

Supporting Authorization Policy Modification in Agile Development of Web Applications

Steffen Bartsch
TZI, Universität Bremen
Bremen, Germany
Email: sbartsch@tzi.org

Abstract—Web applications are increasingly developed in Agile development processes. Business-centric Web applications need complex authorization policies to securely implement business processes. As part of the Agile process, integrating domain experts into the development of RBAC authorization policies improves the policies, but remains difficult. For policy modifications, high numbers of options need to be considered. To ease the management task and integrate domain experts, we propose an algorithm and prototype tool. The AI-based change-support algorithm helps to find the suitable modification actions according to desired changes that are given in policy test cases. We also present a prototype GUI for domain experts to employ the algorithm and report on early results of non-security experts using the tool in a real-world business Web application.

Keywords-Agile Development; Authorization Policy Development; Policy Change Management; Change-Impact Analysis

I. INTRODUCTION

Web applications have become an increasingly popular choice for implementing key applications. One of the aspects that lead to the success is that Web applications may be developed in comparatively small teams. *Agile development* principles [1] are employed to increase the productivity of development teams. To improve software development, Agile development features close customer collaboration and short-term planning, resulting in frequent changes of requirements. Thus, achieving secure software in Agile development is different from classical waterfall development models.

As Web applications are often custom-built, the development of authorization policies¹ move from the administration into the development realm. With continuously changing requirements and early deployment, authorization rules are developed continuously as well. Authorization rules need to be constantly adapted to application changes when new user groups are introduced to the application or some of the existing users should be excluded from new functionality. In order to develop suitable authorization rules, close customer collaboration is required. This way, the development

¹We use the term policy to describe a set of authorization rules that define the privileges of a system's users. Authorization rules consist of RBAC constructs, including user assignments and authorization constraints.

team prevents inefficiencies that are caused by overly strict permissions [2]. Ideally, developers, administrators, and end-users should take part in authorization development [3], [4].

Maintainability is a key aspect for authorization policies, in particular in Agile development models. A large amount of research has been invested into the authorization realm resulting e.g. in *Role-based Access Control* (RBAC, [5], [6], [7]). While RBAC policies promise good maintainability, taking advantage of abstraction levels and hierarchies of roles and privileges, these features make the policies at the same time hard to entirely understand. Also, when modifying a policy, there are often multiple ways to reach a given goal. The purpose of the algorithm and tool that are presented in this paper is to support domain experts in changing existing policies and thus allow a tighter integration into the authorization part of the development process.

The following example policy for a simple conference management Web application illustrates the overwhelming number of options. In the system, a user with an *Attendee* role should receive permissions to modify conferences. The role *Attendee* includes the role *Guest*, the privilege *Manage* includes *Modify*. The following options are available to reach the desired modification:

- Assign an existing role to the user that has the manage or modify privilege (*Conference organizer, Administrator*);
- Add privileges modify or manage to roles attendee or guest;
- Add privileges modify or manage to a new role for users with conference modification permission and assign it to the user.

Even in this simple example with only four roles, there are already eight possible modifications of the policy. Security experts may intuitively rule out most options in this simple example, but might need support in choosing the best suited option in more complex cases. When working closely together with end-users, even explaining an existing policy may be hard. Thus, finding and keeping up with the wealth of modification options asks too much of end-users, who prefer simple and more concrete but often unmaintainable access control lists.

In this paper, we propose an algorithm and a tool to

```

# setup specific objects: an_attendee, a_talk
the_user an_attendee do
  should_be_allowed_to :read, :conferences
  should_not_be_allowed_to :update, :conferences
  should_be_allowed_to :read, a_talk
  should_not_be_allowed_to :update, a_talk
end

```

Figure 1. Authorization Policy Test Example

support the modification of authorization policies as part of the development process. An Artificial Intelligence algorithm is employed to generate the possible sequences of policy modification actions to reach a given goal. The modification goal is specified by test cases for authorization rules. To support domain as well as security experts in choosing the most suitable modification for the specific goal, we developed and present a prototype GUI for the algorithm. Lastly, we report on early results of employing this tool in a real-world business application of a small and medium enterprise (SME).

