Mining Workflow Models from Web Applications

Matthias Schur, Andreas Roth, and Andreas Zeller, *Member, IEEE*

Abstract—Modern business applications predominantly rely on web technology, enabling software vendors to efficiently provide them as a service, removing some of the complexity of the traditional release and update process. While this facilitates shorter, more efficient and frequent release cycles, it requires continuous testing. Having insight into application behavior through explicit models can largely support development, testing and maintenance. Model-based testing allows efficient test creation based on a description of the states the application can be in and the transitions between these states. As specifying behavior models that are precise enough to be executable by a test automation tool is a hard task, an alternative is to extract them from running applications. However, mining such models is a challenge, in particular because one needs to know when two states are equivalent, as well as how to reach that state. We present Process Crawler (ProCrawl), a tool to mine behavior models from web applications that support multi-user workflows. ProCrawl incrementally learns a model by generating program runs and observing the application behavior through the user interface. In our evaluation on several real-world web applications, ProCrawl extracted models that concisely describe the implemented workflows and can be directly used for model-based testing.

Index Terms—Specification mining, dynamic analysis, model-based testing, web system testing

1 Introduction

The ubiquity of web browsers and advancements and standardization in web technologies facilitate the development of cross-platform desktop-like web applications and have made web browsers a dominant client for enterprise software. Moreover, sufficiently high and reliable network bandwidth allows applications to be operated at the vendor side and provided as services to customers, which removes some of the complexity and costs of the traditional software release and update process. While this enables shorter, more efficient and frequent release cycles, it puts high pressure on software development and requires paying close attention to operational aspects, continuous quality assurance (QA) and testing.

Having insight into the behavior of a software component through explicit *models* can largely improve the development, QA and maintenance process. Behavior models can support system understanding and validation. *Model-based testing* [37] (MBT) allows for efficient test creation when a model describing the possible and the expected application behavior is available, and thus for increased automation. However, manually creating behavior models that are precise enough to be executable by a test automation tool and maintaining them throughout the software development process is expensive. Web applications typically come without explicit test models, which implies mostly manual and thus less efficient test creation, and also slows down understanding and maintenance.

- *M. Schur is with SAP SE, and Saarland University-Chair for Software Engineering, Germany. E-mail: matthias.schur@sap.com.*
- *A. Roth is with SAP SE, Germany. E-mail: andreas.roth@sap.com.*
- *A. Zeller is with the Saarland University–Chair for Software Engineering, Germany. E-mail: zeller@cs.uni-saarland.de.*

Manuscript received 7 Nov. 2014; revised 25 May 2015; accepted 31 May 2015. Date of publication 27 July 2015; date of current version 11 Dec. 2015. Recommended for acceptance by M. Pezzè.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2461542

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 and test engineers can apply them in a continuous integration environment to check for regressions after code changes. Specification mining has been used to successfully derive axiomatic specifications such as function and data invariants from programs [15], or finite state machines describing states and transitions for individual classes [11], [13]. 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. Extracting models on system level 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 front end that allows for human interaction.

In this paper, we present ProCrawl, a fully automatic tool that mines behavior models from *workflow web applications*. Such applications support the execution of (usually business-related) processes, which typically involve multiple interacting roles, such as a seller and a vendor in an ordering process; or an author, a reviewer, and a program chair in a peer-review process. They are centered around data stored in a shared data store that is manipulated when executing the processes. The data is collaboratively edited through browser-based clients. Depending on the user role these clients may have varying functionality that is typically separated over multiple UI views.

