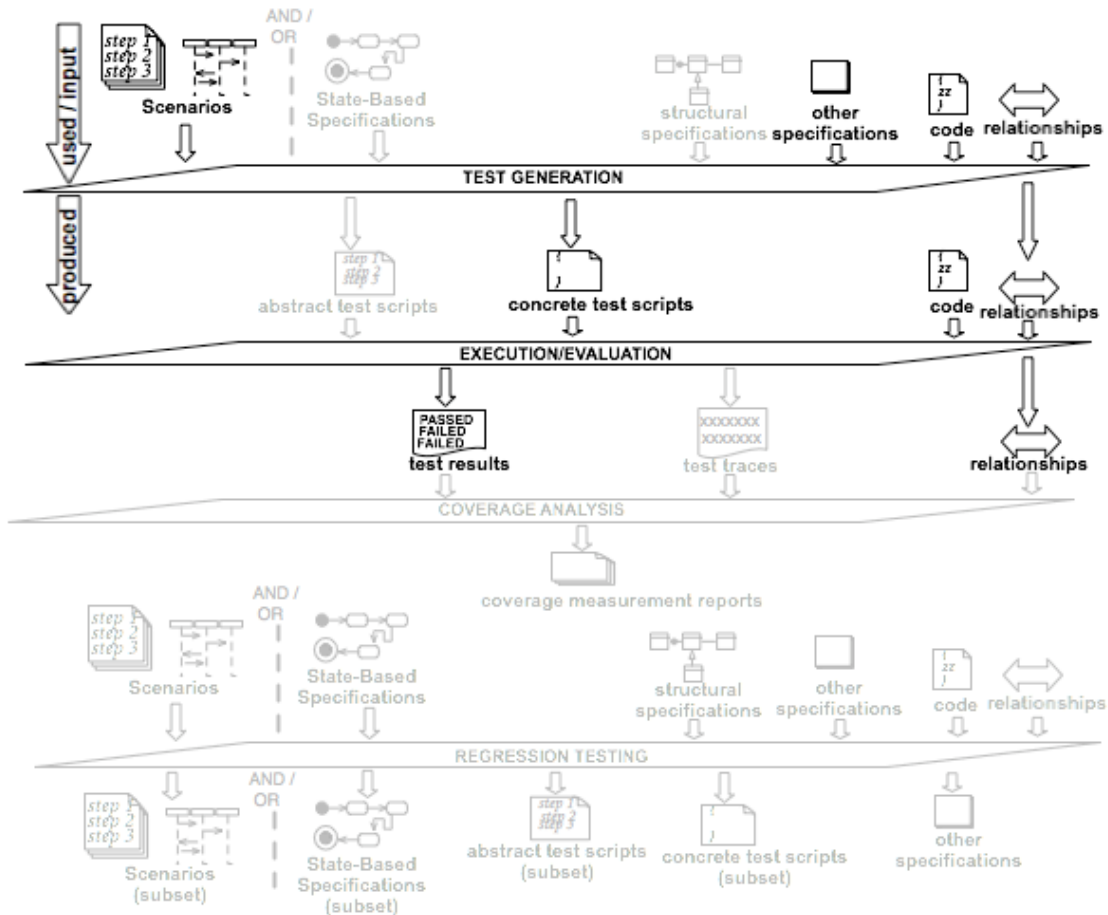


Figure 12 - Technical specification-based testing activities supported by UC-SCSystem.

Test Generation	Generates concrete test scripts?	<i>Yes, the approach generates concrete test scripts using an extension to the JUnit framework.</i>
	Coverage Criteria?	<i>Customizable to some extent, the coverage criteria available are defined with regards to the Use Case System Transition (UCTS): All Edges, All Vertices, All Instantiated Use Cases, All Vertices and All Instantiated Use Cases, All Precondition Terms, Robustness.</i>
	Kinds of relationships needed to support test generation?	Implicit, Horizontal, and Coarse-grained. They are obtained from UML model imported by the tool. They relate UML models at the same level of abstraction (use cases that depend on each other), Implicit, Vertical, and Coarse-grained. They relate each use case to the sequence diagram that realizes it. Implicit, Vertical and Fine-grained. They relate every method call in the sequence diagram to a method in the code. Also, OCL contracts in the sequence diagrams are related –mapped – to JUnit assertions.
	Support for management of these relationships?	<i>No.</i>
Test Execution and Evaluation	Specification used as oracle?	<i>N/A. Yes, the model is used to derive oracles from post-conditions of operations written with OCL</i>
	Automated evaluation?	<i>Yes, it is automatic, but the evaluations only check the contracts originally defined in the sequence diagram (expressed as JUnit assertions), the behavior is not considered.</i>
	Kinds of relationships needed to support execution and evaluation?	Implicit, Horizontal and Fine-grained. They are used to execute the test scripts. They are relationships from the concrete test scripts to the code it tests.
	Support for management of these relationships?	<i>No.</i>
Coverage Analysis	Support for coverage analysis?	<i>No.</i>
	Kinds of relationships needed to support coverage analysis?	<i>N/A.</i>
	Support for management of these relationships?	<i>N/A</i>
Regression Testing	Support for regression testing?	<i>No.</i>
General Aspects	Specification Language?	<i>Use Cases with a particular use case contract language, and UML sequence diagrams annotated with OCL contracts.</i>
	Level of formality?	<i>Medium.</i>
	Testing across levels of abstraction?	<i>No.</i>
	What is the level of abstraction of the specification input?	<i>Requirements-level scenarios (use cases) and Design-level scenarios (sequence diagrams).</i>
	What is the level of automation of the approach?	<i>Medium (one automated activity)</i>

Table 11 - Evaluation framework applied to UC-SCSystem

5.12 Summary

This section summarizes the main observations of this survey. Section 5.12.1 discusses observations with respect to test generation, section 5.12.2 discusses observations with respect to test execution and evaluation, and section 5.12.3 discusses observations with respect to coverage analysis and regression testing.

5.12.1 Test Generation

The following tables summarize important points regarding the test generation activity. The first and second rows describe the input and output required/produced. The third row describes the kinds of relationships used to support test generation, and the fourth row describes if the approach supports management such relationships.