II. AUTHORIZATION POLICY TEST CASES

For modifying an authorization policy, test cases can define the goal state that is to be reached. The formulation of a goal state is in particular necessary if an algorithm should suggest change actions. One approach that is successfully applied in software engineering is extensive testing as in Test-Driven Development (TDD) [8]. In TDD each desired behavior of the application is specified by way of tests, followed by the implementation until the tests succeed.

To define the goal of an authorization policy modification, we carried TDD over to the development of authorization rules. A simple and readable test language based on the Ruby programming language is used to define the test cases. As shown in the example in Figure 1, for a specific user, *positive* (“should be allowed”) and *negative* (“should not be allowed”) permission tests are specified either for object types as a whole (i.e. all `:conferences`) or specific objects.

The readability of the test cases is an important aspect to tightly involve end-users in the development of authorization rules. The test cases serve as an additional basis for discussions, together with the authorization rules. Instead of explicit documentation of the desired permissions, the code is used and is thus always up-to-date and in use for validating the authorization rules. In contrast to an authorization policy, the semantics are focused on specific cases to be checked. While abstract rules may at times be hard to understand for non-experts, these example-like test cases have shown to be easier to use in discussion. Thus, authorization test cases are a good way of defining the goal of modifications through a change-support algorithm.

III. CHANGE ASSISTANT

RBAC specifications tend to be complex and thus difficult for humans to thoroughly understand. When tasked with extending or reducing the permissions of certain users, it is important to evaluate the different options with respect to the consequences of certain changes. Security experts are trained to see the typical patterns and may thus more easily rule out most options intuitively. Still, even experts have to know the general structure of the existing authorization rules to draw the right conclusion. For non-experts, change support is even more essential if authorization rules are to be kept maintainable.

To suggest possible modifications to achieve a goal and to show their consequences, we developed the Change Assistant algorithm to offer change support. In practice, there is a limited number of modification actions that may be applied to the authorization rules. For removing permissions, role assignments may be removed from certain users, permissions may be removed from roles or role or privilege hierarchies may be modified. For extending permissions, the Change Assistant may employ the reverse actions of assigning roles to users or permissions to roles and hierarchy modifications. In addition, new roles might be introduced. The overall goal of the modifications is specified in the above-described test cases of users’ allowed and disallowed permissions. The general idea of the Change Assistant is to combine any of the actions into sequences of modifications that each sequence fulfills all the given test cases.

A. Algorithm

The Change Assistant algorithm is based on the well-known AI best-first search algorithm A* and planning algorithms [9]. The main algorithm for the Change Assistant is shown in Figure 2. The following types are used in the algorithm:

Candidate : $AuthRules \times \mathcal{P}(User) \times Modifications$

AuthRules : $\mathcal{P}(Role \times Privilege \times Type)$

Role : Set of all system roles

Privilege : $\{create, update, \dots\}$

Type : Set of all system object types

User : Set of all system users

Modifications : $SpecificAction^*$

SpecificAction : $AbstractAction \times Parameters$

AbstractAction : Set of abstract actions (Sect. III-B)

Parameters : Parameters of abstract action (Sect. III-B)

TestCase : $\{allow, prohibit\} \times User \times Privilege \times Type$

Beginning with an initial *candidate* derived from the original state, the given *test cases* are solved by generating

```

Input: candidates = [[current_auth_rules, users,  $\emptyset$ ],
non-empty(test_cases)
Output: non-empty(solutions)
while non-empty(candidates) and steps < max_steps do
  candidate  $\leftarrow$  remove-first(candidates)
  if is_positive(last_failing_test(candidate)) then
    abstract_actions  $\leftarrow$  positive_abstract_actions
  else
    abstract_actions  $\leftarrow$  negative_abstract_actions
  end if
  for abstract_action in abstract_actions do
    specific_actions  $\leftarrow$  generate_specifics_for(candidate,
abstract_action)
    for specific_action in specific_actions do
      if specific_action is prohibited by user or reversal
of previous action or superset of known solution
then
        next
      end if
      new_cand  $\leftarrow$  apply(candidate, specific_action)
      check(new_cand, test_cases)
      if empty(failing_tests(new_cand)) then
        append(solutions, new_cand)
      else
        append(candidates, new_cand)
      end if
    end for
  end for
  candidates  $\leftarrow$  sort_by_heuristic(candidates)
end while

```

Figure 2. Change Assistant Algorithm

new candidates with modifications. Each modification is derived in a two-step process: First, the algorithm chooses general modification options, *abstract actions*, that match the first failed test. Then, each abstract action is expanded into *specific actions*.

