Mining Behavior Models from Enterprise Web Applications

Matthias Schur
SAP AG
Darmstadt, Germany
matthias.schur@sap.com

Andreas Roth
SAP AG
Karlsruhe, Germany
andreas.roth@sap.com

Andreas Zeller
Saarland University
Saarbrücken, Germany
zeller@cs.uni-saarland.de

ABSTRACT

Today's enterprise web applications demand very high release cycles—and consequently, frequent tests. Automating these tests typically requires a *behavior model:* A description of the states the application can be in, the transitions between these states, and the expected results. Furthermore one needs *scripts* to make the abstract actions (transitions) in the model executable. As *specifying* such behavior models and writing the necessary scripts manually is a hard task, a possible alternative could be to *extract* them from existing applications. However, mining such models can be a challenge, in particular because one needs to know when two states are equivalent, as well as how to reach that state. We present ProCrawl (PROcess CRAWLer), a generic approach to mine behavior models from (multi-user) enterprise web applications. ProCrawl observes the behavior of the application through its user interface, generates and executes tests to explore unobserved behavior. In our evaluation of three non-trivial web applications (an open-source shop system, an SAP product compliance application, and an open-source conference manager), ProCrawl produces models that precisely abstract application behavior and which can be directly used for effective model-based regression testing.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools, tracing*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Documentation*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*

General Terms

Algorithms, Design, Experimentation

Keywords

Specification mining; dynamic analysis; model-based testing

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
ACM 978-1-4503-2237-9/13/08
http://dx.doi.org/10.1145/2491411.2491426

1. INTRODUCTION

Today's enterprise web applications[1] are characterized by a high frequency of updates. To prevent such updates from breaking functionality, one has to *test*—and frequent updates call for frequent tests. Automating such tests requires a *model* describing the possible and the expected application behavior. However, typically web applications come without explicit models, which implies mostly manual and thus less efficient testing—but also slows down understanding and maintenance of the application.

The field of *specification mining* aims to facilitate these activities by mining abstractions from programs and their executions—typically, models of the program's behavior. If these models are precise enough, they can even be used as post-facto specifications of the program. Specification mining has been used to successfully derive axiomatic specifications such as function and data invariants from programs [8], or finite state automata describing states and transitions for individual classes [4, 6]. For such *small-scale* domains, it is fairly easy to validate specifications, because both program code and program state are accessible and amenable to symbolic reasoning and exhaustive testing.

For enterprise applications, extracting models that describe their possible behavior is much more difficult. Program code and program state, for instance, may not be available for analysis, as the application may be distributed across several layers and sites. In general, the only assumption that can be made is that there is some *user interface* (UI) such as a web interface that allows for human interaction.

In this paper, we present ProCrawl, a fully automatic tool that *mines explicit behavior models of enterprise web applications* for the sake of *system testing* and *maintenance.* All ProCrawl requires is the URL of a web application[2], login credentials for its users, a scope definition (i.e. the parts of the web application to be observed) and a start event. The resulting behavior model is a finite state automaton (FSA) in which the nodes denote abstract individual states of the web application, numbered in the order they were detected by ProCrawl, whereas the transitions denote actions that change the state and are performed by users acting in different roles. This model may serve as a "*gold standard*" oracle [3] detecting regressions which eliminate behavior from one revision of the application to the other.

[1] Enterprise systems are about "display, manipulation, and storage of data and the support or automation of business processes with that data" [10]

[2] ProCrawl handles rich Web 2.0 applications, including dynamic technologies such as AJAX.

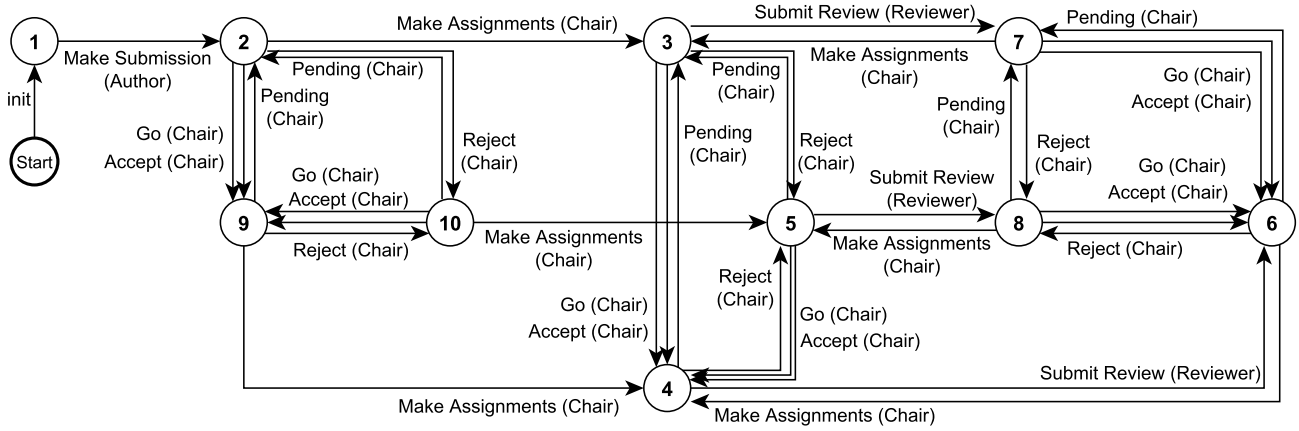

Figure 2: The ProCrawl behavior model for OpenConf, detailing the peer-review process involving authors, reviewers, and chairs. The model is unaltered ProCrawl output; only the graph layout was manually adjusted.

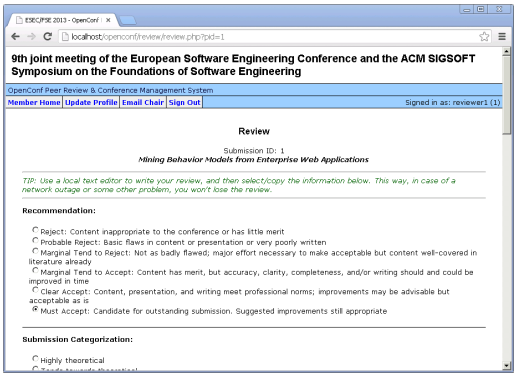

Figure 1: The OpenConf conference manager.