After presenting the results of a comparative experiment on a popular peer-review application (Section 2), we make the following *contributions*:

1) To the best of our knowledge, ProCrawl is the first approach automatically learning multi-user workflow

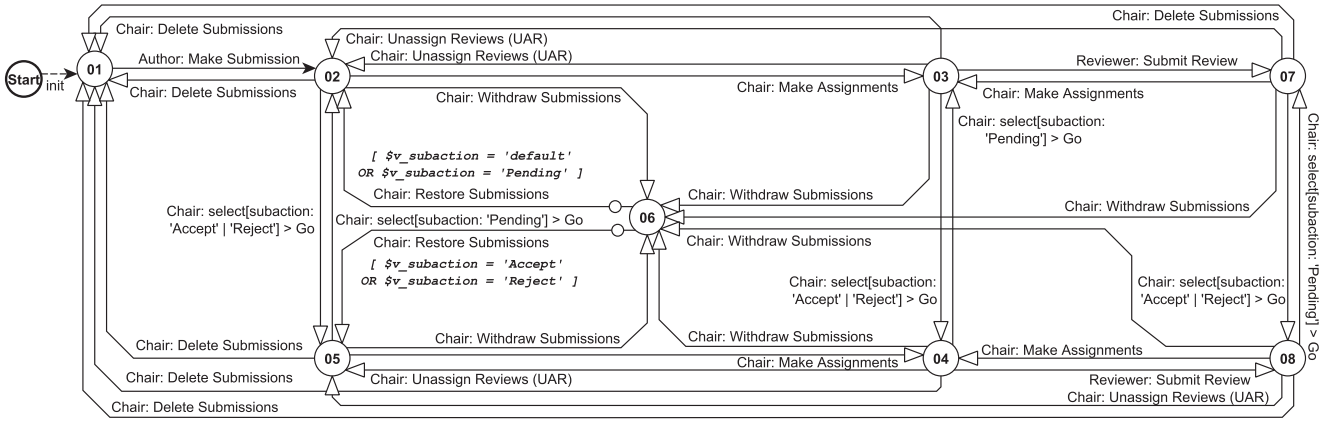

Fig. 1. **The ProCrawl workflow model for OpenConf** , detailing the peer-review process involving an author, a reviewer, and a chair. Transitions are labeled with the actions causing the state change, as well as a transition guard that defines under which conditions the transition is enabled. For instance, transition $6 \rightarrow 2$ is enabled if subaction $=$ default $\vee$ subaction $=$ 'Pending' holds; if subaction $=$ 'Reject' $\vee$subaction $=$ 'Accept', transition $6 \rightarrow 5$ is enabled instead. The model is unaltered ProCrawl output; only the graph layout was manually adjusted and some of the labels have been shortened.

models that are complete enough to be executable by a UI automation tool. Section 3 discusses the approach, and Section 4 gives details on the implementation.

2) We present a machine learning approach for inferring decision rules from data input that serve as transition guard conditions for resolving nondeterminism (Section 3.7).
3) In our evaluation on several real-world web applications, the workflow models mined by ProCrawl concisely capture the essentials of the underlying process. Section 5.1 introduces the case study design and Section 5.2 reports the results.
4) The resulting workflow models can be directly used as input to *model-based testing*. In Section 5.3, we show how automated tests following the extracted models detect structural and logical changes (LC) in new versions of the applications.

After discussing threats to validity (Section 5.4) and related work (Section 6), Section 7 closes with the conclusions and future work.

This article is a substantially revised and extended version of our original papers [33], [34]; we present a novel approach for learning transition guard conditions, as well as an extended evaluation including two additional case studies.

2 Running Example: OpenConf

Fig. 1 shows a behavior model automatically extracted by ProCrawl that represents the peer-review process in OpenConf,[1] a popular web-based conference management system. The model is an *extended finite state machine* (EFSM) in which the nodes denote abstract individual states of the web application, numbered in the order they were detected, and state transitions represent sequences of UI commands that are performed by users acting in different roles (*actors*).

Using OpenConf, authors can submit papers, which are then reviewed by peers and finally accepted or rejected by the chair; a process we assume is familiar to most readers of this paper.