The algorithm generates specific actions from the abstract actions with the matching effect through *generate_specifics_for*. From an abstract action, e.g. “Remove a Permission from a Role”, result several specific actions, e.g. “Remove Permission *Read* from Role *User*.” Depending on the specific failing test that is targeted in each step, only a limited number of specific actions solve the failing test and are thus considered. The abstract actions’ expansions are listed below. The expansions are described informally only because of space limitations. Apart from atomic abstract actions, compound actions are employed that combine multiple actions to guarantee solving one test in each step. This limits the number of branches in the decision tree.

The validity of each new candidate is tested through multiple properties. The candidate should not reverse any of

the previous actions and should not contain any action that the end-user has prohibited (cf. following section). Also, the candidate is rejected if it is a superset of an already known valid solution. *apply* is used to create a new candidate based on the previous one with the current modifications applied. If the new candidate has no more failing tests, it is added to the list of valid solutions. Otherwise, it is added to the list of candidates for further modifications in further steps. A heuristic function is used to sort the remaining candidates [9]. Thus, the most promising candidates are considered first. The heuristic function for a given candidate c is defined as

$$f(c) = |\text{failing_tests}(c)| + \sum_{i=1}^{|\text{steps}(c)|} \text{step_weight}(c, i)$$

The step’s weight depends on how invasive the change is. The weight is assigned to each abstract action. Thus, the simplest solutions are considered best.

The scalability of the Change Assistant algorithm depends on the number of test cases. Each test case results in a branching of the decision tree. Negative tests have a static branching factor of two. For positive tests, the branching factor depends on the existing authorization rules: the number of roles as well as the inheritance levels in privileges and roles. To guarantee the termination of the algorithm, a maximum count of considered candidates is enforced. With the best candidates being considered first, it is sufficient to search only a part of the solution space.

The current implementation of the Change Assistant algorithm is tailored to the *declarative_authorization* authorization model. [10] As the authorization model is similar to the standard RBAC model, the proposed algorithm should be applicable to other RBAC policies as well.

B. Abstract Actions

As described above, test cases are either positive (“should be allowed”) or negative (“should not be allowed”). For both types of test cases, there are abstract actions that have the matching effect to solve the failing test. From abstract actions, specific actions are derived, defining the specific modifications to apply to the authorization rules. First, abstract actions and derived specific actions for positive tests are described, followed by those for negative tests.

1) Abstract actions for positive tests:

Assign Privilege to Role: needs to consider which privileges are to be assigned to which roles. All the roles assigned to the user in the current test case and descendant² roles may be used. As privilege, the privilege in the test case and its ancestors are considered.

Assign Role to User: has to choose from the present roles which to assign to the user in the test case. Considered roles are all that include the privilege in the test case and its ancestors. Also, all ancestor roles may be employed.

²Ancestors include the role or privilege, descendants are included. The role *Guest* is often a descendant of *User*, *User* an ancestor of *Guest*.

Create Role and Assign Role to User: is a compound action. The action creates a role, assigns a privilege to the role and then assigns the new role to the user in the test case. The action has to decide on the privileges to assign to the new role. The privileges are chosen from the one in the test case and all ancestor privileges.

Add Privilege to Role and Assign Role to User: is a compound action and assigns one of the present roles to the user in the test case. To solve the failing test, it also assigns the necessary privileges to the role. As privileges, the privilege in the test case and its ancestors may be used. All roles that are not yet assigned and have together with their descendants none of the privileges assigned are considered.

2) Abstract actions for negative tests:

Remove Privilege from Role: has to find the correct privilege to remove and the role to remove it from to solve the failing test. It searches all roles of the user in the test case and their descendants for the test's privilege or one of its ancestors. If the user has the privilege through multiple roles or privileges, this action removes all of these.

Remove Role from User: solves the failing test by removing the role(s) from the test's user that provide the privilege of the test. The action looks for all roles of the user that have, directly or through a descendant, the privilege or an ancestor of the privilege assigned.

