TESTING POLICY-BASED SYSTEMS WITH SCENARIOS

Mark Strembeck
Institute of Information Systems, New Media Lab
Vienna University of Economics and Business (WU Vienna)
Austria
mark.strembeck@wu.ac.at

ABSTRACT
Policy-based systems consist of interacting software artifacts and, at first glance, can be tested as any other software system. In a policy-based system, however, the behavior of system entities may change dynamically and frequently, depending on the policy rules governing this behavior. Therefore, policy-based systems demand for a testing approach that especially allows for the testing of dynamically changing system behavior. Thus, testing of policy rules has to check if the behavior that is actually enforced by a set of policies, conforms to the intended behavior of the corresponding system entities. Scenarios are an important means to specify behavior of software entities. In this paper, we introduce an approach to test policy-based systems with scenarios, and present an (embedded) domain-specific language for scenario-based testing.

KEY WORDS
Policy-based systems, scenario-based testing

1 Introduction

In the information systems context, *Policies* are rules governing the choices in behavior of a (software) system [22]. They enable the dynamic parametrization of behavior in a system – without changing the system entities interpreting and enforcing these rules. The complexity of today's distributed information systems results in high administrative efforts which become even worse for growing/expanding systems. Policy-based systems are autonomic (self-managing) in a way that they autonomously determine which policies must be applied to a certain management situation. Therefore, policies have become a popular approach to cope with the increasing complexity of system management (see, e.g., [2, 29, 30, 32]).

Each *policy language* is in essence a domain-specific language (DSL) (see, e.g., [23, 26, 33]) that enables the definition of policy rules. Policy languages thus combine the advantages of DSLs and policy-based system management. They are a means for the parametrization and behavioral configuration of software-based systems. In other words, they allow to dynamically change the behavior of software components, without changing the implementation of these components. In principle, policy-based management can be added to each software-based system. However, most benefits are achieved if the corresponding software system is designed to be a policy-based system from the beginning, of course. In this case, each software component in the system is explicitly build for policy-based management. System engineers then define tailored policy languages for the different target domains (e.g. for system backups, access control, or quality-of-service policies) which are used to (dynamically) govern the behavior of the components in the system. Moreover, if a system is designed for policy-based management, it is possible to successively define additional policy languages for other target domains.

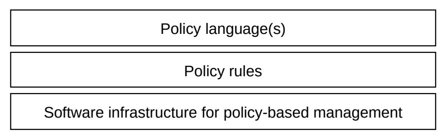

Figure 1. Test layers of policy-based systems

Figure 1 shows the test layers of policy-based systems. The bottom layer, includes a number of interacting software components that provide the infrastructure for policy-based management. The next layer includes policy rules that govern the behavior of system entities. Finally, the topmost layer in Figure 1 represents one or more policy language(s). To thoroughly test a policy-based system, we have to test each of these layers as well as inter-layer dependencies. Being a DSL, each policy language is based on a language (meta) model, including static and dynamic semantics[1]. Thus, testing policy languages especially has to verify that all policy expressions that are defined via a policy language conform to the static and dynamic semantics of this policy language.

One of the main characteristics of policy-based systems is that the runtime behavior of system entities can be changed dynamically, without changing or re-starting the system. This is a significant difference to other software systems. Thus, a testing mechanism for policy-based systems must enable the testing of runtime behavior induced by dynamically changing policies.

[1] Similar to the fact that each (non-trivial) software system has an architecture, no matter if the architecture is "good" or "bad", each language is based on a corresponding language (meta) model that defines the language's elements/alphabet and grammar, at least implicitly (see, e.g., [23, 26, 33])