As an example, Figure 1 shows the reviewer's page of OpenConf[3], a web-based conference management system. Using OpenConf, authors can submit papers which are then reviewed by peers and finally rejected or accepted (a process we assume is familiar to most readers of this paper). Using a login script for an author, a reviewer and a chair each, ProCrawl infers the life cycle of a submission (cf. Figure 2): After the author makes a submission, the state of the submitted paper is *pending* (State 2). The chair can *accept* (State 9) or *reject* (State 10) the paper, before assigning a reviewer. For each of the three *unassigned* states there is a corresponding state after the assignment of a reviewer (State 3, 4, 5) and after the reviewer submitted the review (State 7, 6, 8). At all times, the chair can revoke an earlier decision or assign a new reviewer. However, after assigning reviewers, there is no way back to the unassigned state (2, 9, 10). In order to completely explore the behavior in the unassigned state, ProCrawl requires a command to reset OpenConf to State 1.

To extract such a model, ProCrawl observes the behavior of the web application via the UI. From its observations, ProCrawl infers *state abstractions* allowing it to determine whether a newly detected state is similar to a known one (which implies a cycle in the model) or whether it is not (in which case it generates and executes test cases to explore the yet unobserved behavior). This state abstraction is crucial for the effectiveness of ProCrawl: While efficient web crawling tools have been presented before, they do not make their underlying model explicit [5], or detect inappropriate process states [13], because they take only a single user and view of the web UI as basis for state abstraction.

To cater for the specific requirements of enterprise applications, ProCrawl can handle multiple users, even simultaneously (which is crucial for modeling the interaction of, say, vendors and clients); it assumes that each such user comes with an *init script* for logging into the system. For guiding the mining procedure, the *mining scope*, i.e. the views of the SUT's UI that shall be observed, as well as a start action (e.g. *Make Submission*) have to be configured. For some applications additionally, a command for resetting the application's state has to be provided; otherwise, in our example, ProCrawl could not return to states before paper assignments were made.

After discussing the requirements for analyzing and testing enterprise web applications (Section 2), we make the following *contributions*:

1. To the best of our knowledge, ProCrawl is the first tool to extract *abstract behavior models from multi-user web applications*. Section 3 discusses how ProCrawl works, and Section 4 gives details on the implementation.

2. In our evaluation of ProCrawl on three enterprise systems (an SAP web application, the OXID eShop, and the OpenConf conference manager), behavior models as mined by ProCrawl precisely capture the essentials of the underlying process. Section 5 introduces the case studies and Section 6 reports the results.

3. The resulting behavior models can be used directly as input to *model-based testing*. In Section 7, we show how automated tests following the mined behavior models detect disruptive process changes.

After discussing threats to validity (Section 8) and the related work (Section 9), Section 10 closes with conclusion and future work.

[3]http://www.openconf.com

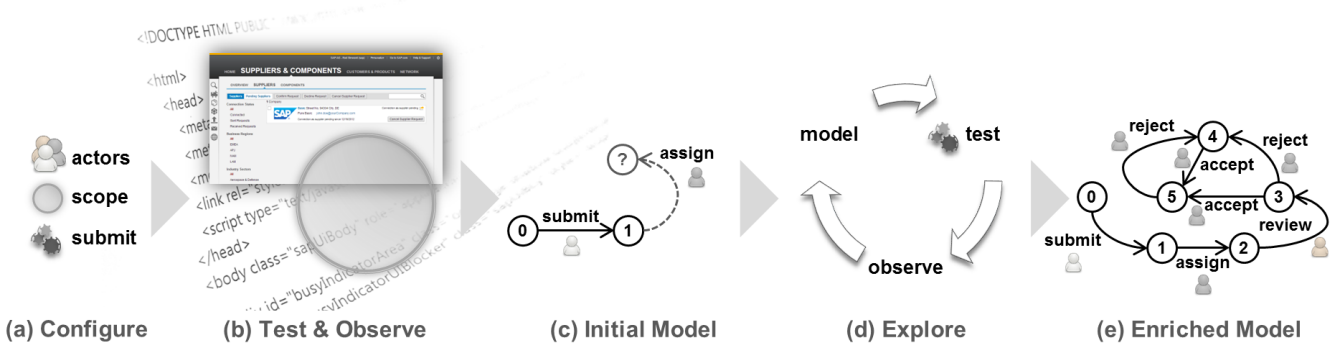

Figure 3: How ProCrawl works. ProCrawl takes a configuration (a) describing individual actors and their initial data, e.g. logins. It then triggers the start event (here: *submit*) in the web application (b) and determines the application state. This leads to an initial behavior model (c), consisting of the start event and the observed states. ProCrawl then systematically and automatically generates further executions (d) to explore additional states and transitions. The result is an enriched behavior model (e) obtained from the web application.

2. ENTERPRISE APPLICATION TESTING

Enterprise web applications support *business processes* centered around data stored in a *back end* which is manipulated when executing these business processes. As typical business processes involve *multiple interacting roles* (such as a seller and a vendor, or an author, a reviewer, and a conference chair), the central data is collaboratively edited through clients. These clients may have varying functionality depending on the user role and typically consist of several UI *views*, separating the functionality of the application. In state-of-the-art architectures the back end is accessed by browser-based clients. An illustrative example is OpenConf, while more business-oriented ones are presented in Section 5.

To manage complexity, the *testing of enterprise web applications* usually takes place on several layers [20]: *Unit tests* ensure that software units are functionality correct in isolation with the goal of high code coverage, while *service* and *integration tests* ensure the fulfillment of contractual obligations of a service/component and correct communication between coupled services/components. In this paper, we are focusing on the top most level in this hierarchy where the system undergoes a cross module *system test*, replaying the business processes that the application shall support. Note that on this level we are not concerned with fully covering the source code through the tests but rather in successfully executing all required business processes from an end-user perspective, i.e. testing through the UI with multiple users acting in different roles. In case of a system which integrates many services/components, this has the advantage that access to the source code is not required. However it is a valid assumption that we can at least *reset* the system state to the initial one, e.g. by resetting the application's database.

The release frequency in today's industrial practice of engineering enterprise applications (especially of applications deployed "in the cloud") is usually very high. Therefore *regression testing* [15] which ensures that functionality is preserved from one revision to the other on the system level needs to be highly efficient and automated.

With the aim of such increased test automation, Model-based Testing (MBT) [18] has expanded the automation of software testing towards the test design phase: behavior models can be used to derive a test suite which is then executed through a test automation framework. Today, industrial application has shown positive effects of MBT on development productivity (e.g. [11]), however the wide-scale adoption suffers from the absence of explicit models for testing and maintenance. Especially for regression testing, automatically *mining* such models reflecting the actual behavior of a running enterprise system ("gold standard" [3]) would be a significant step in pushing test automation even further.

Overall speaking, such models need to cover as much required behavior from the informal documentation, user stories, etc. as possible, but not more (*precision and recall*, see Section 5). Moreover they should be *correct* w.r.t. the application implementation, i.e. MBT test generators should transform them into reproducible program executions. These executions should ideally only lead to failures if paths through the business process implemented in the predecessor version are no longer possible. We will revisit these high-level goals in Sections 5 to 7.