Level of abstraction of the scenarios

It was expected that scenario-based testing approaches would address the need for using specifications that non-technical stakeholders find easier to manipulate. In particular, this need would be addressed by accepting scenarios at the requirements level. Only few approaches accept such scenarios (they are highlighted in gray in table 12).

The highlighted approaches, however, do not address this need:

- UCSC-System accepts use cases annotated with a formal language for describing pre and pos conditions contracts for the use cases.
- SOOTF accepts scenarios at the requirements level, and supports the user with a tool to transform this scenario into semi-formal test scenario specifications.
- AsmL accepts their definition of use cases, which requires the user to learn their language to describe the scenarios programmatically.

* Manually ** Outputs test requirements X Not Applicable			UCSC-System	SeDiTeC	SCENTOR	COW-SUITE	SOOTF	TOTEM	AGEDIS	UMLTest	UMLAUT	TESTOR	AsmL
INPUT	SCENARIOS	Requirements	✓				✓						✓
		Design	✓	✓	✓	✓	✓	✓		✓	✓		✓
	STATE MACHINES								✓		✓		✓
	STRUCTURAL DESC.			✓		✓		✓	✓		✓		
	OTHER SPECIFICATIONS		✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
	CODE		✓	✓	✓		✓						
OUTPUT	CONCRETE TEST SCRIPTS		✓	✓	✓		✓						
	ABSTRACT TEST SCRIPTS							✓	✓	**	✓	✓	✓
Kinds of relationships needed to support test generation?	IMPLICIT		✓✓✓	✓✓✓	✓			✓✓	✓✓		✓✓	✓✓	
	EXPLICIT					✓✓✓	✓✓	✓✓	✓✓				
	COARSE-GRAINED		✓✓		✓		✓	✓✓	✓		✓	✓	
	FINE-GRAINED			✓✓	✓	✓✓	✓	✓✓	✓		✓	✓	
	VERTICAL		✓✓✓	✓✓✓	✓	✓✓✓		✓✓	✓				
	HORIZONTAL		✓		✓	✓	✓✓	✓	✓		✓✓	✓✓	
Support for management of these relationships?							✓*			X			X

Table 12 – Test generation.

Kinds of relationships required by approaches that generate concrete test scripts

As highlighted in table 13, with exception of SOOTF, approaches that output concrete test scripts need fine-grained vertical relationships among the artifacts used to generate the test scripts. As already discussed, this kind of relationship is used to map concepts from high-level specifications to low-level ones (code). Since with SOOTF the user is charge of creating the test scenario specification, fine-grained vertical relationships are not used to support test generation.

* Manually ** Outputs test requirements X Not Applicable			UCSC-System	SeDiTeC	SCENTOR	COW-SUITE	SOOTF	TOTEM	AGEDIS	UMLTest	UMLAUT	TESTOR	AsmL	
INPUT	SCENARIOS	Requirements	✓				✓						✓	
		Design	✓	✓	✓	✓	✓	✓		✓	✓		✓	
	STATE MACHINES								✓		✓		✓	
	STRUCTURAL DESC.			✓		✓		✓	✓		✓			
	OTHER SPECIFICATIONS		✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	
	CODE		✓	✓	✓		✓							
OUTPUT	CONCRETE TEST SCRIPTS		✓	✓	✓		✓							
	ABSTRACT TEST SCRIPTS							✓	✓	**	✓	✓	✓	
Kinds of relationships needed to support test generation?	IMPLICIT		<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>				<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>		<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	
	EXPLICIT					<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>							
	COARSE-GRAINED		<div><div></div><div></div><div></div></div>			<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>		<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>		
	FINE-GRAINED		<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>		<div><div></div><div></div><div></div></div>		<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>		
	VERTICAL		<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>		<div><div></div><div></div><div></div></div>					
	HORIZONTAL		<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>		<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>		<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	
Support for management of these relationships?							✓*			X			X	

Table 13 - Test generation

Support for relationship management

As it could be expected, approaches that use implicit relationships do not support management of such relationships (highlighted in gray in table 14). Inline with this observation, SOOTF uses explicit relationships and provides manual support for relationship management.
































































* Manually ** Outputs test requirements X Not Applicable			UCSC-System	SeDiTeC	SCENTOR	COW-SUITE	SOOTF	TOTEM	AGEDIS	UMLTest	UMLAUT	TESTOR	AsmL	
INPUT	SCENARIOS	Requirements	✓				✓						✓	
		Design	✓	✓	✓	✓	✓	✓		✓	✓		✓	
	STATE MACHINES								✓		✓		✓	
	STRUCTURAL DESC.			✓		✓		✓	✓		✓			
	OTHER SPECIFICATIONS		✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	
	CODE		✓	✓	✓		✓							
OUTPUT	CONCRETE TEST SCRIPTS		✓	✓	✓		✓							
	ABSTRACT TEST SCRIPTS							✓	✓	**	✓	✓	✓	
Kinds of relationships needed to support test generation?	IMPLICIT		  	 					 	 		 	 	
	EXPLICIT					  	 							
	COARSE-GRAINED		 					 	 					
	FINE-GRAINED			  		  	 							
	VERTICAL		  			  								
	HORIZONTAL							 				 	 	
Support for management of these relationships?							✓*			X			X	

Table 14 – Test Generation

5.12.2 Test Execution and Evaluation

The following tables summarize important points regarding the test execution and evaluation. The first and second rows summarize input and output required/produced. The third row informs if the approach uses the specifications as oracles. The fourth row summarizes the support provided for automated test evaluation. The fifth row describes the kinds of relationships used to support test execution and evaluation, and the sixth row describes if the approach supports management of such relationships.

Support for automated evaluation and use of specifications as oracles

As highlighted in table 15, it can be observed that approaches whose test generation engine outputs concrete scripts (shown as input in this table) use specifications as oracles and support automated evaluation. It is worth noting, however, that from the highlighted approaches, only SeDiTeC actually compares the expected behavior to the obtained. The others compare only the results.