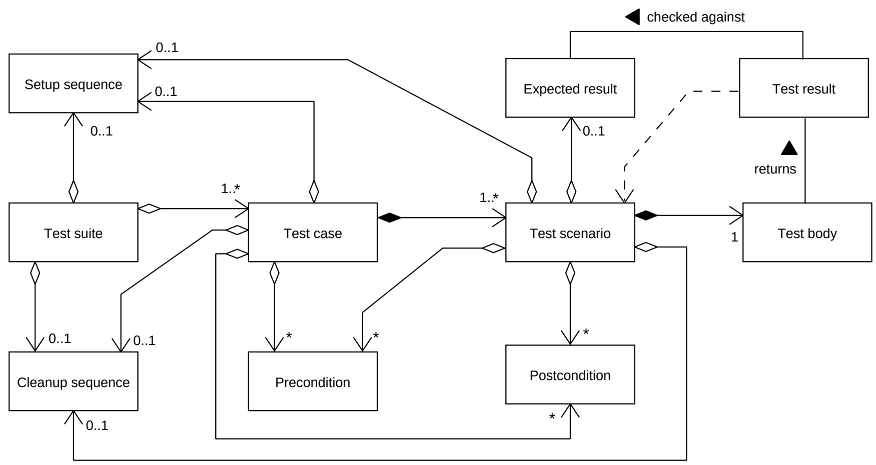

Figure 2. Concept formation: scenario-based testing

Because policies and scenarios are complementary artifacts (see [24]), scenario-based testing (see, e.g., [16, 20, 25]) is well-suited to meet the testing demands of dynamic policy-based systems. Scenarios describe action and event sequences and make process descriptions explicit. In software engineering, scenarios are used to explore and to define (actual or intended) system behavior as well as to specify user needs (see, e.g., [1, 8]). Scenarios can be described in many different ways and exist on different levels of abstraction (cf. [8]). When specifying a software system they are defined with different types of models, such as UML interaction models or UML activity models.

2 Scenario-based Testing

As it is almost impossible to completely test a complex software system, one needs effective means to select relevant test scenarios, express and maintain them, and automate tests whenever possible. Scenarios are a means to reduce the risk of omitting or forgetting relevant test cases, as well as the risk of insufficiently describing important tests (see, e.g., [16, 20, 25]). We especially follow the assumption that if each design-level scenario is checked via a corresponding test scenario, we reach a sound test coverage of the most relevant system functions. Moreover, in a thorough engineering approach, changes in behavior of a system are first identified at the scenario level (see also [8]). Thus, if a scenario changes, we can rapidly identify affected test scenarios and to propagate the changes into the corresponding test specifications.

A *test scenario* tests one particular behavioral facet of a system. In the first place, it represents one particular action and event sequence which is specified through the test body of the respective scenario. In addition to the test body, each test scenario includes an expected result and may include a number of preconditions and postconditions, as well as a setup sequence and a cleanup sequence (see Figure 2). When a test scenario is triggered, it first executes the corresponding setup sequence. A setup sequence includes an action sequence that is executed to set up a runtime environment for the corresponding scenario, for example a setup sequence may create several objects that are required by the commands invoked through test body.

Next, the preconditions of the scenario are checked. If at least one precondition fails, the test scenario is aborted and marked as incomplete. In case all preconditions are fulfilled, the test body is executed. In particular, the action sequence in the test body produces a test result. This test result is then checked against the expected result (for example using binary infix operators as $=$, $\geq$, or $\leq$). If the check fails, the test scenario is aborted and marked as incomplete. In case the check is successful, the postconditions of the scenario are checked. Again, if at least one postcondition fails, the test scenario is aborted and marked as incomplete. If all postconditions are fulfilled, the cleanup sequence is called and the scenario is marked as complete. A cleanup sequence includes an action sequence that is executed to undo changes that were made during the test scenario. For example, the cleanup sequence can delete objects that were created by the setup sequence.

Each test scenario is part of a test case. In particular, a *test case* consists of one or more test scenarios and may include a number of test case preconditions and test case postconditions, as well as corresponding setup and cleanup sequences (see Figure 2). When a test case is triggered, it first executes the respective setup sequence. The runtime structures produced by the setup sequence are then available to all test scenarios of the corresponding test case. Subsequently, the preconditions of the test case are checked. Similar to test scenarios, a test case is aborted and marked as incomplete if one of its preconditions or postconditions fails. Next, each test scenario of the test case is executed as described above. If at least one test scenario is incomplete, the corresponding test case is also marked as incomplete. After the test scenarios, the test case's postconditions are checked before the test case cleanup sequence is executed. Similar to the scenario cleanup sequence, the test case cleanup sequence is executed each time the corresponding test case is triggered. Each test case is part of a *test suite* (see Figure 2) and a test suite includes one or more test cases. Furthermore, a test suite may have an own setup sequence and a cleanup sequence. The runtime structures produced by the test suite setup sequence are available to all test cases of the corresponding suite.