Technically, these considerations lead to a number of essential *requirements* a model needs to satisfy to serve as reasonable input for model-based regression testing of enterprise web applications:

- First, models need to provide an *appropriate abstraction of the back-end state*, reflecting the fact that both the user interaction *and* the data stored in the back end determine the application behavior.
- Second, business processes typically span across multiple users in *different roles*. Hence, the state abstraction in the test model must take into account the combination of states from multiple user sessions. Moreover tests which cut across all implemented processes are not maintainable and thus need to be *modularized*, i.e. split in one test model per business process.
- Third, though UI tests are the perfect means to test user-level requirements, they are volatile as the UI changes frequently. To prevent the models from becoming obsolete in case of UI changes, they should only include business logic actions, i.e. *actions for navigating between UI views should not lead to a new state* in the behavior model, and be separated in *scripts* making the abstract actions in the model executable.

3. MODEL INFERENCE

ProCrawl highly automatically infers behavior models with appropriate state abstraction covering multi-user process scenarios. The process facilitated by ProCrawl is detailed as follows. First, almost completely automatically, ProCrawl generates a behavior model, by systematically crawling the SUT. This technique is described in Section 3.1. The quality of the inferred behavior models strongly depends on the applied abstraction mechanisms, which are described in Section 3.2. Second, a test engineer may complement the inferred model with additional properties. Based on these models and the UI scripts generated by ProCrawl, acceptance tests can be derived and automatically executed (cf. Section 7), to check whether scenarios across multiple users with different roles can be reproduced on a modified version of the SUT and lead to the expected result.

3.1 Experimental Behavior Model Mining

ProCrawl is an *experimental program analysis* tool, i.e. it "generates findings from multiple executions of the program, where the executions are controlled by the tool" [23]. To handle business processes involving multiple interacting roles, ProCrawl simulates multiple users (*actors*) operating the SUT via a web browser. Actors are configured with a login script to initialize a user session. During the crawling procedure, ProCrawl uses a separate web browser[4] for each actor. Figure 3 illustrates an example run of ProCrawl on the conference management system introduced in Section 1.

a) ProCrawl is configured with
   - a set of *actors* operating the SUT via its web UI,
   - a *mining scope* for each actor, i.e. a set of scripts (usually simply URLs) for navigating to the UI views to be considered for state abstraction,
   - a UI script for triggering the *start event* (here: *submit*),
   - optionally, a selector for UI elements to be considered for state abstraction (cf. Section 3.2), and
   - a command for resetting the state of the SUT.

b) ProCrawl runs the start script and determines the state of the SUT by abstracting over the active UI elements.

c) An initial model with the observed behavior is generated.

d) The model is refined in an iterative process of state observation, modeling and testing: *Test*: ProCrawl generates executable scripts from the behavior model and executes them on the SUT to explore as yet unobserved behavior. *Observe*: ProCrawl determines the state of the SUT. *Model*: The model is refined based on the observations.

e) The enriched model is stored as a *finite state automaton* (cf. Figure 2), i.e. a directed multigraph with properties annotated to nodes and edges, and can be directly used as a specification for model-based testing (cf. Section 7). *Nodes* in the graph represent the states of the SUT as observed by ProCrawl through the SUT's web UI; *edges* represent a sequence of UI events triggered by ProCrawl with one of the configured actors.

3.2 State and Event Abstraction

As discussed in Section 2, enterprise web applications separate functionality in different *views*, which can be accessed by triggering *navigational events* on the UI. To extract a model describing the business logic of the SUT that is exposed via the UI, ProCrawl abstracts over the active UI elements to identify similar states and excludes navigational events from the behavior model. Without abstraction every single change in the UI presentation would lead to a distinct state.

State Abstraction. The idea behind the state abstraction mechanism applied by ProCrawl is that the current state of the SUT is reflected by the UI, where each actor/view relation exposes a specific part of the application state. Therefore ProCrawl distinguishes different states of the SUT using an abstraction function over the state of the *Document Object Models* (DOMs) of these actor/view relations. This allows ProCrawl to determine the application state without requiring access to the SUT's source code. The set of views (mining scope) is configured for each actor. ProCrawl automatically navigates to the respective views with all actors in parallel, extracts the DOMs and computes the state by applying filters on this set of DOMs. In its default configuration ProCrawl abstracts over all interactive HTML elements (visible *hyperlinks* and *buttons*) extracted from the DOMs. However, the type of HTML elements to be considered for state abstraction can be configured by providing XPath expressions such as `//*[@class='ActionButton']` or selectors for text with certain keywords such as `Text('status:*')`.

Event Abstraction. ProCrawl classifies events that do not change the observed state of the SUT (self-loops in the FSA) as navigational. Navigational events are specific to the SUT's UI and therefore volatile throughout the application lifecycle. Furthermore, the usually high number of navigational events clutters the behavior model, masks the effective business logic, and makes the models likely to break in case of UI changes. However, since process scenarios typically span several views, ProCrawl may need to execute navigational events in order to activate an HTML element triggering a functional event. So the derivation of a single step in the process scenario may require the execution of (several) navigational events plus one functional event. ProCrawl addresses this issue by introducing two layers of abstraction: a *model layer* which is independent of navigational events, and automatically executable *UI scripts* in Java as a second layer, binding navigational to functional events. In this way, if the UI of the SUT changes, it is sufficient to adapt the navigational events in the generated scripts only; the model itself remains untouched.

3.3 Steering the Crawling Procedure

By computing the diff between the set of active UI elements before and after triggering an event, ProCrawl is able to determine the causal dependencies between events, such as triggering the *Make Submission* event on the *Submission* view of the *author* actor activates the *Assign* event on the *Assign Reviewers* view for the *conference chair*. ProCrawl only triggers events that have a causal relationship to the start event defined in the configuration. Consequently the behavior models only contain events that are part of a *causal chain* as it is common in business processes; resulting in *modularized* behavior models that are easier to understand and maintain.

4. IMPLEMENTATION

ProCrawl is implemented in Java SE 7, using Selenium[5] for web browser automation. The tool provides an extension mechanism based on the observer pattern; observers can register to certain events, such as before or after clicking an

[4]The number and type of browsers can be configured.

[5]http://seleniumhq.org

HTML element. Test oracles [15] that shall be called during the crawling procedure are implemented as observers.