* Manually X Not Applicable		UCSC-System	SeDiTeC	SCENTOR	COW-SUITE	SOOTF	TOTEM	AGEDIS	UMLTest	UMLAUT	TESTOR	AsmL
INPUT	CODE	✓	✓	✓		✓		✓				✓
	CONCRETE TEST SCRIPTS	✓	✓	✓		✓						
	ABSTRACT TEST SCRIPTS							✓				✓
OUTPUT	TEST RESULTS	✓	✓	✓		✓		✓				✓
	TEST TRACES		✓					✓				
Oracles are specifications?		✓	✓	✓		✓	✓	✓			✓	✓
Automated evaluation?		✓	✓	✓		✓		✓				✓
Kinds of relationships needed to support execution and evaluation?	IMPLICIT	✓	✓	✓		✓						
	EXPLICIT					✓		✓				✓
	COARSE-GRAINED					✓						
	FINE-GRAINED	✓	✓	✓		✓		✓				✓
	VERTICAL		✓					✓				✓
	HORIZONTAL	✓	✓	✓		✓						
Support for management of these relationships?					X	✓*	X	✓*	X	X	X	✓*

Table 15 – Test execution and evaluation

Other two approaches compare the expected behavior to the obtained behavior (AGEDIS and AsmL). This is highlighted in table 16. Note that these approaches receive as input abstract instead of concrete test scripts, and thus require fine-grained and vertical relationships to execute and evaluate such test scripts.

* Manually X Not Applicable		UCSC-System	SeDiTeC	SCENTOR	COW-SUITE	SOOTF	TOTEM	AGEDIS	UMLTest	UMLAUT	TESTOR	AsmL
INPUT	CODE	✓	✓	✓		✓		✓				✓
	CONCRETE TEST SCRIPTS	✓	✓	✓		✓						
	ABSTRACT TEST SCRIPTS							✓				✓
OUTPUT	TEST RESULTS	✓	✓	✓		✓		✓				✓
	TEST TRACES		✓					✓				
Oracles are specifications?		✓	✓	✓		✓	✓	✓			✓	✓
Automated evaluation?		✓	✓	✓		✓		✓				✓
Kinds of relationships needed to support execution and evaluation?	IMPLICIT	✓	✓	✓		✓						
	EXPLICIT					✓		✓				✓
	COARSE-GRAINED					✓						
	FINE-GRAINED	✓	✓	✓		✓		✓				✓
	VERTICAL		✓					✓				✓
	HORIZONTAL	✓	✓	✓		✓						
Support for management of these relationships?					X	✓*	X	✓*	X	X	X	✓*

Table 16 - Test execution and evaluation

Support for relationship management

As expected, approaches that use explicit relationships provide support for management of such relationships. However, the support available with these approaches require user to create and manage the relationships. Thus, they are considered as manual support.

* Manually X Not Applicable		UCSC-System	SeDiTeC	SCENTOR	COW-SUITE	SOOTF	TOTEM	AGEDIS	UMLTest	UMLAUT	TESTOR	AsmL
INPUT	CODE	✓	✓	✓		✓		✓				✓
	CONCRETE TEST SCRIPTS	✓	✓	✓		✓						
	ABSTRACT TEST SCRIPTS							✓				✓
OUTPUT	TEST RESULTS	✓	✓	✓		✓		✓				✓
	TEST TRACES		✓					✓				
Oracles are specifications?		✓	✓	✓		✓	✓	✓			✓	✓
Automated evaluation?		✓	✓	✓		✓		✓				✓
Kinds of relationships needed to support execution and evaluation?	IMPLICIT	✓	✓	✓		✓						
	EXPLICIT					✓		✓				✓
	COARSE-GRAINED					✓						
	FINE-GRAINED	✓	✓	✓		✓		✓				✓
	VERTICAL		✓					✓				✓
	HORIZONTAL	✓	✓	✓		✓						
Support for management of these relationships?					X	✓*	X	✓*	X	X	X	✓*

Table 17 - Test execution and evaluation

5.12.3 Coverage Analysis and Regression Testing

The following tables summarize important points regarding coverage analysis and selective regression testing activities. The First and second rows summarize input and output required/produced. The third row informs if the approach supports coverage analysis. The fourth and fifth rows describe the kinds of relationships required to support such activity and if the approach supports management of such relationships. The sixth row informs if the approach supports regression testing.

It can be observed in table 18 that the majority of the approaches do not support such coverage analysis or regression testing. In fact, coverage analysis supported by the AGEDIS tool is based on code coverage (at the method level).

Support for relationship management

Table 18 highlights that both approaches that support coverage analysis, support automatic creation and persistence of the relationships required for such activity. This can be explained by the nature of the artifacts that stand at the end points of the relationships used to support coverage analysis. These artifacts are test results and test traces. The relationships are created when the artifacts are created by the previous activities (test execution and evaluation). If such relationships were modified, it would result on the modification of the artifacts used as basis for the coverage analysis. As a consequence, the coverage analysis would not consider data that resulted from actual execution of test scripts.

* Automatically creates and persists relationships.		UCSC-System	SeDiTeC	SCENTOR	COW-SUITE	SOOTF	TOTEM	AGEDIS	UMLTest	UMLAUT	TESTOR	AsmL
INPUT	TEST RESULTS					✓		✓				
	TEST TRACES							✓				
OUTPUT	COVERAGE REPORT					✓		✓				
Support for coverage analysis?						✓		✓				
Kinds of relationships needed to support coverage analysis?	IMPLICIT							✓				
	EXPLICIT					✓		✓				
	COARSE-GRAINED					✓		✓				
	FINE-GRAINED							✓				
	VERTICAL					✓		✓				
	HORIZONTAL							✓				
Support for management of these relationships?						✓*		✓*				
Support for regression testing						✓	✓					

Table 18 - Coverage analysis and regression testing

6 Other Related Work

6.1 *SCENT-Method*

SCENT-Method [64, 65] is a method for SCENario-Based Validation and Test of Software. It describes a systematic way to generate test sequences for system testing from scenarios at the requirement level. The method aims at using the scenarios not only for requirements elicitation, but also for validating the system. For that reason, scenarios are used throughout the process of software development. With this method, scenarios are developed and later formalized into state charts. The state charts are annotated with additional information relevant for test generation. This Information consists of preconditions, data input, data output, data ranges, and nonfunctional requirements. Then, state charts are traversed to generate test sequences. They are traversed using some existing algorithm for traversing finite state machines.