3 Testing Policy-based Systems

In the context of policy-based system management, a *managed object* (or *managed entity*) is an object whose behavior is controlled via policy rules. Thus, policies refer to managed objects and to test policy rules we define intended system behavior via action and event sequences that have well-defined observable effects on managed objects – such as the invocation of a certain method, the controlled manipulation of a certain variable, or sending and receiving of a particular message over a network, for example. Subsequently, these sequences are executed and it is checked if the actual (policy governed) behavior of the respective system entities yields the exact effects specified through the test sequences. For the purposes of this paper, we subdivide testing of policy-based systems in policy runtime testing (Section 3.1), testing of policy language semantics (Section 3.2), and user acceptance testing of policy languages (Section 3.3).

3.1 Policy Runtime Testing

A test suite for policy runtime testing sets up a well-defined runtime environment and specifies test scenarios that define typical usage patterns of the system. Then the test scenarios are executed to check if the policy controlled behavior of the corresponding system entities conforms to the expected/intended behavior. Figure 3 depicts the process of policy runtime testing.

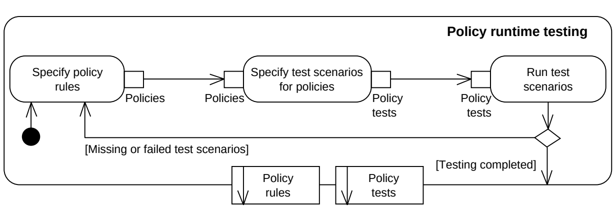

Figure 3. Policy runtime testing

Example test scenarios for policy runtime testing are:

- Context: hospital information system. A primary care physician (respectively a software agent representing the physician) legitimately tries to access a certain patient record. The test scenario then checks if the legitimate request is actually granted through the (authorization) policies of the system.
- Context: e-learning and teaching system. In an online exam, a student illegitimately tries to access another student's test realm. A test scenario may then check, for example, that a) the access request is denied and b) an obligation policy is discharged which forces an "exam observer" object to send a message to the corresponding teacher to further investigate the incident.
- Context: online brokerage system. The price of a certain company share listed at the New York Stock Exchange drops below a predefined value. The test scenario checks if a respective obligation policy is discharged which forces the "stop loss observer" object to sell the corresponding stock.

However, most often it is not possible to perform test runs on live systems. To actually test the behavior of policy controlled system entities, test scenarios thus require a controlled runtime environment which allows to simulate the corresponding test situation. Therefore, we use mock objects in test scenarios (see Figure 4).

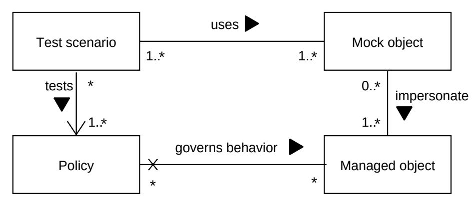

Figure 4. Concept formation: policy runtime tests

A *mock object* (see [5, 13]) is an object that impersonates a certain system entity, in particular a certain managed object. The mock object is a lightweight object (compared to the system entity it impersonates) that simulates some behavioral aspects of the corresponding impersonated system entity. Thus, mock objects are always used for behavior verification. With mock objects we can rapidly set up testing environments to check if the policy rules to be tested actually induce the correct (intended) behavior on the respective system entities.

3.2 Testing Policy Language Semantics

A *semantics test* checks if a certain policy expression yields the exact behavior induced by the corresponding policy language's static and dynamic semantics. In other words, the static and dynamic semantics of the policy language must never be violated. This means that semantics tests check actual (platform-specific) policy rules resulting from policy expressions that are specified via the respective policy language.

In general, semantics tests can be subdivided into static semantics tests and dynamic semantics tests. Static semantics tests check "semantic invariants". This means, they verify that the policy language's meta-model, including additional invariants on meta-model elements (see, e.g.,

[26]), are not violated and hold at any time. For example, testing of static semantics includes syntax checking during or after parsing the policy language expressions. Testing of dynamic semantics verifies that the language commands of a policy language behave exactly as specified. This means, the corresponding test scenarios check if a certain language command exactly yields the specified results. For example, in a role-based access control policy language, we have to test if the "createRole" command actually creates a role object, or if the "roleSubjectAssign" command correctly assigns a role to a subject.

Moreover, it is important to understand the differences between policy runtime testing (see Section 3.1) and testing of policy language dynamic semantics. In contrast to dynamic semantics testing, policy runtime testing verifies the correct behavior of platform-specific policy rules (see Section 3.1). The differences between policy runtime testing and dynamic semantics testing are thus similar to the differences between testing a general purpose programming language (e.g. C, C#, Java, Perl, Ruby, or Tcl) and testing actual programs written in that programming language. With respect to this analogy, the policy language then refers to the programming language and actual policies (respectively platform-specific policy artifacts) refer to the program.