C. Examples

To demonstrate how the Change Assistant algorithm works, we come back to the example given in the introduction. The user *Attendee 1* with the role *attendee* should receive the permissions to modify conferences. This goal is formulated as a policy test as described in Section II. As depicted in Figure 3, the algorithm starts off with the policy before any changes. This is the initial candidate. When this candidate is checked against the goal policy test, the test fails. To solve this test failure, the abstract actions for positive test cases are considered. In Figure 3, abstract actions are shown as octagons.

Each of the chosen abstract actions is instantiated for the failed test into specific actions, shown in Figure 3 in the lower row. Specific actions are validated as described in the previous section. Each valid specific action is then applied to the initial candidate to create new candidates. The new candidates are checked against the policy tests. With all of the test cases succeeding on the generated candidates, no further operations are necessary and the identified approaches may be presented to the user.

In a more complex example, the intended policy change may be indicated through several test cases. While there might be one attendee who should be able to modify conferences, maybe others should not. In this case, adding the role "conference organizer" to Attendee 1 still solves all policy test cases in one step. Adding the permission to the attendee role would require removing this role from the other users,

though. Thus, for each candidate with remaining failing test cases, suitable abstract and specific actions are generated and applied as described above. In this way, the algorithm creates valid approaches with multiple modification steps.

IV. USER INTERFACE

The above-described Change Assistant algorithm takes existing authorization rules and conditions that specify the intended changes as inputs and derives several possible suggestions. To formulate the conditions as test cases and evaluate the resulting suggestions, end-users need additional support that we provide through a GUI.

We have integrated the support GUI into the authorization framework `declarative_authorization` for *Ruby on Rails*³ that we have recently proposed [10]. The Change Assistant user interface follows the ideas of *Programming by Example* user interfaces [11], [12]. The GUI is integrated into the web application as part of the authorization management back-end. In Figure 4, the GUI is depicted as part of a demo application for managing conferences⁴. The GUI works in four steps:

- 1) Choosing the permission that should be changed, such as "creating conferences" in the demo application. Although the Change Assistant algorithm supports the mixing of different permissions in the test cases, the GUI currently limits the tests to one permission to decrease the complexity of the user interface.
- 2) Answering basic questions to improve the result by taking the end users' intentions into account. Currently, the user is queried whether she would like to affect only few or many system users.
- 3) Defining the test cases by deciding which system users should or should not have the above-chosen permission. Yellow markers show for each user whether she currently has the permission. Typically, end-users define permission tests for several users, both positive and negative, to improve the precision of the algorithm suggestions. It also helps to indicate for which users the permissions should remain unchanged.
- 4) Requesting and reviewing suggestions. The suggestions are listed, ordered by the complexity of the changes and the number of affected users. Similar results are grouped if the same abstract actions are used. A graphical representation may be requested for each of the suggestions to better understand the changes. To improve the results, specific actions may be prohibited, removing all suggestions that make use of the specific action. The prohibitions may either be very specific ("Don't assign role Admin to User A") or partly defined ("Don't assign role Admin"). The

³<http://rubyonrails.org/>

⁴`declarative_authorization` Demo Application at http://github.com/stffn/decl_auth_demo_app