All ProCrawl requires to infer this model is a login script for a reviewer and the chair, a start action submitting a paper, a test fixture and a scope definition, i.e. the URLs of the OpenConf views that shall be explored; based on this configuration, ProCrawl fully automatically explores the life cycle of a submission:

After the author makes a submission (i.e. the start action), the state of the submitted paper is pending (State 02). The program chair can accept or reject (State 05) before assigning a reviewer. For each of the two unassigned states there is a corresponding state after the assignment of a reviewer (State 03, 04) and after the reviewer submitted the review (State 07, 08). At all times, the chair can revoke an earlier decision or assign a new reviewer. Furthermore the chair can withdraw a paper (State 06) or restore a withdrawn paper to its previous status: pending (State 02) which is the default, or accepted/rejected (State 05).

To extract such a model, ProCrawl systematically generates and executes sequences of UI commands for the configured actors and observes changes on the configured set of views (*exploration scope*). From its observations, ProCrawl infers state abstractions allowing it to determine whether an observed state is considered equivalent to a known one (which implies a cycle in the model) or whether it is not (in which case it generates additional program runs to explore the yet unobserved behavior). This state abstraction and the support for multiple actors and views is crucial for the effectiveness of ProCrawl: While efficient web crawling tools have been presented before, they do not make their underlying model explicit [12], or detect inappropriate process states.

Table 1 shows the results of an experiment running Crawljax[2] [26], a state-of-the-art web crawling tool on OpenConf. Crawljax has been published in 2008 for crawling modern web applications that could not be crawled by existing web crawlers and is actively maintained and extended since then. Given a URL and an optional configuration, Crawljax automatically explores a web application and creates a state-flow graph (SFG) of the UI states and

1. http://www.openconf.com

2. http://crawljax.com (version 3.6)

TABLE 1
CRAWLJAX [26] on OPENCONF

Crawljax	Depth	States	Trans.	Time (m)	Coverage
min.	2	12	15	1	9.1%
max.	20	481	1390	141	36.4%
best	**15**	**66**	**177**	**13**	**36.4%**
ProCrawl	**2**	**8**	**31**	**58**	**90.9%**

Results over 13 runs with click depth 2-20, 19 data-input & 18-35 crawl rules.

the event-based transitions between them. These models can be used for automatically testing web applications [27]. CRAWLJAX can be extensively configured using a Java API. We performed 13 runs with a maximum click depth between 2 and 20; 19 data-input rules including the user credentials and form data for the paper submission; 18-35 additional crawl rules specifying elements not to click, such as "Configuration" and "forgot password?"; enabled and disabled random input data generation.

CRAWLJAX finished within 1-141 minutes and produced models with 12-481 states and 15-1,390 transitions, covering 9.1-36.4 percent of the actions that are relevant for the peer-review workflow. Over all 13 runs the smallest among the models with the highest coverage (36.4 percent) has 66 states and 177 transition. While the inferred models precisely describe the different UI states and transitions caused by single click events, they miss a large part of the underlying workflow steps, such as submitting a review, which are distributed across multiple views and actors. Furthermore, the models are hard to validate due to their large number of states and transitions. This is a result of the missing support for multiple actors and views and the default state abstraction applied by CRAWLJAX, which is affected by minor structural changes (SC) on the UI.

To validate the model, we used it as input for an open-source MBT tool[3] and executed the generated test cases (full transition coverage) on OPENCONF: 27 of 85 (31.8 percent) test cases failed due to nondeterminism in the model and missing causality, e.g., trying to access a paper before it has been submitted.

In contrast, the model extracted by PROCRAWL covers 90.9 percent of the 11 workflow-relevant actions (cf. Section 5.2.1) with only eight states and 31 transitions based on the configuration described in Section 3.1(1).