3.3 User Acceptance Test

User acceptance testing checks if a software product is ready for deployment and operational use (see, e.g., [10]). In particular, the developers and users (the domain experts) agree upon a set of acceptance criteria that must be met by the respective software application. Subsequently, these criteria are checked to validate and verify the system's functionality. Scenarios are a popular means to describe behavior-based acceptance tests (see, e.g., [6, 7]). Scenarios on a user level then describe the action and event sequences that must be supported by the corresponding system (see also [8]). Moreover, the user-level acceptance tests (defined by domain experts) can be refined to executable test scenarios. Using this technique, user acceptance testing can be automated to a certain degree (see, e.g., [11, 14, 25, 27]).

Moreover, agile software engineering methods regularly apply scenario-based techniques, such as use cases and user stories, to model user requirements. In each feedback loop the developers and domain experts then check the existing test scenarios, add new test scenarios, or adapt test scenarios if necessary (see, e.g., [12, 15]).

User acceptance testing of a policy language has to validate that the policy language being build, meets the needs of the domain experts. In particular, we have to ensure that the policy language's expressiveness is adequate for the respective target domain, and that the concrete syntax of our policy language is convenient to use. Therefore, user acceptance testing is conducted in close cooperation with domain experts (as physicians and nurses for the health care domain; analysts, traders, and controllers for the financial services domain; or network specialists for the quality-of-service domain).

This way we establish multiple feedback cycles and make sure that the change requests of domain experts directly enter the engineering process. The close cooperation of software engineers and domain experts then yields a technically sound policy language with a user interface (the concrete syntax, see, e.g., [26]) which is tailored to the needs of domain experts. User acceptance testing is therefore just as important as technical testing of a policy language (which is described in the above sections).

4 STORM: A DSL for Scenario-Based Runtime Testing

STORM (Scenario-based Testing of Object-oriented Runtime Models) is an embedded DSL (see, e.g., [26, 33]) implemented in eXtended Object Tcl (XOTcl) [17]. STORM provides all features of scenario-based testing (see Section 2) and can, in principle, be used to test arbitrary software components. However, as a DSL for scenario-based testing, it is especially well-suited for policy runtime testing (see Section 3.1) and testing of a policy language's (dynamic) semantics (see Section 3.2).

As an embedded DSL, STORM directly benefits from the dynamic language features of XOTcl. The extensive reflection options of XOTcl allow to introspect the runtime structures of test cases and test scenarios. In addition, STORM preconditions, postconditions, test bodies, setup sequences, and cleanup sequences may include arbitrary Tcl/XOTcl source code and thus allows for an extensive flexibility and supports rapid test development. For example, native XOTcl objects can directly be used as mock objects. Moreover, because XOTcl is a homoiconic language[2] one can dynamically generate code in and from STORM test scenarios, e.g. to define or manipulate mock objects at runtime. In addition, as an embedded DSL, STORM can be extended in a straightforward manner via the flexible language features of its host language, such as mixins [34] for example. Thereby, STORM can serve as a base DSL for a family of other scenario-based testing DSLs that provide additional features required by specialized testing domains, such as testing of distributed/remote components or Web-based testing for example.

Figure 5 shows four UML interaction diagrams that depict the internal call sequences of STORM test cases and test scenarios (see also Section 2). When the `runTestScenarios` command is called, the respective test case first executes the corresponding setup script (the setup script defines the setup sequence of a STORM test case, see

[2] In *homoiconic languages* the source code is represented via the (or one of the) language's fundamental data type(s). Thereby, programs written in homoiconic languages can generate new and/or adapt existing source code dynamically to redefine or extend program behavior at runtime.