The method suggests that invalid sequences of event should be included when generating the test sequences. This is because path traversal will only describe valid sequences. The method also proposes the use of a diagram called dependency charts to enhance the test suite derived traversing the state charts. Dependency charts describe the dependencies among the scenarios. They are used to determine the sequences in which the scenarios should be executed. An algorithm that traverses the dependency chart should also be used to derive test cases. This time however, the traversal is specifying the order in which the scenarios should be executed. This activity is more complex due to the kinds of dependencies that might be described in the graph. If the chart describes a certain order of execution for the scenarios, the test sequences should also include sequences that do not respect that order.

6.2 *Engineered Use Cases*

As discussed in section 3.1, a use case is composed of a set of individual traces. Each trace is a scenario. Use cases have also been used to systematically create test scripts. However, approaches that automate this creation/generation are scarce. The reasons for the difficulty of automation can be attributed to the level of abstraction at which they are usually described, to their textual representations and to the variations of their semantics.

The need for providing a precise foundation for use case development was recognized and discussed in [74], where authors identify potential solutions that support an engineering approach to solve common problems for the full exploration of use cases as basis for test generation. This work precisely define use cases as a means to describe what the system does and not how. Engineered use cases can be defined based on a domain model, predefined basic actions and flow control actions (the semantics of these actions are also defined). The domain model is created using UML class diagrams. These elements, in addition to primitive types, are used to detail the use cases. The Engineered use case also has pre and post conditions described with respect to the domain model. As argued by the authors, Engineered use cases have the potential to be used for prototyping, estimation, refinement to design and test case generation. In fact, they have been used for test case generation with automatic tool support. This approach generates a test suite that optimizes coverage of the use case input parameters and reduces the size of the test suite. This tool was developed at IBM Research and is under evaluation.

6.3 *TAOS*

Testing with Analysis and Oracle Support (TAOS) provides support for the testing process. The level of testing supported is Unit testing. TAOS is integrated with two tools that allow it to support management of testing artifacts and testing activities. The first tool is Program

Dependence Analysis Graph (ProDAG) toolset. This integration allows TAOS to use program dependence analysis for testing, debugging, and maintenance. The second tool, PLEIADES, is a tool that supports flexible object management capabilities. This integration allows TAOS to support management of testing artifacts and persist them to a repository built atop PLEIADES. TAOS supports the following testing activities: *test management*, *program dependence analysis*, *test development*, *test execution* and *test measurement*.

Support for *test management* means support for creation, manipulation and access to test artifacts and relationships among these artifacts. The integration with the program analysis tool also allows TAOS to support relationships between test artifacts and analysis artifacts. Other relationships include relationships between test artifacts and the artifacts from which they were generated, or which they are to test. *Program dependence analysis* is a very significant activity for testing. It supports identification of semantic relationships between components. Therefore, it supports identification of components whose behavior was affected by a semantic change to another component. TAOS supports these kinds of analysis due to its integration to ProDAG. TAOS provides the user with a GUI that is capable of depicting control flow graphs and dependence graphs. That allows the user to browse the programs' dependencies. Thus, it improves users' understanding about the program. This also allows the users to find anomalies not easy to identify without such a graph (e.g., undefined use, unused definitions). Additionally, it supports filtering. Filtering allows the user to perform forward and backward slicing. *Test development* means support for creating test cases, test suites and test oracles. Test cases can be created manually. It can also be created using a random generator. TAOS also supports the creation and maintenance of test oracles using a particular formal specification language Graphical Interval Logic (GIL). It allows the user to specify some properties that are automatically checked after tests' execution. TAOS also supports *test execution* by following the preferences configured by the user (the tester). It retrieves the necessary information from a test artifact repository. It also verifies the execution by using the relationships between the artifact executed and the oracle associated with it. Additionally, TAOS supports *test measurement*, which is done by using the test criterion input by testers. Test criterion is considered the test requirement. Thus, TAOS uses the criterion to measure how much of the criterion was satisfied by the tests executed. The criterion should have been created beforehand and associated with the test suites.

This tool is relevant to the survey because it applies concepts of traceability to testing activities. These concepts are identified through the use of the aforementioned relationships. It is also identified by the use of a repository infrastructure to store and manage test artifacts and relationships. However, since it is not focused on specification-based testing, it does not use the relationships to generate tests from the specification, nor to perform other activities described in this survey. The relationships are used to manage the artifacts. The idea of using relationships among testing artifacts to support testing activities that inspired this survey is analogous to the idea of this tool. Nevertheless, compared to TAOS, the survey suggests that we should use tests artifacts at a higher level of abstraction. It also suggests that we should use the specifications to generate the tests, to measure coverage obtained and to perform regression testing.

6.4 Model-Driven Architecture and Testing

Model-Driven Architecture (MDA) is a standard defined by the Object Management Group (OMG). It describes approaches to software development that focuses on creating and transforming UML models. These models are transformed throughout different levels of abstraction until an implementation is produced. Tools that implement this concept (e.g open source tools like Eclipse EMF, StarUML) mostly concentrate on generating code from UML models. Approaches to MDA transformations can be considered to implement mapping relationships, which are expressed as transformation rules. These relationships are implicit

(represented as transformation rules), fine-grained and vertical. Therefore, MDA approaches usually do not support management of this kind of relationships among the manipulated models.

Some of the approaches studied in this survey implement the idea of MDA's transformations. For instance, AGEDIS provides a compiler for the AML model and the test generation directives. It converts the interpreted information into the intermediate format (IF) [20]. This is a transformation from model to model. In MDA terms, it is a Platform Independent Model to Platform Independent Model (PIM to PIM). Another transformation implemented by AGEDIS is from abstract to executable test cases, this is a transformation from model to code. In MDA terms, it is a Platform Independent Model to Platform Specific Model (PIM to PSM). This transformation relies on additional information (test execution directives), which are explicit, fine-grained and vertical relationships.