3 MINING WORKFLOW MODELS

Technically, web applications have a client and a server part. The client code runs in a web browser and is usually implemented in HTML, CSS and JavaScript. The browser parses the HTML code and creates a *Document Object Model*[4] (DOM) tree with the content and structure of the page, representing HTML elements such as hyperlinks, buttons and text as nodes, which are rendered using the style information defined in HTML attributes or CSS. Content, structure and style can be dynamically modified through client-side JavaScript event handlers that are called on DOM events such as `onclick` or server-side state changes that are propagated to the client. To improve the user experience and reduce the amount of data to be transferred from the server to the client in order to update a page, modern web applications usually exchange data with the server asynchronously using JavaScript and the XMLHttpRequest object (AJAX), i.e. the UI can be updated without a full page reload.

Applications typically separate functionality into different *views* $\mathcal{V}$, each view $v \in \mathcal{V}$ consisting of a set of elements $\mathcal{E}_v$. For role-based applications, $\mathcal{V}$ as well as $\mathcal{E}_v$ usually depend on the role of the actor $\alpha \in \mathcal{A}$ interacting with the application. In OPENCONF for instance, program chair and reviewer both can list submissions, but only the chair can see all submissions, access their review score and accept or decline submissions. In web applications each view is represented by a distinct DOM tree that is accessible via a URL followed by an optional sequence of *UI events*, such as $\langle(\text{open}, \mathit{openconf.com/chair}), (\text{click}, \mathit{List\ Submissions})\rangle$. We classify UI events in *navigation events*, which only change the DOM tree (UI state) of the view on which the event was triggered and *system-interaction events* [45] that cause a change in the application state that is observable on one or more views. While view changes caused by system-interaction events remain after a view reload, changes caused by navigation events are volatile. Our assumption is that only system-interaction events are relevant for the underlying workflow.

3.1 Experimental Workflow Exploration

As many other approaches to behavior model inference, PROCRAWL applies *dynamic analysis*, i.e. generalizes upon observed application executions (traces). However, when this set of traces is predefined (e.g., [3], [24]), there is the risk of inferring partial models only reflecting the behavior from the incomplete set of traces. To mitigate this risk, PROCRAWL applies an *experimental analysis* [47] approach, i.e. explicitly controls and generates additional executions instead of relying on a given set of execution traces. Applying a web crawler to create a more extensive set of execution traces has also been done in [4]. However, PROCRAWL applies an online algorithm performing execution (trace) generation and model inference in each iteration, instead of producing all traces before starting the model inference. In this way, PROCRAWL can leverage the behavior model for efficient execution generation.

PROCRAWL applies a black-box approach, i.e. it mines application behavior without accessing the source code of the *system under test* (SUT); instead the state and active actions are determined by observing a configured set of UI views (technically DOM trees).

Given a configuration, PROCRAWL automatically explores the behavior of a web application and generates a behavior model (cf. Fig. 2):

1. *Configuration.* The configuration includes

- the participating *actors* $\mathcal{A}$: in the OPENCONF example an author, a chair and a reviewer, including their login credentials.
- a *start action* a_0 which is executed by one of the actors (Author : Make Submission in OPENCONF).
- an *exploration scope* $\mathcal{V}_\alpha$ for each actor $\alpha \in \mathcal{A}$, i.e. the views of the SUT to be checked for changes after

3. http://graphwalker.org (version 3.3.0)
4. http://www.w3.org/DOM

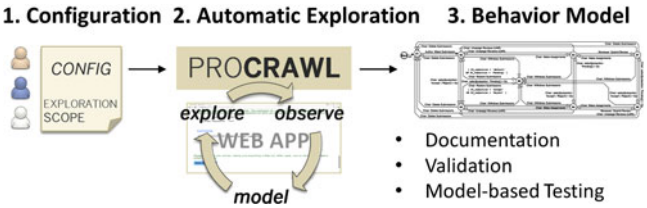

Fig. 2. **ProCrawl overview**. Given a configuration **(1)**, ProCrawl **(2)** automatically explores the behavior of the web application (cf. Fig. 3), and **(3)** generates a behavior model.

executing an action; for each view a sequence of UI commands, such as $\langle(\text{open}, \mathit{openconf.com/chair}), (\text{click}, \mathit{List\ Submissions})\rangle$.