Algorithm 1 shows the initialization of the crawling procedure for inferring the behavior model. First, the initial state s_0 of the SUT is retrieved (Line 1). As described before, a state has a set of *pending events* which are triggered by ProCrawl through the SUT's UI in that state; for the initial state s_0 the set of pending events is set to the start event defined in the crawl configuration (Line 2). Furthermore, we maintain a set of *pending states* S_p, which have still to be explored; this set is initialized with s_0 (Line 3). An empty FSA is created with s_0 as initial state (Lines 4). Finally, the crawling procedure is called with s_0 and the FSA (Line 5), which is then iteratively built up over the set of states.

Algorithm 1: INIT

```
   Data: set of actors A, set of pending states S_p, config
   Output: FSA
1  s_0 ← retrieveState();
2  s_0.pending ← {config.startEvent};
3  S_p ← {s_0};
4  FSA ← new FiniteStateAutomaton(s_0);
5  return crawl(s_0, FSA);
```

The state of the SUT is retrieved as shown in Algorithm 2. For each actor, the DOMs of the views defined in the actor configuration are retrieved, by navigating to the view and inspecting the resulting HTML code (Line 4). After removing deactivated and invisible HTML elements from the DOM (Line 5), interactive HTML elements are extracted and added to the state's multiset of events (Line 6). Two states are considered as equal, iff the set of distinct elements in the multiset of events is equal. For each distinct event, a UI script is generated (Line 9) that triggers the event by navigating to the respective *view* with the respective *actor* and clicking the HTML element associated with the event.

Algorithm 2: retrieveState

```
   Data: set of actors A
   Output: current state s of the SUT
1  s ← new State();
2  foreach actor ∈ A do
3  |  foreach view ∈ actor.config.scope do
4  |  |  dom ← actor.retrieveDom(view);
5  |  |  d_f ← filter(dom);
6  |  |  s.events ← s.events ∪ events(actor, view, d_f);
7  |  end
8  end
9  generateScripts(s.events);
10 return s;
```

Algorithm 3 shows the crawl procedure recursively building up the behavior model. In each recursion the current state of the SUT s_0 is checked for pending events. If the set is not empty, an event is removed (Lines 2–3) and the UI script associated with the event is executed (Line 4). After the script execution, the current state s_1 of the SUT is retrieved and the set of pending events, i.e. the events activated by the executed script, is computed as the relative complement of the multiset of events in s_1 with respect to the events in s_0 (Lines 5–6). If not already present, the current state s_1 is added to the set of pending states S_p and FSA nodes; an edge from s_0 to s_1, labeled with the triggered *event* is added to the set of FSA edges (Lines 7–10). Finally, the crawling procedure is recursively called with s_1 and the updated FSA (Line 11). If the set of pending events is empty, s_0 is removed from S_p (Line 13). In case there are still pending states in S_p, the backtracking procedure (Algorithm 4) is called to reach a pending state s_p and the crawling procedure is called with s_p (Lines 15–16). If the set of pending states is empty, the FSA is returned (Line 18).

Algorithm 3: crawl

```
   Data: set of pending states S_p
   Input: current state s_0 of the SUT, initial FSA
   Output: enriched FSA
1  if s_0.pending ≠ ∅ then
2  |  event ∈ s_0.pending;
3  |  s_0.pending ← s_0.pending \ {event};
4  |  executeScript(event);
5  |  s_1 ← retrieveState();
6  |  s_1.pending ← s_1.events \ s_0.events;
7  |  S_p ← S_p ∪ {s_1};
8  |  edge ← new Edge(s_0, event, s_1);
9  |  FSA.nodes ← FSA.nodes ∪ {s_1};
10 |  FSA.edges ← FSA.edges ∪ {edge};
11 |  FSA ← crawl(s_1, FSA);
12 else
13 |  S_p ← S_p \ {s_0};
14 |  if S_p ≠ ∅ then
15 |  |  s_p ← backtrack(s_0, S_p);
16 |  |  FSA ← crawl(s_p, FSA);
17 |  else
18 |  |  return FSA;
19 |  end
20 end
```

Algorithm 4 shows the backtracking procedure, which computes the shortest path from the current state s_0 of the SUT to a pending state s_p using Dijkstra's algorithm (Line 2) and executes the UI scripts to reach s_p (Line 5). If the FSA does not contain a path from s_0 to a state in S_p, the SUT is set to the initial state by executing the command provided in the configuration (Line 10), s_0 is set to the initial state, and the backtracking procedure is called again (Lines 11–12).

5. EVALUATION METHODOLOGY

To assess the effectiveness of ProCrawl in inferring behavior models, we conducted three case studies on real-world web applications: an SAP enterprise web application, an open-source web shop and a peer-review system. We use ProCrawl to infer behavior models for the core processes and evaluate them in terms of three sets of measures:

Accuracy. What is the fraction of relevant events, i.e. events related to the target process, in the inferred model (*precision*)? What is the fraction of relevant events covered by the model (*recall*)? What is the fraction of events correctly classified as relevant or not relevant (accuracy)? To obtain these measures, we manually compare the inferred models with the application and available documentation (Section 6).

Algorithm 4: BACKTRACK

Data: FSA, $config$
Input: current state s_0 of the SUT, pending states S_p
Output: pending state s_p

```
1  foreach p ∈ S_p do
2      edges ← FSA.shortestPath(s_0, p);
3      if edges ≠ ∅ then
4          foreach e ∈ edges do
5              executeScript(e.event);
6          end
7          return retrieveState();
8      end
9  end
10 executeCommand(config.initSut);
11 s_0 ← FSA.initialState;
12 return backtrack(s_0, S_p);
```

Correctness. Can the event sequences described in the model be reproduced on the application? For this purpose, we automatically generate test cases from the model and execute the tests on the application, using the UI scripts generated by ProCrawl (Section 7.1).

Suitability for regression testing. Are the inferred models useful for regression testing (Section 7.2)?

Another criterion is the runtime; ProCrawl should pass the "overnight challenge" [9] of completing within 12-16 hours.

5.1 Evaluation Subjects

SAP Web Application (S1). In our first case study, we applied ProCrawl on an SAP enterprise web application for exchanging and processing product compliance information (cf. Figure 4). The main business processes supported by the application include the connection with suppliers and customers, and the exchange of product and component declarations. The user interface provides a navigation menu for accessing several views: *inbox* for notifications and messages, *network* for connecting with suppliers and customers, *components* for managing component declarations and requests, *products* for product declarations, *suppliers* and *customers* for processing connection requests. The server side of the application comprises about 10.5K lines of Java code and the web interface about 48K lines of Java/JSP and 31K lines of JavaScript, using the SAPUI5 SDK[6], an OpenAjax compliant library for building rich internet applications.