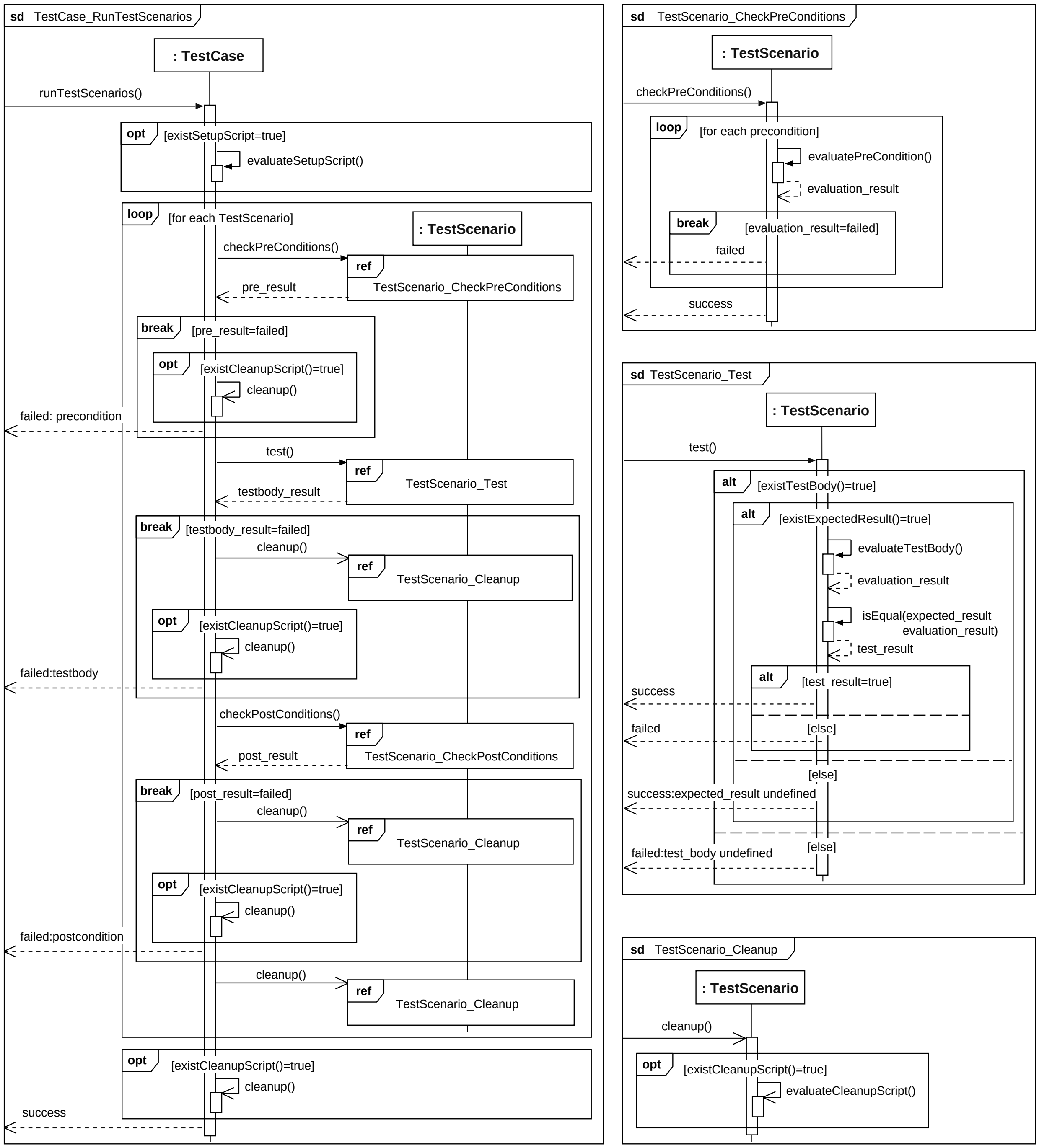

Figure 5. Internal call sequences of STORM test cases

also Section 2). Next, the test case evaluates its test scenarios. For each test scenario, the respective test case first checks if all preconditions hold. In case at least one precondition is violated, the test run is stopped. The test case then executes the test case cleanup script (the cleanup script defines the cleanup sequence of STORM test cases, see also Section 2) and returns a "failed" message indicating the violated precondition that caused the failure.