In OpenConf this includes the URLs to the reviewer assignments, as well as the submission, review and score list for the chair, and the submissions list for the reviewer.

- a *test fixture*[5] to set the SUT to the initial state (test context), which may be necessary to complete workflow exploration after a nonreversible action.

Optionally for each actor the type and number of browsers, maximum click depth and DOM elements to be considered or ignored, as well as the input elements for which different values are generated can be specified. Furthermore, the state abstraction can be changed by providing DOM filters and selectors, and test oracles can check the SUT for errors during behavior exploration.

2. *Automatic exploration.* Based on this configuration, ProCrawl automatically explores the behavior of the web application. Fig. 3 illustrates the iterative exploration process for OpenConf; each iteration consists of the following steps (cf. Fig. 4):

(a) *Run action*. An action (α, ς), i.e. a sequence of UI commands ς (*script*, cf. Section 3.2) that is assigned to an actor α, is chosen from a set of pending actions $A(s)$ of the current state s, executed and removed from $A(s)$. In the first iteration, $A(1)$ only contains a configured start action such as (Author, Make Submission) for OpenConf.

(b) *State abstraction*. The current state s of the SUT is determined by applying an abstraction function f over the DOM trees (views) extracted with the configured actors (cf. Section 3.3). If a new state is detected, a set of actions that shall be explored is generated using a script function f_ς (cf. Section 3.2) and added to $A(s)$ (cf. Section 3.5).

(c) *Model*. The observed behavior is modeled as an EFSM (Section 3.4), where each executed action (α, ς) that changed the state of the SUT, becomes a state transition in the model.

The exploration continues until the set of pending actions A is empty, or a configured stop condition holds and results in an incrementally constructed model as the one shown in Fig. 1 for OpenConf.

If an action with varying behavior such as Restore Submissions in state 06 of OpenConf is detected that results in a nondeterministic model, *transition guard learning* (Section 3.7) tries to automatically resolve the nondeterminism by learning decision rules over the provided input data, such as in Fig. 1 (State 06).

5. Typically a script for resetting the database of the SUT.

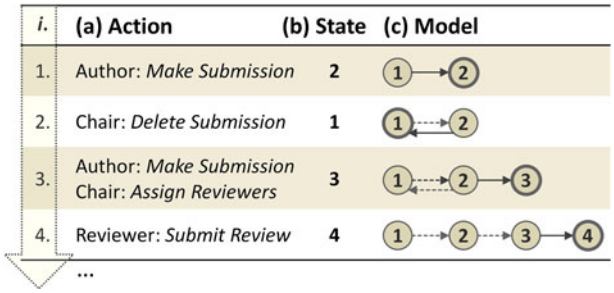

Fig. 3. **Automatic exploration**. ProCrawl iteratively **(a)** executes an action with a configured actor, **(b)** determines the state of the web application, and **(c)** updates the model.

3. *Model validation.* In absence of configured test oracles, ProCrawl cannot know if the observed behavior is correct (*oracle problem* [42]); except for strong indicators such as crashes (*weak oracles*). Therefore the extracted behavior models have to be validated, i.e. compared with the intended behavior before using them for documentation or testing. Model validation is an issue of all reverse engineering approaches; the effort highly depends on the size and complexity of the model. Therefore the models should be *concise and easy to understand*. As many other approaches, ProCrawl uses state machines for behavior modeling, which are well known and conceptually simple. State transitions are labeled with actors and sequences of UI commands and can be easily retraced by users.

However, to keep the model size small and make them more stable to UI changes, actions for navigating between UI views should not lead to new states.

To verify that the extracted models correctly represent the application behavior, tests can be generated from the models and executed on the version of the SUT the models have been extracted from. If the SUT has already been validated, it is sufficient to verify that the models do not miss important functionality.