As part of the MDA standard, OMG also defined a UML Testing profile (UML2TP), which is a metamodel that describes the information required by black-box testing approaches to evaluate the correctness of a system's implementation. Therefore, test cases can be considered as another kind of model. Given the existence of such model for describing test artifacts, the tools that comply with the MDA standards would support transformations to that model. This profile is relatively new, so currently available MDA compliant tools that adopt it (e.g. Hyades Eclipse project, and Telelogic TAU G 2) are scarce.

7 Conclusions and Research Recommendations

Although software testing is still regarded by practitioners as the central activity used for ensuring that a system behaves as expected, traditional software development processes leave their activities to the end of the software life cycle, so schedule slippage, time-to-market pressures, and cost-constraints result in neglected testing. Automated testing approaches can ameliorate this problem because automation reduces the effort spent on testing activities. Current code-based testing automated approaches generate, run and analyze statement and/or method coverage of test scripts. Therefore, they reduce test and development efforts. However, code-based testing alone is not enough to ensure that expectations about the system are fulfilled by its implementation, which can be achieved with automated specification-based testing.

This survey studies scenario-based and state-based automated testing approaches. It shows that such approaches still demands improvements on support for some testing activities. The following points are research recommendations that could contribute to these improvements.

Increase the level of abstraction of specifications used as basis for deriving testing

As observed, the majority of scenario-based approaches accept scenarios at the design or lower level. This means there is still a gap between such scenarios and those that non-technical stakeholders' would find easier to manipulate. Therefore, it is important to investigate the feasibility of using scenarios at the requirements level as input to specification-based testing approaches for the following reasons: (1) they are actually closer to non-technical stakeholders' comprehension, (2) this would leverage the effort put into creating this kinds of specifications, (3) the semantic gap between requirements in this format and scenarios at other levels of abstraction is smaller than the gap between requirements expressed in prose and scenarios at other levels of abstraction. This could attenuate the effort spent on establishing the relationships between them, which are required for testing activities. The interesting challenge is the tradeoff between the informality of scenarios at the requirement level and the need of formality in scenarios used as input for current testing approaches.

Improve support for (scenario-based/state-based) selective regression testing

Software is usually subject to constant changes, as are its specifications. When changes happen, selective regression testing plays an important role at reducing the chances that new faults were inserted to the system. Selective regression testing has been implemented in relation to code, specification and architecture. Usually, the main activity is the comparison of different versions of the code, specification or architecture to identify entities that changed. The next activity is identification of possibly impacted test scripts. This identification is done with the relationships between the artifact used to generate the test scripts and the test scripts themselves. As observed, the majority of studied approaches do not support selective regression testing. Given the importance of such activity, automated testing approaches should improve their support for selective regression testing.

A factor that influences the precision of selective regression testing is the level of granularity of the end points of the relationships used to support identification of impacted test cases (or impacted specifications). The more fine-grained these endpoints are, the more precise the selection could be (e.g. code-based regression testing). Therefore, one challenge related to (scenario-based/state-based) selective regression testing is identifying the ideal level of granularity of the artifacts related that would achieve gains with selective regression testing.

Improve support for (scenario-based/state-based) coverage analysis

Inline with previous recommendation, it was also observed that only two approaches, AGEDIS and SOOTF, support the user with further information about effectiveness of the tests run beyond displaying results (test pass or fail). The majority of the approaches are still lacking the support for coverage analysis based on high-level artifacts (or based on entities that compose these artifacts). Users should receive information such as percentage of the requirements, or of scenarios covered by the tests run. This kind of information could be inferred from relationships between the specification used to generate the tests scripts, the test scripts and their results. Given the importance of such activity, automated testing approaches should improve their support for coverage analysis based on other (higher level) artifacts than code-level.

Improve for relationship management in automated testing approaches

As expected, studied approaches rely on some mechanism to establish relationships between entities that compose the software artifacts. Relationships can connect entities across models, or entities in models and entities in the implementation. They are used to support test scripts generation, or to automate the test execution and evaluation.

The support provided for establishing and creating these relationships varies with the approaches. Some approaches have manual support for managing those relationships and require the users to explicitly establish them. When possible, other approaches infer relationships considering name matching and the semantics of the models. For instance, AsmL uses the information input by the user about the namespace to match classes and methods at design and implementation levels. Additionally, approaches that use UML models rely on implicit relationships available from the models. Unfortunately, even if established, these relationships are not maintained.

Some traceability tools recover relationships between artifacts with different techniques such as dynamic and static reverse engineering [24, 7, 67]. Therefore, these recovered relationships could be used to support testing. The precision of such tools, however, remains a research subject in the traceability community. Additionally, it is important to note that there is a restriction to the level of abstraction of the artifacts that these tools support. Approaches that automate the relationship recovery are usually able to recover relationships between artifacts at the code level and artifacts at the design level [25]. Manual intervention is used to close the gap between artifacts at the design level and artifacts at the requirements level. Other tools assume existence (or manual creation) of such relationships and concentrate on managing changes to fine-grained artifacts and to the relationships among them [51, 50].

This survey showed specification-based testing approaches are currently lacking in support for automated management of the artifacts used and produced, and the relationships among them. Therefore, in spite the possible reduced precision and the need for manual intervention, integration of traceability tools with specification-based testing approaches should improve the level of automation of these approaches. This integration has the potential of leveraging relationship-related testing activities (e.g. regression testing, and coverage analysis). It also has the potential of reducing the burden of relationship maintenance for the users of specification-based testing approaches. Last but not least, it leverages the effort spent on establishing relationships, and thus could encourage the use of traceability tools.

8 Glossary

Abstract State Machines (ASM) [3] idea was originally developed by Yuri Gurevich, are a formal method for specifying and verifying systems, based on abstract state machine (most often used as synonym for finite state machine). The theory states that the behavior of an algorithm can be step-for-step simulated (at its own natural abstraction level) by an appropriate ASM. Thus, ASM support specification of algorithms in many levels of abstraction, and they come in three flavors: sequential, parallel and distributed.