In case all preconditions of a test scenario hold, the `test` command of the respective test scenario is called. The `test` command then evaluates the test body. If the actual result of the test body (i.e. the value(s) or the object returned by the test body) matches the expected result of this test scenario a "success" message is returned, otherwise the `test` method returns a "failed" message (see Figure 5). A failed test body again stops the test run and triggers the cleanup script. If the evaluation of the test body is successful, however, the next step is to check the test scenario's postconditions. If at least one postcondition is violated the test run is stopped. In case all postconditions hold, the complete procedure is repeated for the next test scenario. In case all test scenarios are successfully executed, the corresponding test case returns a "success" message (see Figure 5). Note that (differing from Section 2) one may also choose *not* to define the expected result of a STORM test scenario. In this case, the evaluation of the test body results in a "success" message, with an additional comment indicating that the expected result for this scenario is undefined (see also Figure 5). This option is sensible a) to define test scenarios were the result of the test body shouldn't be tested or b) for test scenarios were it is difficult to predetermine the result of the test body (e.g. for functions choosing a random object from an object pool). All checks of the test scenario are then included in the corresponding pre and postconditions.

```
::STORM::TestScenario someIdentifier
  -test_body {
       arbitrary Tcl/XOTcl code }
  -expected_result someString
  -preconditions {
       list of arbitrary Tcl/XOTcl scripts
       to check preconditions }
  -postconditions {
       list of arbitrary Tcl/XOTcl scripts
       to check postconditions }
  -cleanup_script {
       arbitrary Tcl/XOTcl code }
```

Above we have the skeleton of a STORM test scenario. Because of the page restrictions, we had to cut the detailed example from the paper. However, on our Web page we provide an extended version of this paper where we reinserted the example. The STORM DSL syntax specification is found in Appendix A.

5 Related Work

In [28] Tsai et al. present an object-oriented and scenario-based test framework implemented in Java. In particular, they extended the JUnit test environment to load scenario information from a database before running test scenarios. Another scenario-based testing extension for JUnit is presented by Wittevrongel and Maurer [31]. They introduce SCENTOR as an approach for scenario-based testing of e-business applications. Wittevrongel and Maurer use UML 1.x sequence diagrams to define scenarios. These sequence diagrams then serve as input to write actual tests. In [21], Sadilek and Weißleder suggest an approach to test the abstract syntax of DSLs. In particular, they generate JUnit tests to check positive and negative example models for conformance with the respective abstract syntax. A test passes if the corresponding positive example model is actually an instance of the abstract syntax while the negative example model is not.

King et al. [9] propose a framework for self-testing of autonomic computing systems (ACS). In particular, they introduce so called "test manager components" that communicate with an ACS to dynamically validate change requests. This means, each time an ACS receives a change request for a managed resource, the corresponding test manager component needs to ensure that all functional and non-functional requirements are still being met after the change. Based on the results of this validation procedure, the ACS either accepts or rejects the change request. FIT for rules [4] is a testing framework for JBoss rules [19]. FIT for rules is built on FIT (Framework for Integrated Testing) and captures test scenarios in the form of HTML tables. In particular, all input data (functions calls, parameters, etc.) and assertions (e.g. expected results) of a test scenario are defined using a tabular layout. These test scenarios are then passed to the JBoss rule engine. Finally, the results produced by the rule engine are compared to the expected results.

In [3], Dietrich and Paschke suggest an approach to apply test-driven development to the definition of business rules. In [18], Paschke further discusses the verification, validation, and integrity checking of policies via test-driven development techniques. In [25], Zdun and Strembeck presented an approach for the use case and scenario-based testing of software components. Tests are derived from use case scenarios via continuous refinement. This means, tests are defined as a formalized and executable description of a requirements level scenario. The use case and test information can be associated with a software component as embedded component metadata. In particular, the approach provides a model-based mapping of requirements level scenarios to test scenarios, as well as (runtime) traceability of these links.

Ryser and Glinz [20] present the SCENT method. They describe natural language scenarios which are then formalized using a state chart notation. The state charts are annotated with additional test relevant information and provide the basis for the derivation of test cases.

6 Conclusion

In this paper, we presented an approach to test policy-based systems with scenarios. In particular, we implemented STORM as an embedded DSL for scenario-based runtime testing. It allows for the testing of dynamically changing system behavior and thereby directly supports testing of policy-based systems. However, STORM does not prescribe any particular type of development or testing process, and can be integrated with different testing approaches including test-driven development of policy languages and/or policy rules.

We used STORM to define test suites for a general purpose policy framework (71 test cases including 439 test scenarios), a policy decision point for RBAC policies (25 test cases including 322 test scenarios), and a number of different applications using these components.

To test policy rules with STORM, we define test scenarios which specify intended system behavior via action and event sequences that have well-defined observable effects on test objects (e.g. mock objects specifically designed for testing purposes). Subsequently, the corresponding test scenarios are executed and STORM checks if the actual (policy governed) behavior of the respective system entities yields the exact effects defined through the test sequences.