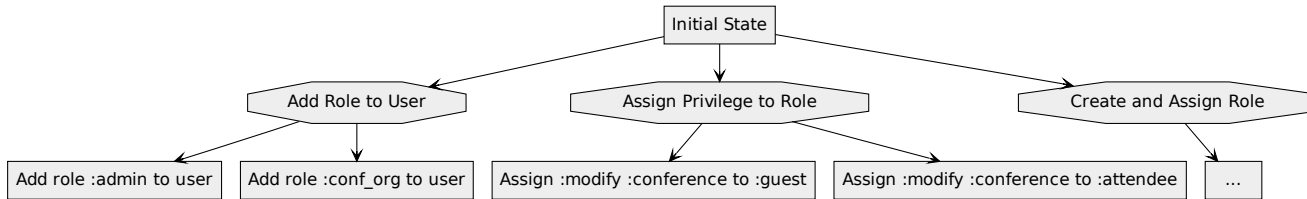


Figure 3. Simple Change Assistant Example

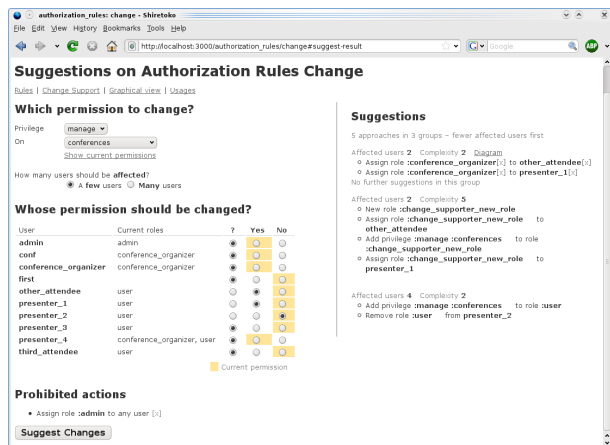


Figure 4. Change Assistant GUI

end-user may repeat this step in a drill-down process until the suggestion count is decreased to a number that allows an appropriate choice of modifications.

V. EARLY RESULTS

The Change Assistant algorithm and user interface was tested in a real-world business Web application of an SME. The application currently has 13 roles and 31 object types. On a standard PC, the current algorithm implementation has response times of under 1 second, depending on the number of conditions and resulting suggestions.

To test the Change Assistant’s real-world viability, we asked domain experts that use the Web application frequently but are no security experts with finding a solution to a typical change of authorization rules. One user was to receive additional permissions while his co-workers’ permissions were to remain unchanged. By using the Change Assistant GUI, the domain experts could decide to create a new role for this specific type of users with the additional permissions.

The overall feedback from the domain experts was positive. The domain experts had previously asked to take a more active part in the definition of authorization policies, but had commented on the textual representation as being too complex to understand well enough to directly edit it. When employing the Change Assistant prototype, the domain experts were able to formulate the intended changes.

Two main problem areas have been identified by working with the domain experts. Firstly, the number of modification options offered by the Change Assistant is still high. Although the grouping and drill-down mechanisms help, we would still like to improve the selection of relevant solutions. The second issue is the difficulty of understanding the application’s semantics of object types and privileges. We will experiment with tighter integration of the Change Assistant into the application to solve this issue. One option are links directly from certain parts of the application to the corresponding object types in the Change Assistant GUI.

VI. RELATED WORK

Several approaches have been suggested for helping organizations to define suitable RBAC policies. In *role mining*, existing information such as access control lists (ACLs) are employed to derive suitable roles [13], [14], [15]. Also, commercial products are in use that employ role mining technology, such as CA’s Role & Compliance Manager [16]. While role mining certainly helps in defining roles and privilege assignments, it has an organization-global and one-shot approach. The Change Assistant offers a more agile approach instead by not defining all roles in one step, but focusing on one specific problem to solve at a time. Also, the Change Assistant tries to integrate end-users in a tighter way than what is possible in role mining tools.

Change-impact analysis has been researched as another way of supporting authorization policy change. Fisler et al. demonstrate a formal approach to change-impact analysis with their Margrave tool employing BDDs [17]. In Margrave, specific properties of XACML policy changes can be queried. The semantics of Margrave queries cover a broader scope than those of the Change Assistant, but are harder to formulate for end-users. The Change Assistant gives higher priority to end-users’ understanding of the consequences of suggested changes.

Margrave may not only be employed to analyze changes, but also to guarantee certain policy properties. Formal analysis of access control policies is also shown in [18], [19]. Further approaches for property validation are model checking or the analysis of UML models [20], [21]. In model checking approaches, all possible states are checked if certain properties always hold. The testing-based approach in this paper has not the same scope and completeness of

the above-mentioned approaches. Still, it helps the end-user in keeping important properties intact through examples. In addition, formulating examples is easier for end-users than formulating abstract conditions. Further policy testing approaches can be found in [22].

VII. CONCLUSION

In this paper, we described the problems related to managing authorization policies with a focus on Agile development and Web applications. To support the authorization management in those areas, we proposed the Change Assistant algorithm to help applying changes to authorization rules. As the tool is aimed at end-users, we developed the Change Assistant GUI, helping end-users to take part in the authorization development process. Early results from deploying the Change Assistant in a real-world business application have been promising. The Change Assistant should allow a tighter integration of domain experts into the authorization policy development part of the software development.