OXID eShop (S2). The second case study was performed on *OXID eShop 4.7.3 Community Edition*[7], an open-source e-commerce platform with about 245K lines of code, including 233K lines of PHP, 5.6K lines of JavaScript using the jQuery[8] library and 6K lines of CSS code. The web UI consists of a front-end interface for ordering products, and a back-end interface for shop administration and order processing. A noteworthy characteristic of the back end is the heavy use of *framing*, i.e. the UI consists of multiple web pages displayed simultaneously within the same browser window.

[6]https://sapui5.netweaver.ondemand.com/sdk
[7]http://www.oxid-esales.com/en
[8]http://jquery.com

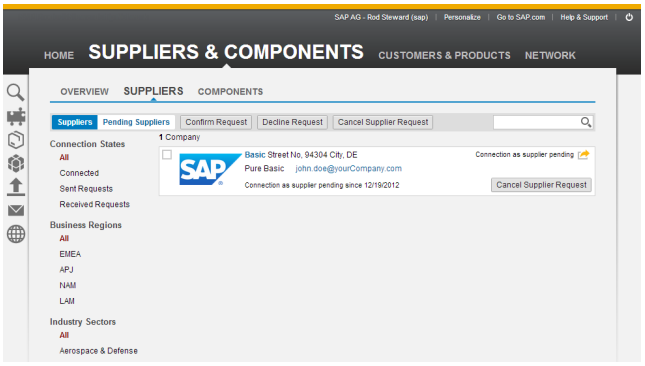

Figure 4: SAP web application for exchanging and managing product compliance information.

OpenConf (S3). In the third case study, we applied ProCrawl on OpenConf *5.10 Community Edition*, an open-source peer-review and conference management system (cf. Figure 1). OpenConf comprises about 14K lines of PHP code and 360 lines of JavaScript code.

5.2 Evaluation Environment

The evaluation has been done on an Intel Core 2 Duo 2.5 GHz with 4 GB RAM, running Windows 7 x64 with Google Chrome 24.

6. MODEL ACCURACY

In this section we evaluate *precision*, *recall* and *accuracy* (ACC) of the models inferred by ProCrawl, as described in Section 5 (cf. Table 1). Besides the configuration, no user input was necessary during the crawling procedure.

6.1 Study 1: SAP Web Application

Target Process. The objective of the first experiment was to infer a behavior model for the *connect with supplier* process scenario, which is the first core scenario of the application involving two partners. In the scenario a manufacturer (*customer*) sends a connection *request* to a *supplier*, by clicking the *Add As Supplier* button on the *network* view of the application. The request can either be *confirmed* or *declined* by the *supplier*, by clicking the corresponding button on the *customers* view, or *revoked* by the *customer*, by clicking the *Cancel* button on the *suppliers* view. Each event except for *revoke* results in a *notification* in the *inbox* of the other partner, which can be *deleted* at any time.

Setup. We conducted three runs with different configurations: (C1) *Default* state abstraction considering HTML button elements and links; and including all views of the application into the mining scope. (C2) *Default* state abstraction with a *reduced* set of views in the mining scope. (C3) *Custom* state abstraction only considering buttons with a certain CSS class[9] and a *reduced* mining scope.

For all runs we configured ProCrawl with two *actors* representing a customer and a supplier, provided *login credentials*, a batch file for *resetting* the application database and a *start script*, triggering the connection request event by clicking the *Add As Supplier* button on the *network* view.

Findings. With C1 and C2, ProCrawl inferred the FSA depicted in Figure 5. Reducing the mining scope in C2,

[9]This can simply be done by providing an XPath expression such as `//*[contains(@class, 'ActionButton')]`.

Table 1: Precision, recall and accuracy (ACC) of ProCrawl in terms of target processes covered in the model.

Subject	SLOC	Config	States	Trans.	Events	Scripts	Precision	Recall	ACC	Time
SAP	Java/93K	default	14	31	114	5	1.0 (31/31)	1.0 (31/31)	1.0	42 min
		custom	12	28	53		1.0 (28/28)	1.0 (28/28)		17 min
OXID	PHP/245K	default	6	13	170	9	0.92 (12/13)	0.75 (12/16)	0.97	49 min
		custom		12	61	8	1.0 (12/12)		0.93	18 min
Open-Conf	PHP/15K	default	10	34	279	7	0.82 (28/34)	0.76 (28/37)	0.95	87 min
		custom			102				0.86	36 min

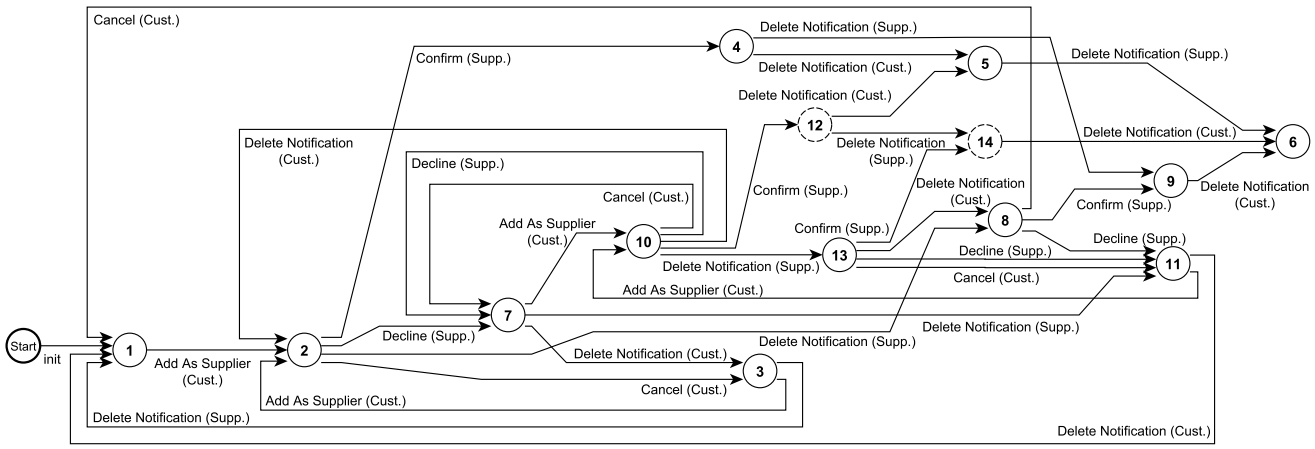

Figure 5: The ProCrawl behavior model of the connection process in the SAP web application.

reduced the runtime from 64 to 42 minutes and limiting the set of HTML elements considered for state abstraction in C3 further reduced the runtime to 17 minutes. The FSA inferred with C1 and C2 consists of 14 states and 31 transitions, where self-loops, i.e. transitions that do not change the state, are excluded from the model. The FSA precisely covers the process description, modeling the interaction behavior between a customer (Cust.) and a supplier (Supp.) in the connection process. All transitions represent events relevant for the target process and all events of the target process are covered by the model, resulting in a *precision* and *recall* of 1.0. ProCrawl generated 5 UI scripts, removing navigational events from the model and triggered 114 compound events, consisting of multiple UI events.