A STORM DSL Synatx

STORM::TestSuite

```
suite = "STORM::TestSuite" , identifier ,
  [" -setup_script" , " {" , script , "}"],
  [" -cleanup_script" , " {" , script , "}"],
  [" -detailed_report", (1 | 0)],
  [" -halt_on_first_error", (1 | 0)],
  [" -order" {identifier}];
identifier = letter | digit | other ,
             {letter | digit | other} ;
letter = "a"|"b"|..."z"|"A"|"B"|..."Z" ;
digit = "0"|"1"|"2"|..."9" ;
other = "-"|":"|"_" ;
script = "an arbitrary Tcl/XOTcl script" ;
```

STORM::TestCase

```
test_case = "STORM::TestCase" , identifier ,
  [" -setup_script" , " {" , script , "}"],
  [" -cleanup_script" , " {" , script , "}"],
  [" -preconditions" , " {" , condition_list , "}"],
  [" -postconditions" , " {" , condition_list , "}"],
  [" -order" {identifier}];
condition_list = condition , {condition} ;
condition = "{" , script , "}" ;
identifier = letter | digit | other ,
             {letter | digit | other} ;
letter = "a"|"b"|..."z"|"A"|"B"|..."Z" ;
digit = "0"|"1"|"2"|..."9" ;
other = "-"|":"|"_" ;
script = "an arbitrary Tcl/XOTcl script" ;
```

STORM::TestScenario

```
test_scenario = "STORM::TestScenario" , identifier ,
  " -test_body" , " {" , script , "}",
  " -expected_result" , "" , string ,
  [" -cleanup_script" , " {." , script , "}"],
  [" -preconditions" , " {" , condition_list , "}"],
  [" -postconditions" , " {" , condition_list , "}"];
condition_list = condition , {condition} ;
condition = "{" , script , "}" ;
identifier = string ;
string = letter | digit | other ,
         {letter | digit | other} ;
letter = "a"|"b"|..."z"|"A"|"B"|..."Z" ;
digit = "0"|"1"|"2"|..."9" ;
other = "-"|":"|"_"| " ";
script = "an arbitrary Tcl/XOTcl script" ;
```

References

[1] J. Carroll. Five Reasons for Scenario-Based Design. In *Proc. of the IEEE Annual Hawaii International Conference on System Sciences (HICSS)*, 1999.

[2] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *Proc. of the 2nd International Workshop on Policies for Distributed Systems and Networks (POLICY)*, January 2001. Lecture Notes in Computer Science (LNCS), Vol. 1995, Springer Verlag.

[3] J. Dietrich and A. Paschke. On the Test-Driven Development and Validation of Business Rules. In *Proc. of the International Conference on Information Systems Technology and its Applications (ISTA)*, May 2005.

[4] FIT for rules – Homepage. http://fit-for-rules.sourceforge.net/, [accessed September 2010].

[5] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes. Mock roles, not objects. In *Companion to the 19th annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2004.

[6] P. Hsia, J. Gao, J. Samuel, D. Kung, Y. Toyoshima, and C. Chen. Behavior-Based Acceptance Testing of Software Systems: A Formal Scenario Approach. In *Proc. of the International Computer Software and Applications Conference (COMPSAC)*, November 1994.

[7] P. Hsia, D. Kung, and C. Sell. Software requirements and acceptance testing. *Annals of Software Engineering*, 3, 1997.

[8] M. Jarke, X. Bui, and J. Carroll. Scenario Management: An Interdisciplinary Approach. *Requirements Engineering Journal*, 3(3/4), 1998.

[9] T. King, D. Babich, J. Alava, P. Clarke, and R. Stevens. Towards Self-Testing in Autonomic Computing Systems. In *Proc. of the International Symposium on Autonomous Decentralized Systems (ISADS)*. IEEE Computer Society, March 2007.

[10] H. Leung and P. Wong. A study of user acceptance tests. *Software Quality Journal*, 6(2), 1997.

[11] K. Leung and W. Yeung. Generating User Acceptance Test Plans from Test Cases. In *Proc. of the International Computer Software and Applications Conference - Vol. 2- (COMPSAC)*. IEEE Computer Society, July 2007.

[12] M. Lippert, P. Becker-Pechau, H. Breitling, J. Koch, A. Kornstädt, S. Roock, A. Schmolitzky, H. Wolf, and H. Züllighoven. Developing Complex Projects Using XP with Extensions. *IEEE Computer*, 36(6), June 2003.

[13] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: Unit testing with mock objects. In *Extreme programming examined*. Addison-Wesley, 2001.