3.2 UI Command Abstraction

Certain behavior of the SUT may only be exposed with a complex sequence of UI commands. For instance to accept a paper in OpenConf, the chair has to open the

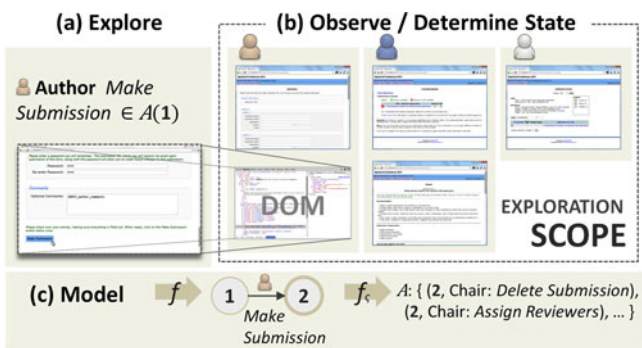

Fig. 4. **First iteration**. ProCrawl **(a)** executes an action with a configured actor, **(b)** determines the state of the SUT by applying an abstraction function over the DOM trees of the views in the exploration scope, and **(c)** updates the model.

Submission Scores view via URL, select the checkbox next to the paper and the Accept option and click the Go button. Such sequences consist of a *navigation* part to open a specific view v, optional data input commands such as selecting a checkbox, and a click command.

When navigation events become state transitions, the resulting models mix UI navigation, such as $\langle(\text{click}, \mathit{List\ Submissions})\rangle$, with system-interaction events, such as $\langle(\text{select}, \mathit{Accept}), (\text{click}, \mathit{Go})\rangle$. Since navigation events usually outnumber the system-interaction events, they effectively mask the workflow making the models less concise and vulnerable to UI changes. To raise the level of abstraction and minimize the risk of breaking the model when changing the UI, we introduce two layers of abstraction: a *model layer* (Section 3.4) that is independent of UI commands, and executable *scripts* as a second layer.

Definition 1. *A* ***script*** $\varsigma \in \Sigma$ *is a sequence of UI commands* $\langle(c, l_t, p)_n\rangle$*, where* $c \in \{open, click, type, select\}$*,* l_t *is a locator for a target* t *and* p *is a value for data input commands (type and select). ProCrawl generates scripts using a* ***script function*** $f_\varsigma(\varsigma_v, e_v) := \varsigma_v + \langle cmd_n, (click, l_e, null)_{n+1}\rangle$*, where* ς_v *is a script to open view* v*,* e_v *is a DOM element in* v*, and* cmd_n *is an optional sequence of click and data input commands. The* ***click length*** *clicks* (ς) *counts the click commands in* ς*.*

Each transition in the model represents an action (α, ς) that refers to an actor and a script encapsulating the UI commands. This is similar to a *keyword-driven testing* [37] approach, where the transition labels represent the keywords and scripts are the keyword implementations. If the SUT's UI changes, it is sufficient to update the scripts; the model is not affected.

3.3 State Abstraction

As workflow applications typically separate functionality into multiple actor-dependent views with each view exposing a specific part of the application state, ProCrawl considers multiple configured views $\mathcal{V}_\alpha$ and actors $\{\alpha\}$. Without appropriate abstraction each change in this views would produce a distinct state, resulting in large UI specific models. Therefore ProCrawl applies state abstraction:

Definition 2. *An* ***abstract multi-DOM state*** $s \in \mathcal{S}$ *is a multi-set of 3-tuples* $\mathcal{A} \times \mathcal{V} \times \mathcal{E}$ *with* $\mathcal{A}$ *being a set of actors,* $\mathcal{V} \subseteq \{\Sigma_\alpha \mid \alpha \in \mathcal{A}\}$ *a set of view navigation scripts and* $\mathcal{E} \subseteq \{\mathcal{E}_v \mid v \in \mathcal{V}\}$ *a set of abstractions over DOM elements* E *produced by a* ***state abstraction function*** $f : \mathcal{A} \times \mathcal{V} \times E \to \mathcal{S}$*.*

In the default configuration, f abstracts over visible hyperlinks, buttons and text nodes having a parent element with an id or name; this can be changed by providing XPath or CSS selectors for selecting elements from E and filters transforming the selected elements. As an example, state 01 of OpenConf consists of eight hyperlink elements across four views of the chair and four links on the submissions view of the reviewer. State 02 has 24 hyperlink, nine button and 12 text elements across the four chair views and the same four links on the submissions view of the reviewer.