With the custom config C3, ProCrawl merged state 12 with state 4 and state 14 with state 9, resulting in 12 states and 28 transitions. By manually comparing the states with the application, we figured out that the two additional states in the default configuration represent different message types in the *inbox* view of the application. In state 12 and 14 the customer's *inbox* contains a *declined* and a *confirmed* notification. However, in the custom configuration C3, hyperlinks, which determined the message type, were not considered for state abstraction.

6.2 Study 2: OXID eShop

Target Process. The objective of the second case study was to infer a behavior model for the OXID *ordering process* involving a *customer* and a *retailer*. The customer interacts via the shop front end, whereas the retailer operates via a separate back-end interface:

1. The *customer* adds a product to the shopping cart by selecting an article in the *product* view and clicking the *To Cart* button.

2. The *customer* selects a payment method in the *shopping cart* and clicks the *Continue to Next Step* button.

3. The *customer* orders the product by clicking the *Order now* button in the *shopping cart* view.

4. The *retailer* sets the status of the order to *shipped* by clicking the *Ship Now* button in the *orders* view.

Between Step 1 and 2, the *customer* may submit a coupon by entering a valid code and clicking the *Submit Coupon* button. Between Steps 1 and 3, the *customer* may remove articles by selecting them in the *shopping cart* view and clicking the *remove* button. After Step 3, the *retailer* may

- *cancel* the order, by clicking a link in the *orders* view and confirming the popup; this changes the status of the order and prevents shipment (Step 4).[10]

- *delete* the order by clicking a link in the *orders* view and confirming the popup; this deletes the order from all views in the system, disabling all events related to that order.

- generate an *invoice* by clicking the *Create PDF* button in the *orders* view.

[10]However, the *Ship Now* button is not deactivated.

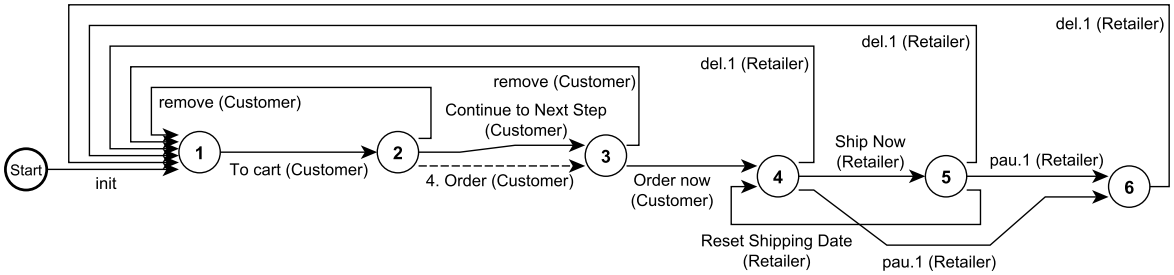

Figure 6: The ProCrawl behavior model of the ordering process in OXID eShop (del.1 = delete; pau.1 = cancel).

After Step 4 the *retailer* may reset the order status by clicking the *Reset Shipping Date* button in the *orders* view.

Setup. We conducted two runs: (C1) *Default* state abstraction. (C2) *Custom* state abstraction considering buttons and links with specific CSS classes. For both runs we configured ProCrawl with two *actors* representing a customer and a retailer, provided login credentials and a start script adding a product to the *shopping cart*. The *mining scope* was set by providing UI scripts to access the *shopping cart* and *order history* of the front end, as well as the *orders* view of the back end. Since deleting an order via the back end resets the process, ProCrawl did not require a command for resetting the application database.

Findings. With the default configuration ProCrawl inferred the FSA depicted in Figure 6. Reducing the number of HTML elements considered for state abstraction in C2, reduced the runtime from 49 to 18 minutes, as well as the number of triggered compound events from 170 to 61. The FSA inferred with C1 consists of 6 states and 13 transitions. The FSA inferred with C2 is equal, but does not contain the dashed transition from state 2 to state 3, which represents a link with the same effect as the *Continue to Next Step* button. For each distinct event in the model, ProCrawl generated a UI script, i.e. 9 in C1 and 8 in C2.

In the FSA inferred with C1, all transitions except for the dashed one represent relevant events, resulting in a *precision* of 0.92. With C2, all of the 12 transitions are relevant. In both configurations, ProCrawl did not include the *Submit Coupon*, as well as the *Create PDF* event. Although both events were triggered during the crawling procedure, they did not change any elements considered for state abstraction; both models cover 12 out of 16 relevant events (8 out of 10 distinct events), resulting in a *recall* of 0.75. With 154 (170-16) true negatives in C1 and 45 (61-16) in C2, the *accuracy* is 0.97 and 0.93 respectively.

6.3 Study 3: OpenConf

Target Process. Objective of the third case study was to infer a behavior model for the OpenConf *peer-review process* involving an *author*, a *reviewer* and a conference *chair*:

1. The *author* makes a submission by filling out the submission form and clicking the *Make Submission* button.
2. The *chair* assigns reviewers for the submission by navigating to the *Auto Assign Reviewers* view and clicking the *Make Assignments* button.
3. The *reviewer* submits a review by filling out the review form and clicking the *Submit Review* button.
4. The *chair* changes the status of the submission (*pending*, *accepted*, *rejected*) by clicking the corresponding button in the *Submission Scores* view. Note that the chair does not have to wait for the reviews.

After Step 1, the author may withdraw the submission at any time by providing the submission id and the password from the submission form, clicking the *Withdraw Submission* button and confirming the pop-up dialog.

Setup. We conducted two runs: (C1) *Default* state abstraction. (C2) *Custom* state abstraction considering buttons of type `submit` and links with specific attributes. For both runs we configured three actors: a *chair*, a *reviewer* and an *author*, provided login credentials where necessary and a start script for making the submission. The *mining scope* was defined by providing UI scripts to access the views mentioned in the scenario description. Furthermore we provided a batch file for resetting the application database.

Findings. ProCrawl inferred the FSA depicted in Figure 2 with both configurations, generating 7 UI scripts. Reducing the number of HTML elements considered for state abstraction in C2, reduced the runtime from 87 to 36 minutes, as well as the number of triggered compound events from 279 to 102. The FSA consists of 10 states and 34 transitions, as described in Section 1. From the 34 transitions, 6 represent *Go* events having the same effect as *Accept* and therefore being irrelevant for the model, resulting in a *precision* of 0.82 (28/34). However, ProCrawl was not able to successfully trigger the *Withdraw Submission* event[11], which can be triggered in each of the 9 states after making the submission, resulting in a *recall* of 0.76 (28/37). With 242 (279-37) true negatives in C1 and 65 (102-37) in C2, the *accuracy* is 0.95 and 0.86 respectively.