Code-Base Testing is a way to test the software based on its code, knowledge of the source code is required. The test cases are generated with the objective of exercising parts of the code such as statements, methods, branches, paths, etc.

Conformance testing is similar to runtime verification, which is comparing the implementation to its specification dynamically (runs the implementation and compares it to the specification).

Executable test script consists of calls to system methods with what inputs to provide, and what outputs to expect, the evaluation is either made manually by the tester, or automatically by an oracle.

Explicit relationship is a relationship between two different artifacts (regardless their level of granularity), which the user of the tool is aware of. The relationship can either be made explicit automatically by the tool, or the tool might require the user to explicitly create it.

Finite State Machine (FSM) [3] is a model behavior that consists of states, transitions and actions. States indicate the state of the system, the transitions indicate a change of state and the actions are activities to be performed in a certain moment (e.g. when entering or exiting a state). The connection of Abstract State Machines (ASM) to Finite State Machines (FSM) is as follows. As described in 21, the ASM is the abstraction of an algorithm, and may have infinitely many states that can be grouped into finitely many hyperstates, giving rise to a finite state machine (what is done by the AsmL test tool approach with their FSM generation feature).

Implicit relationship is a relationship between two different artifacts (regardless their level of granularity), which the user of the tool is not aware of the existence. The user is aware of the existence of the related artifacts.

Integration level test script is supposed to test that the units implemented and tested individually work together as expected, in this case, testing can take place not only to compare the specification to the code, but also to exercise the specifications in different levels of abstraction comparing a lower level to a higher one.

JUnit is a framework that supports development of java unit testing. The framework defines how the test cases should be structured. It provides tools for running the unit tests and for evaluating the results obtained, which is done through assertions.

Manual test script describes what steps the tester should follow when using the system, what inputs to provide and what outputs to expect, the evaluation of the result is also made manually.

Oracle is “some method for checking whether the system under test behaved correctly on a particular execution” [10].

Precondition is a condition that needs to be met by the input value.

Postcondition is a condition that needs to be met by the output value.

Scenario “A scenario is a sequence of events. What constitutes an event is variously described in the literature: something that happens, an action, an interaction, a step, or in some usages only an instantaneous change from one state to another” [48].

Specification-Based Testing is a way to test the software based on its specification. The specification describes how the software is expected to behave and is used as a reference against which the implementation should be compared. The test scripts used to test the software are derived based on the specification.

Specification and Description Language (SDL) is a formal specification language defined by the ITU-T (Z.100). It aims at specifying and describing unambiguously the behavior of reactive and distributed system.

System level test script evaluates the system as a whole against the requirements; it can also be called the Black-box test, since internal knowledge of the system is not necessary for that kind of test.

Test Case “has an identity and is associated with a program behavior, It also has a set of inputs and a list of expected outputs” [42]. Different users and approaches use different format of test cases, some of them have other information about it such as its status, its actual results, an unique identifier and so forth, but in its core, all of them have input and expected outputs. When used in the context of this survey, a test case is an abstract test script. Therefore, it consists of test values input and output, and a step of steps for manual execution.

Test Input is the same as test values, or values provided as input so as the test can executed.

Test Scripts are a set of steps that should be followed for testing a program.

Test Purpose is the information about the specification in which the test generation should be focused.

Unit level test script verifies if a unit of code is running as expected, this unit can be a module, a class, a procedure, and so forth.

Acknowledgments

This effort was also sponsored in part by the National Science Foundation under grant CCR-0205724.

References

- [1] *AsmL - Abstract State Machine Language.*
- [2] *FoCuS - Functional Coverage Tool.*
- [3] *Wikipedia - The Free Encyclopedia.*
- [4] *AsmL Test Tool Users' Guide*, 2004.
- [5] I. Alexander, *Towards automatic traceability in the industrial practice, 1st international workshop on traceability in emerging forms of software engineering*, Edinburgh, UK, 2002.
- [6] I. F. Alexander, Maiden, N., *Scenarios, Stories, Use Cases Through the Systems Development Life-Cycle*, John Wiley & Sons, 2004.
- [7] G. C. Antoniol, G.; Casazza, G.; De Lucia, A.; Merlo E., *Recovering Traceability Links between Code and Documentation*, *IEEE Transactions on Software Engineering*, 2002, pp. 970-983.
- [8] A. I. Antón, Potts, C., *A Representational Framework for Scenarios of System Use*, *Requirements Engineering Journal*, 3 (1998), pp. 219-241.
- [9] R. Balasubramaniam, Matthias, J., *Toward Reference Models for Requirements Traceability*, IEEE Press, 2001.
- [10] L. Baresi, Young, M., *Test oracles*, Dept. of Computer and Information Science, University of Oregon, 2001.
- [11] M. Barnett, Grieskamp, W., Gurevich, Y., Schulte, W., Tillmann, N., Veanes, M., *Scenario-Oriented Modeling in AsmL and its Instrumentation for Testing, 2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM 2003)*, Portland, Oregon, 2003.
- [12] A. P. I. L. Basanieri, G.; Marchetti, E., *An Industrial Experience in Comparing Manual vs. Automatic Test Cases Generation*, *ACIS Fourth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, Lübeck, Germany, 2003.
- [13] F. B. Basanieri, A.; Marchetti, E.; Ribolini, E. A.; Lombardi, G.; Nucera, G., *An Automated Test Strategy Based on UML Diagrams*, *Proc. Ericsson Rational User Conference*, Upplands Vasby Sweden, 2001.
- [14] F. B. Basanieri, A.; Marchetti, E., *The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects*, *Proceedings of the 5th International Conference on The Unified Modeling Language*, Springer-Verlag, 2002, pp. 383-397.
- [15] A. B. Bertolino, F., *A Practical Approach to UML-Based Derivation of Integration Tests*, *4th International Software Quality Week Europe and International Internet Quality Week Europe*, Brussels, Belgium, 2000.
- [16] L. C. Briand, Labiche, Y., *A UML-Based Approach to System Testing*, *4th International Conference on the Unified Modeling Language (UML)*, Toronto, Canada, 2001, pp. 194-208.
- [17] L. C. Briand, Labiche, Y. and Soccar, G., *Automating Impact Analysis and Regression Test Selection Base on UML Designs*, Carleton University, 2002.
- [18] L. C. Briand, Labiche, Y. and Soccar, G., *Automating Impact Analysis and Regression Test Selection Base on UML Designs*, *International Conference on Software Maintenance (ICSM)*, 2002, pp. 252-261.
- [19] L. C. Briand, Labiche, Y., Soccar, G., *Automating impact analysis and regression test selection based on UML designs*, *International Conference on Software Maintenance*, 2002, pp. 252 - 261.
- [20] A. Cavarra, Chrichton, C., Davies, J., Hartman, A., Jeron, T., Mounier, L., *Using UML for Automatic Test Generation, Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'2000)*, Springer Verlag, 2000.
- [21] J. Cleland-Huang, Chang, C., Wise, J., *Supporting Event Based Traceability through High-Level recognition change of events*, *IEEE COMPSAC Conferenc*, Oxford, England, 2002.