In future work, we will further improve the end-user support in the GUI by coupling it more tightly to the application for easier selection of the desired permissions that are to be modified. To increase the maintainability of authorization rules, we would like to combine the Change Assistant with change management, recording the applied changes and corresponding test cases. Finally, we will look at larger authorization policies to test the Change Assistant's viability for further domains.

REFERENCES

- [1] A. Cockburn, *Agile Software Development*. Addison-Wesley Professional, December 2001.
- [2] L. Church, "End user security: The democratisation of security usability," in *Security and Human Behaviour*, 2008.
- [3] H. Lieberman, *End user development*. Springer, 2006.
- [4] M. E. Zurko and R. T. Simon, "User-centered security," in *NSPW '96: Proceedings of the 1996 workshop on New security paradigms*. New York, NY, USA: ACM, 1996, pp. 27–33.
- [5] D. Ferraiolo and R. Kuhn, "Role-based access controls," in *15th NIST-NCSC National Computer Security Conference*, 1992, pp. 554–563.
- [6] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.
- [7] A. I. 359-2004, *Role-Based Access Control*. American Nat'l Standard for Information Technology, 2004.
- [8] D. Hamlet and J. Maybee, *The Engineering of Software: Technical Foundations for the Individual*. Addison-Wesley, 2001.
- [9] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed., ser. Prentice Hall series in Artificial Intelligence. Upper Saddle River: Prentice Hall, 2003.
- [10] S. Bartsch, K. Sohr, and C. Bormann, "Supporting Agile Development of Authorization Rules for SME Applications," in *3rd International Workshop on Trusted Collaboration (TrustCol-2008)*. Springer, 2009.
- [11] B. A. Myers, R. McDaniel, and D. Wolber, "Programming by example: intelligence in demonstrational interfaces," *Commun. ACM*, vol. 43, no. 3, pp. 82–89, 2000.
- [12] H. Lieberman, Ed., *Your wish is my command: programming by example*. San Francisco: Morgan Kaufmann, 2001.
- [13] M. Kuhlmann, D. Shohat, and G. Schimpf, "Role mining - revealing business roles for security administration using data mining technology," in *SACMAT '03*. New York, NY, USA: ACM, 2003, pp. 179–186.
- [14] J. Schlegelmilch and U. Steffens, "Role mining with orca," in *SACMAT '05*. New York, NY, USA: ACM, 2005, pp. 168–176.
- [15] J. Vaidya, V. Atluri, and Q. Guo, "The role mining problem: finding a minimal descriptive set of roles," in *SACMAT*, V. Lotz and B. M. Thuraisingham, Eds. ACM, 2007, pp. 175–184.
- [16] M. Liou, "Role management and identity compliance: The business imperatives," Online, April 2009. [Online]. Available: http://www.ca.com/files/WhitePapers/role-management-identity-compliance-wp_203735.pdf
- [17] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies," in *ICSE*, G.-C. Roman, W. G. Griswold, and B. Nuseibeh, Eds. ACM, 2005, pp. 196–205.
- [18] D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Specifying and reasoning about dynamic access-control policies," in *IJCAR*, ser. Lecture Notes in Computer Science, U. Furbach and N. Shankar, Eds., vol. 4130. Springer, 2006, pp. 632–646.
- [19] N. Li and M. V. Tripunitara, "Security analysis in role-based access control," *ACM Trans. Inf. Syst. Secur.*, vol. 9, no. 4, pp. 391–420, 2006.
- [20] A. Schaad, V. Lotz, and K. Sohr, "A model-checking approach to analysing organisational controls in a loan origination process," in *SACMAT*, D. F. Ferraiolo and I. Ray, Eds. ACM, 2006, pp. 139–149.
- [21] K. Sohr, M. Drouineaud, G.-J. Ahn, and M. Gogolla, "Analyzing and managing role-based access control policies," *IEEE Trans. Knowl. Data Eng.*, vol. 20, no. 7, pp. 924–939, 2008.
- [22] J. Hwang, E. Martin, T. Xie, and V. C. Hu, "Policy-based testing," 2008, entry submitted for publication in the Encyclopedia of Software Engineering.