6.4 Summary and Discussion

ProCrawl highly automatically inferred behavior models with high *precision* (≥ 0.82), *recall* (≥ 0.75) and *accuracy* (≥ 0.86). The default state abstraction of ProCrawl, considering buttons and links was effective for all three case studies. However, adapting the state abstraction to the target application, reduced the number of triggered events and consequently the runtime of ProCrawl by 60% on average. The runtime was below 1.5h for all case studies, passing the "overnight challenge" of completing within 12-16 hours; by customizing the state abstraction ProCrawl finished within

[11]To trigger the event, ProCrawl would need to provide the submission id displayed after submitting the paper, as well as the password provided in the submission form.

Table 2: Using ProCrawl behavior models for detecting disruptive process changes (A) and UI changes (B).

Subject	Model	SUT	Test Cases	Avg. Test Case Length	Failure Type A	Failure Type B	Runtime
SAP app	1.0	1.1	6	9.8	0	1	10:23 min
	1.0	0.9			21	0	9:10 min
OXID eShop	4.6.5	4.7.3	1	26	0	1	3:57 min
	4.5.12	4.6.5			0	0	6:09 min

36 minutes, even meeting the "lunch challenge". A major part of the overall runtime depends on the responsiveness of the SUT; over the 7 runs, $62.6\% \pm 6.3\%$ of the time has been spent waiting for the web browser to load the application.

7. TEST CASE GENERATION

The state automata inferred by ProCrawl serve as *gold standard oracles* [3], checking whether behavior observed on a trusted system can be reproduced on another version of that system. Failing test cases can be caused by

- disruptive changes in the process implementation, i.e. process steps have been removed or mandatory steps have been added. (Type A)
- changes on the UI, resulting in obsolete UI scripts. UI changes appear frequently, especially in an early stage of development. Therefore the manual effort to fix the test cases should be minimal. (Type B)

Setup. Nodes and edges in the models are annotated with *properties* used for testing, such as the URL to the SUT, and references to the generated UI scripts. Optionally, the user can refine the model by adding additional properties such as requirement tags, enabling requirement coverage in test generation, define the probability of following an edge or adding *guards* and *actions* to edges in order to handle complex scenarios. In addition, user-defined oracles and handlers can be added. However, for the case study we only annotated the URL to the SUT and a reference to a batch file for initializing the SUT. For test generation we used GraphWalker[12], with the A* algorithm for path generation and 100% edge coverage as stop condition.

7.1 Model Correctness

To evaluate the *correctness* of the inferred models, we use them as a specification for GraphWalker and check if the behavior captured in the models can be reproduced on the source application by executing the test cases generated by GraphWalker via the UI scripts generated by ProCrawl.

SAP Web Application. GraphWalker generated 6 test cases with 9.8 events on average, covering 100% of the transitions. All of the test cases could be executed via the UI scripts generated by ProCrawl; none of the test cases failed.

OXID eShop. GraphWalker generated 1 test case with 26 events covering 100% of the transitions. There was no error during the execution.

OpenConf. GraphWalker generated 3 test cases with 14.3 events on average, covering 100% of the transitions. There was no error during test execution.

[12]http://graphwalker.org

7.2 Regression Testing

To evaluate the suitability of the models for regression testing, we inferred models from different versions of the SAP application and the OXID eShop and executed the generated test cases on another version[13] (cf. Table 2). The main objective of regression testing is to detect disruptive process changes (Type A failure), while non-disruptive extensions are tolerated. Testing on the UI level naturally also detects UI changes (Type B failure). In that case, the effort for fixing broken test cases should be minimal.

SAP Web Application. We used the behavior model ProCrawl inferred from v1.0 of the SAP web application to generate test cases with GraphWalker, covering 100% of the transitions. We executed the 6 test cases on v1.1 and v0.9 of the application. On v1.1 one of the test cases failed due to a changed button label. This could simply be fixed by adapting one of the generated UI scripts. Although the connection process has been extended in v1.1, all of the test cases passed after adapting the UI script; indicating that the process extension was done in a non-disruptive way. To simulate a disruptive process change, we executed the tests on v0.9 of the application, where the notification procedure was not yet implemented. During the execution of the 6 test cases, 21 errors were reported; in all of them the execution of the `DeleteNotification` script failed, because the HTML element was not found on the UI.

OXID eShop. We conducted two runs, using ProCrawl to infer a behavior model and generating UI scripts for OXID v4.5 and v4.6. From the inferred models, we generated test cases covering 100% of the transitions and executed them on the subsequent version of the system.

From v4.5 to the latest version 4.7.3 there was no change in the ordering process, so the inferred models are essentially identical and GraphWalker generated 1 test case with 26 events to cover all transitions. However, in version 4.7 the label of the ordering button was changed from *Purchase* to *Order now*, resulting in a failed test case. Therefore we had to adapt the label in one of the generated UI scripts.

7.3 Summary and Discussion

All of the test cases generated from the behavior models inferred by ProCrawl could be executed on the source application without errors, i.e. the behavior captured in the FSA correctly simulates the respective parts of the SUT.

Using the models for regression testing, we detect changes on the UI, such as changed button labels, as well as disruptive process changes. In the former case, the test cases could simply be fixed by adapting the affected UI scripts, whereas the model was unaffected.

[13]For OpenConf, only the latest version was publicly available.

8. THREATS TO VALIDITY

The evaluation of ProCrawl is subject to several threats to validity. First and foremost comes *external validity*—the ability to generalize from our findings. We have evaluated ProCrawl on three diverse, nontrivial web applications, and found that the resulting models represent the underlying processes in a reasonably accurate manner. We have chosen web applications that implement a multi-user process, provide an HTMLUI and are to be set up in reasonable time, and selected the applications' respective core processes as target. However, the assumption underlying ProCrawl is that most behavior can be explored through simple interactions. If the behavior depends on specific features of the supplied data, for instance, then a black-box technique like ProCrawl is unlikely to explore this through guessing. This is a limitation affecting all kinds of automated test generators, of course, and ProCrawl is no exception. In the current implementation input data can either be provided via the configuration or is randomly generated by ProCrawl until it is accepted by the SUT, i.e. no error message is shown on the SUT's UI. However, processes may vary based on the provided input data; addressing this issue is left to future work.