- [22] C. Csallner, Smaragdakis, Y., *JCrasher: An automatic robustness tester for Java*, 2004, pp. 1025-1050.
- [23] S. C. Easterbrook, J.; Wiels V., *V & V through Inconsistency Tracking and Analysis*, 9th International Workshop on Software Specification and Design, 1998, pp. 43-51.
- [24] A. Egyed, *A scenario-driven approach to traceability*, IEEE Computer Society, Toronto, Ontario, Canada, 2001.
- [25] A. Egyed, *A scenario-driven approach to trace dependency analysis*, IEEE Transactions on Software Engineering, 2003.
- [26] E. Farchi, Hartman, A., Pinter, S. S., *Using a model-based test generator to test for standard conformance*, IBM Systems Journal, 41 (2001).
- [27] R. A. Fiutem, G., *Identifying Design-Code Inconsistencies in Object-Oriented Software: a Case Study*, International Conference on Software Maintenance, 1998, pp. 94-103.
- [28] F. Fraikin, Leonhardt, T., <http://www.pi.informatik.tu-darmstadt.de/forschung/seditec/>.
- [29] F. Fraikin, Leonhardt, T., *SeDiTeC — Testing Based on Sequence Diagrams*, 17th IEEE International Conference on Automated Software Engineering, 2002, pp. 261 - 266.
- [30] E. Gamma, Beck, K., *JUnit - framework to write repeatable tests*.
- [31] T. L. Graves, Harrold, M. J., Kim, J.-M., Porter, A., Rothermel, G., *An Empirical Study of Regression Test Selection Techniques*, ACM Transactions on Software Engineering and Methodology, 2001, pp. 184-208.
- [32] W. Grieskamp, Gurevich, Y., Schulte, W., Veanes, M., *Generating finite state machines from abstract state machines*, ACM Press, Roma, Italy, 2002.
- [33] W. Grieskamp, Nachmanson, L., Tillmann, N., Veanes, M., *Test Case Generation from AsmL Specifications - Tool Overview*, 10th International Workshop on Abstract State Machines, Taormina, Italy, 2003.
- [34] W. Grieskamp, Schulte, W., Lepper, M., Tillman, N., *Testable Use Cases in the Abstract State Machine Language*, Proceedings of Asia-Pacific Conference on Quality Software (APAQSO'01), 2001.
- [35] Y. Gurevich, *Abstract State Machines: An Overview of the Project*, Symposium on Foundations of Information and Knowledge System, Vienna, Austria, 2003.
- [36] A. N. Hartman, K., *AGEDIS Final Project Report*, 2004.
- [37] A. N. Hartman, K., *The AGEDIS tools for model based testing*, 2004 ACM SIGSOFT international symposium on Software testing and analysis, ACM Press, Boston, Massachusetts, USA, 2004, pp. 129-132.
- [38] J. H. Hayes, Dekhtyar, A., Osborne, J., *Improving Requirements Tracing via Information Retrieval*, IEEE Computer Society, 2003.
- [39] M. Jarke, Tung Bui, X., Carroll, J. M., *Scenario management: An interdisciplinary approach*, Requirements Engineering Journal, 3 (1998), pp. 155-173.
- [40] W. Jeremiah, Frank, M., *Using UML to Partially Automate Generation of Scenario-Based Test Drivers*, 7th International Conference on Object Oriented Information Systems, 2001, pp. 303-306.
- [41] T. Jeron, Morel, P., *Test generation derived from model-checking*, CAV'99, Springer-Verlag, Trento, Italy, 1999, pp. 108-122.
- [42] P. C. Jorgensen, *Software Testing - A Craftsman Approach*, 2002.
- [43] A. v. Knethen, *Automatic Change Support based on a Trace Model*, International Workshop on traceability in emerging forms of software engineering (TEFSE 2002), Edinburgh, UK, 2002.
- [44] Y. d. B. Ledru, L.; Bontron, P.; Maury, O.; Oriat, C.; Potet, M.-L., *Test Purposes: Adapting the Notion of Specification to Testing*, International Conference on Automated Software Engineering (ASE), 2001, pp. 127.
- [45] M. Lindval, Sandahl, K., *Practical Implications of Traceability*, Software Practice and Experience, 1996, pp. 1161-1180.
- [46] A. Marcus, Maletic, J. I., *Recovering documentation-to-source-code traceability links using latent semantic indexing*, IEEE Computer Society, Portland, Oregon, 2003.
- [47] H. Muccini, Dias, M. S., D. J. Richardson, *Towards software architecture-based regression testing*, Journal of Systems and Software, Special Issue on "Architecting Dependable Systems" (2006), pp. 1-7.