[14] R. Martin. The Test Bus Imperative: Architectures That Support Automated Acceptance Testing. *IEEE Software*, 22(4), 2005.

[15] G. Melnik, F. Maurer, and M. Chiasson. Executable Acceptance Tests for Communicating Business Requirements: Customer Perspective. In *Proc. of the AGILE Conference*. IEEE Computer Society, July 2006.

[16] C. Nebut, F. Fleurey, Y. L. Traon, and J. Jezequel. Automatic Test Generation: A Use Case Driven Approach. *IEEE Transactions on Software Engineering (TSE)*, 32(3), March 2006.

[17] G. Neumann and U. Zdun. XOTcl, an Object-Oriented Scripting Language. In *Proc. of Tcl2k: 7th USENIX Tcl/Tk Conference*, February 2000.

[18] A. Paschke. Verification, Validation and Integrity of Distributed and Interchanged Rule Based Policies and Contracts in the Semantic Web. In *Proc. of the Workshop on Semantic Web Policy*, November 2006.

[19] M. Proctor, M. Neale, et al. JBoss Drools Documentation. Version 5.1, http://labs.jboss.com/drools/, August 2010.

[20] J. Ryser and M. Glinz. A Scenario-Based Approach to Validating and Testing Software Systems Using Statecharts. In *Proc. of the International Conference on Software and Systems Engineering and their Applications (ICSSEA)*, December 1999.

[21] D. Sadilek and S. Weißleder. Towards Automated Testing of Abstract Syntax Specifications of Domain-Specific Modeling Languages. In *Proc. of the Workshop on Domain-Specific Modeling Languages*. CEUR Workshop Proceedings, Vol.324, http://ceur-ws.org, March 2008.

[22] M. Sloman. Policy Driven Management for Distributed Systems. *Journal of Network and Systems Management*, 2(4), Plenum Press, December 1994.

[23] T. Stahl and M. Völter. *Model-Driven Software Development*. John Wiley & Sons, 2006.

[24] M. Strembeck. Embedding Policy Rules for Software-Based Systems in a Requirements Context. In *Proc. of the 6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, June 2005.

[25] M. Strembeck and U. Zdun. Scenario-based Component Testing Using Embedded Metadata. In *Proc. of the Workshop on Testing of Component-based Systems (TECOS)*, September 2004. Lecture Notes in Informatics (LNI), Vol. 58.

[26] M. Strembeck and U. Zdun. An Approach for the Systematic Development of Domain-Specific Languages. *Software: Practice and Experience (SP&E)*, 39(15), October 2009.

[27] D. Talby, O. Nakar, N. Shmueli, E. Margolin, and A. Keren. A Process-Complete Automatic Acceptance Testing Framework. In *Proc. of the IEEE International Conference on Software - Science, Technology & Engineering (SWSTE)*, February 2005.

[28] W. Tsai, A. Saimi, L. Yu, and R. Paul. Scenario-based Object-Oriented Testing Framework. In *Proc. of the Third International Conference on Quality Software (QSIC)*. IEEE Computer Society, November 2003.

[29] A. Vedamuthu, D. Orchard, F. Hirsch, M. Hondo, P. Yendluri, T. Boubez, and U. Yalcinalp. Web Services Policy 1.5 - Framework. http://www.w3.org/TR/ws-policy, September 2007. W3C Recommendation.

[30] M. Winslett. Policy-driven Distributed Authorization: Status and Prospects. In *Proc. of the International Workshop on Policies for Distributed Systems and Networks (POLICY)*, June 2007.

[31] J. Wittevrongel and F. Maurer. SCENTOR: Scenario-Based Testing of E-Business Applications. In *Proc. of the IEEE International Workshops on Enabling Technologies (WETICE)*, June 2001.

[32] R. Yavatkar, D. Pendarakis, and R. Guerin. A Framework for Policy-based Admission Control. IETF, RFC 2753 (Informational), http://ietf.org/rfc/rfc2753.txt, January 2000.

[33] U. Zdun and M. Strembeck. Reusable Architectural Decisions for DSL Design: Foundational Decisions in DSL Projects. In *Proc. of the 14th European Conference on Pattern Languages of Programs (EuroPLoP)*, July 2009.

[34] U. Zdun, M. Strembeck, and G. Neumann. Object-based and class-based composition of transitive mixins. *Information and Software Technology*, 49(8), August 2007.