To determine the application state, each of the views configured in the exploration scope is opened using the script provided for the view. Opening a view rebuilds the DOM tree, removing all changes that were caused by previous navigation events. Hence, a new state s always represents a change in $\mathcal{E}$ of one or more actor views. Pure navigational event sequences therefore do not affect the abstract state determined by ProCrawl. Events that do affect the state are considered as system-interaction events and finally become transitions in the model. This is a major difference to other reverse engineering approaches for web applications such as [3], [24] and [26], which only consider the single DOM tree rendered at the time the state abstraction is performed and cannot distinguish between navigation and system-interaction events.

In our evaluation this abstraction proved to be effective in identifying the different workflow states. However, when the state of a system/workflow depends on previous data input that is not reflected in the UI, such as Restore Submissions in state 06 of OpenConf, the resulting model may be nondeterministic, i.e. not precise enough for test generation. Section 3.7 describes a method for learning transition guards that effectively resolved nondeterminism in our case studies.

3.4 Application Behavior Models (ABMs)

ProCrawl captures application behavior in form of an *extended finite state machine* [9], i.e. a *finite state machine* with additional *data state variables*, *transition guards* which must hold with respect to the data state variables to enable a transition, and *update functions* changing the state variables when a transition is executed. More formally

Definition 3. *An* ***extended finite state machine*** *is a tuple* $(S, s_0, \Lambda, D, G, \delta)$ *where* S *is a set of states,* $s_0 \in S$ *is the initial state and* Λ *is a set of labels. The global data state* D *with* $D \cap S = \emptyset$ *is a set of tuples* $\{(variable, value)\}$*;* G *is a set of enabling functions (guards)* $g : D \to \{true, false\}$*, and transitions* $\delta : S \times D \times \Lambda \to S \times D$*.*

A transition $\delta(s, d, \lambda) = (s', d')$ *is called enabled iff the EFSM is in state* s *and* $g(d) = true$*; we also write for short* (s, λ, s') *and* $u(d) = d'$*, where* u *is called update function.*

Definition 4. *An* ***application behavior model*** *is an EFSM* $(\mathcal{S}, s_0, \Lambda, D, G, \delta)$ *where* $\mathcal{S}$ *is a set of abstract multi-DOM states,* $\Lambda = \mathcal{A} \times \Sigma_k$ *with* Σ_k *being a set of scripts* $\{\varsigma \mid clicks(\varsigma) \leq k\}$*.* ABM_k *is a depth* k *ABM.*

We visualize ABMs as state diagrams (cf. Fig. 1). The update functions and guards of the ABM transitions are computed based on the input data provided by ProCrawl (cf. Section 3.6 & Section 3.7) and annotated to the transitions in the ABM diagram. To reduce the diagram size, transitions with the same actor, source and target state are merged, i.e. a transition may represent multiple scripts that have the same effect on the SUT's state.