For applications that are similar to the ones we studied, including web shops, submission systems and the like, we are confident that the ProCrawl results will be similarly accurate as in our three case studies.

The second important threat in our study is *internal validity*—our ability to draw conclusions about the connections between our independent and dependent variables. As we check precision and recall against scenarios that we extracted from the applications and available documentation, there is an obvious risk of researcher bias—that is, we could have selected interactions that ProCrawl can find, and left out those which ProCrawl cannot find. We counter this threat by including open-source systems in our evaluation set, such that readers can compare the target processes and models described in this paper with the applications themselves.

The final concern in our study is *construct validity*—the adequacy of our measures for capturing dependent variables. As it comes to extracting models from systems, comparing them against ground truth established through static analysis or testing is common practice; and thus we believe in the adequacy of our accuracy measures.

In this paper, we have not researched the question of whether the resulting models are readable or understandable by humans; however, we found that the resulting models, as reported in this paper, strive a nice balance between accuracy (reflecting as much behavior as possible) and abstraction (allowing for simple understanding), which gave us important insights into hitherto unknown systems.

9. RELATED WORK

Approaches for automatically inferring properties of software components [16] can be classified in *static* and *dynamic* techniques. Static techniques analyze source code and are usually sound, but have limited scalability. Dynamic approaches infer properties from multiple program runs, making them applicable for real-world applications where source code may not be available for each component. However, the quality of the results heavily depends on the choice of runs. There exist numerous tools that infer various types of properties by abstracting over a given set of program runs; a prominent representative in the area of *process mining* is the ProM[14] framework for analyzing process event logs. Various other approaches build upon the work of Biermann and Feldman [2]. However, the fact that such tools rely on possibly incomplete logs and cannot generate additional program runs, poses the problem of under- versus overfitting the given set of program runs [19].

Experimental approaches [23] such as ProCrawl handle this problem by systematically generating further program runs. The idea of generating and executing test cases to infer a more complete model was first published by Xie and Notkin [21], who implemented the approach in the Obstra [22] tool. Dallmeier et al. implemented this idea for Java classes in TAUTOKO [6]. Whereas the aforementioned tools infer models describing the possible behavior of a class of Java objects using code instrumentation, ProCrawl infers behavior models of processes involving multiple software components operated by users acting in different roles. ProCrawl does not modify the application code, but leverages the UI to observe the behavior and generate test cases simulating possible user input and therefore being realistic and relevant by construction [12]. In this matter ProCrawl builds on former experiences with our previous approach [17] and web crawling tools like Crawljax [13, 14]; on top it abstracts from UI specifics with a state abstraction specially designed to capture multi-user business processes and produces models which are accurate enough for model-based testing. In its default configuration the state abstraction applied by ProCrawl is similar to *enabledness preserving abstractions* proposed by de Caso et al. [7] in the context of validating pre-post condition based specifications. However, ProCrawl considers different users performing the operations and allows to include displayed text in the state abstraction function.

On the *service test* layer, StrawBerry [1] synthesizes an automaton modeling the behavior protocol of a web-service (WS) from the WSDL description and refines the automaton through testing against the WS implementation; while ProCrawl's focus is to test and therefore model the SUT's behavior on the *system test* layer, i.e. from an end-user perspective.

10. CONCLUSION

ProCrawl is a fully automatic tool to mine behavior models from enterprise web applications for system testing, understanding, and maintenance. Its state abstraction is specific enough to capture essential process steps, including cycles; yet generic enough to be applicable to a diverse range of web applications. Requiring little customization, ProCrawl can be easily deployed on new applications. As our evaluation shows, the models inferred by ProCrawl are small, accurate, and cover all to almost all events. In future work, we will improve our approach to infer process variances based on the provided input data and investigate further use cases, like program comprehension.

11. ACKNOWLEDGMENTS

We thank Clemens Hammacher, Jeremias Rößler, Sebastian Wieczorek and the anonymous reviewers for useful comments on earlier revisions of this paper. The work presented herein is partially funded by the German Federal Ministry of Education and Research (BMBF) under grant no. 01IC12S01.

[14] http://www.processmining.org/prom

12. REFERENCES

[1] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic Synthesis of Behavior Protocols for Composable Web-Services. In *ESEC/FSE '09*, pages 141–150, New York, NY, USA, 2009. ACM.

[2] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, C-21(6):592–597, 1972.

[3] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison Wesley, 1999.

[4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00*, pages 439–448, New York, NY, USA, 2000. ACM.

[5] V. Dallmeier, M. Burger, T. Orth, and A. Zeller. Webmate: a tool for testing web 2.0 applications. In *Proceedings of the Workshop on JavaScript Tools*, JSTools '12, pages 11–15, New York, NY, USA, 2012. ACM.

[6] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *ISSTA '10 Proceedings of the 19th international symposium on Software testing and analysis*, pages 85–96, New York, NY, USA, 2010. ACM.

[7] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Automated abstractions for contract validation. *IEEE Transactions on Software Engineering*, 38(1):141–162, 2012.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[9] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 500–517, London, UK, 2001. Springer-Verlag.

[10] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[11] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification & Reliability*, 21(1):55–71, Mar. 2011.

[12] F. Gross, G. Fraser, and A. Zeller. Exsyst: Search-based gui testing. In *ICSE '12*, pages 1423–1426, Piscataway, NJ, USA, 2012. IEEE Press.

[13] A. Mesbah, A. Van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1):1–30, Mar. 2012.

[14] A. Mesbah, A. Van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering*, 38(1):35–53, Jan. 2012.

[15] M. Pezze and M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2007.

[16] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated api property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, May 2013.

[17] M. Schur. Experimental specification mining for enterprise applications. In *ESEC/FSE '11*, pages 388–391, New York, NY, USA, 2011. ACM.

[18] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[19] W. M. P. van der Aalst, V. Rubin, H. M. W. Verbeek, B. F. van Dongen, E. Kindler, and C. W. Günther. Process mining: a two-step approach to balance between underfitting and overfitting. *Software and System Modeling*, 9(1):87–111, 2010.

[20] S. Wieczorek and A. Stefanescu. Improving testing of enterprise systems by model-based testing on graphical user interfaces. In *ECBS '10 Proceedings of the 2010 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 352–357, Washington, DC, USA, 2010. IEEE Computer Society.

[21] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *International Workshop on Formal Approaches to Testing of Software (FATES)*, pages 60–69. Springer, 2003.

[22] T. Xie and D. Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *International Conference on Formal Engineering Methods (ICFEM)*, pages 290–305. Springer, 2004.

[23] A. Zeller. Program analysis: A hierarchy. In *ICSE Workshop on Dynamic Analysis (WODA 2003)*, pages 6–9, 2003.