- [48] L. Naslavsky, Alspaugh A. T., Richardson J. D., Ziv H., *Using Scenarios to Support Traceability*, *International Workshop on Traceability in Emerging Forms of Software Engineering*, ACM Press, Long Beach, 2005, pp. 25-30.
- [49] C. Nebut, Fleurey, F., Traon, Y. L., Jézéquel, J., *Automatic Test Generation: A Use Case Driven Approach*, *IEEE Transactions on Software Engineering*, 32 (2006), pp. 140-155.
- [50] T. N. M. Nguyen, E. V.; Boyland, J. T., *The molhado hypertext versioning system*, ACM Press, Santa Cruz, CA, USA, 2004.
- [51] T. N. M. Nguyen, E. V.; Boyland, J. T.; Thao, C., *An infrastructure for development of object-oriented, multi-level configuration management services*, ACM Press, St. Louis, MO, USA, 2005.
- [52] J. A. Offutt, A., *Generating Tests from UML specifications*, *Second International Conference on the Unified Modeling Language*, Fort Collins, CO, 1999, pp. 416-429.
- [53] J. A. Offutt, A., *Using UML Collaboration diagrams for static checking and test generation*, *Third International Conference on UML*, York, UK, 2000.
- [54] J. A. Offutt, A.; Baldini, A., *A Controlled experiment evaluation of Test cases generated for UML diagram*, 2004.
- [55] J. L. Offutt, S.; Abdurazik, A.; Baldini, A.; Ammann P., *Generating Test data from State based Specifications*, *The Journal of Software Testing, Verification and Reliability*, 13 (2003), pp. 25-53.
- [56] T. Olsen, Grundy, J.C., *Supporting traceability and inconsistency management between software artefacts*, *IASTED International Conference on Software Engineering and Applications*, Boston, MA, 2002.
- [57] T. J. B. Ostrand, M. J., *The Category-partition method for specifying and generating functional tests*, *Communications of ACM*, 1988, pp. 676-686.
- [58] Parasoft, <http://www.parasoft.com/jsp/home.jsp>.
- [59] P. Pelliccione, Muccini, H., Bucchiarone, A., Facchini, F., *TeStor: Deriving Test Sequences from Model-based Specifications*, *Eighth International SIGSOFT Symposium on Component-based Software Engineering (CBSE 2005)*, *Lecture Notes in Computer Science*, LNCS 3489, St. Louis, Missouri (USA), 2005, pp. 267-282.
- [60] S. J. Pickin, C.; Le Traon, Y.; Jéron, T.; Jézéquel, J.-M.; Le Guennec, A., *System Test Synthesis from UML Models of Distributed Software*, in D. A. V. Peled, M.Y., ed., *Formal Techniques for Networked and Distributed Systems - FORTE 2002*, Springer Berlin / Heidelberg, Houston, Texas, USA, 2002, pp. 97 - 113.
- [61] R. M. Poston, *Automating Specification-Based Software Testing*, IEEE Computer Society Press, 1996.
- [62] D. J. Richardson, *TAOS: Testing with Analysis and Oracle Support*, ACM Press, Seattle, Washington, United States, 1994.
- [63] D. J. Richardson, O'Malley, O., Tittle, C., *Approaches to Specification-Based Testing*, *ACM Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, 1993, pp. 86-96.
- [64] J. Ryser, Glinz, M., *A Scenario-Based Approach to Validating and Testing Software Systems Using Statecharts*, *International Conference on Software and Systems Engineering and their Applications ICSSEA'99*, Paris, France, 1999.
- [65] J. Ryser, Glinz, M., *Using Dependency Charts to Improve Scenario-Based Testing*, *International Conference on Testing Computer Software TCS'2000*, Washington, D.C., 2000.
- [66] S. A. Sherba, Anderson, K. M., *A Framework for Managing Traceability Relationships between Requirements and Architecture*, *Second International Software Requirements to Architectures Workshop (STRAW'03)*, *Part of the 2003 International Conference on Software Engineering*, Portland, Oregon, 2003, pp. 150-156.
- [67] S. A. Sherba, Anderson, K. M., Faisal, M., *A Framework for Mapping Traceability Relationships*, *International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'03)*, Montreal, CA, 2003.
- [68] G. Spanoudakis, Kim, H., *Supporting the reconciliation of models of object behaviour*, *Journal of Software and Systems Modelling*, 3 (2004), pp. 273-293.
- [69] G. Spanoudakis, Zisman, A., *Software Traceability: A Roadmap*, *Advances in Software Engineering and Knowledge Engineering*, World Scientific Publishing, 2005.
- [70] W. T. B. Tsai, X.; Paul, R.; Shao, W.; Agarwal, V., *End-To-End Integration Testing Design*, *25th Annual International Computer Software and Applications Conference (COMPSAC'01)*, 2001, pp. 166 - 171.

- [71] W. T. S. Tsai, A.; YU, L.; R. Paul, *Scenario-based Object-Oriented Testing Framework*, *Third International Conference On Quality Software*, 2003, pp. 410 - 417.
- [72] S. Uchitel, Chatley, R., Kramer, J., Magee, J, *System Architecture: the Context for Scenario-based Model Synthesis*, *12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, 2004, pp. 33-42.
- [73] K. Weidenhaupt, Pohl, K., Jarke, M., Haumer, P., *Scenarios in System Development: Current Practice*, *IEEE Software*, 1998.
- [74] C. Williams, Kaplan, M., Klinger, T., Paradkar, A., *Toward Engineered, Useful Use Cases*, *Journal of Object Technology*, 4 (2005), pp. 45-57.
- [75] J. Wittevrongel, Maurer F., *Using UML to Partially Automate Generation of Scenario-Based Test Drivers*, *Object-Oriented Information Systems*, Springer, 2001.
- [76] J. Wittevrongel, Maurer, F., *SCENTOR: Scenario-Based Testing of E-Business Applications*, *Tenth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 2001, pp. 41 - 46.
- [77] T. Xie, Marinov, D., Notkin, D., *Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests*, *19th IEEE International Conference on Automated Software Engineering (ASE'04)*, 2004, pp. 196-205.
- [78] L. Xu, Dias, M., Richardson, D. J., *Generating Regression Tests via Model Checking*, *28th Annual International Computer Software and Applications Conference*, 2004.