3.5 Workflow Exploration

ProCrawl maintains a set of *pending actions* $A(s)$ for each state s, where a pending action for a DOM element e_v on view v_α of actor α is a tuple $(\alpha, \varsigma_\varepsilon)$. During behavior exploration, ProCrawl removes and executes an action from $A(s)$ for the current state s, determines the new state s', updates the ABM, possibly adds new actions to be explored to $A(s')$

and iterates until $A(s)$ is empty for all $s \in \mathcal{S}$ or a configurable stop condition holds. However, since the number of possible event sequences in GUI applications is usually too large to be explored exhaustively, ProCrawl has to sample and restrict the sequence length (click depth). Therefore, instead of generating click events for all (or a configured subset of) DOM elements in s', ProCrawl by default only considers DOM elements that were not present in the initial state s_0, i.e. $s' \setminus s_0$, where $A(s_0) = \{a_0\}$ with a_0 being a configured start action such as (Author, Make Submission) in OpenConf. Hence, $A(s)$ only contains actions that were not present before executing a_0, i.e. having a causal relationship to a_0. Consequently, the behavior models only contain actions that are part of a causal chain as it is common in workflows, which improves the readability and maintainability of the models.

If the state of the SUT did not change after executing an action (α, ς), it is considered as not workflow relevant. In this case, ProCrawl compares the DOM tree before with the one after executing the action and, provided that ς is below a configurable click depth, generates additional scripts $\langle \varsigma + (click, l_e, null) \rangle$ for each volatile DOM element e that was added to the DOM.

ProCrawl systematically explores the behavior graph using a weighted nearest neighbor approach, i.e. if $A(s) = \emptyset$, a path to the pending state with the smallest distance from s is computed, weighted by the number of UI commands to be executed. If there is no known path from s to a pending state or a configurable reset condition holds, ProCrawl resets the SUT to s_0 by executing the configured test fixture.

Together with the applied state abstraction, only considering nonvolatile DOM elements, this approach proved to be effective in exploring the workflows in our case studies. However, for applications where all actions are already active in s_0, as common in editors the approach would fail.

3.6 Input Data Provisioning

In contrast to inductive analysis approaches, which extract models from multiple given execution traces, ProCrawl generates executions and has to provide input data in order to reach certain states of the SUT. This includes input choices such as selects, checkboxes and radio buttons, as well as textual input. In case of multi-choice inputs, i.e. a combination of input choices, ProCrawl generates multiple runs testing different combinations. However, in case of a large number of choices, it is infeasible to test every possible combination (combinatorial explosion). Therefore, ProCrawl by default applies a deterministic combination strategy generating actions for a tractable subset of all combinations: For each choice element $(\text{select}, l, option_n)$ with n possible options, ProCrawl generates $n - 1$ scripts, one for each non-default option, while setting the other choice elements to their default.

Textual input is much harder to generate, especially since web applications usually do not provide data type annotations for input fields. If the SUT annotates input elements with W3C WAI-ARIA[6] states, ProCrawl can repeatedly generate textual input and determine the validity via the aria − invalid attribute. However, for structured input data that is hard to generate, ProCrawl has to rely on user defined input (cf. [2]).

Input data tracing. Generated input data may end up in the DOM tree affecting the state abstraction of ProCrawl and cause a state explosion [38]. Therefore, ProCrawl traces input data in the DOM and replaces matching text nodes with a reference to the data source.

3.7 Transition Guard Learning

With the procedure described above, ProCrawl can already derive suitable models. Sometimes however, the ABM is nondeterministic. While this is natural as the system state is only partially reflected in the UI and ProCrawl applies state abstraction, nondeterministic transitions lead to problems during test generation. In our example (Fig. 1) for instance, withdrawing a paper leads to State 06, regardless of the previous state, i.e. the status of the paper (accepted, rejected, pending) is not reflected in the UI anymore. Restoring the submission also restores the status, leading to two Restore Submissions transitions with different target states. However, when using nondeterministic models as input for test generators, these may produce paths through the model that are infeasible in the SUT. Therefore, ProCrawl strives to resolve nondeterminism by learning mutually exclusive *guard conditions* on global state variables computed from data input elements in the DOM trees and the input data generated during behavior exploration. These guard conditions must evaluate to true in order to enable the transition. For instance the Restore Submissions transition $6 \rightarrow 5$ is only enabled in State 06 if a decision has been taken, i.e. the paper has been accepted or rejected. Nondeterministic transitions are then annotated with the inferred guards, producing a deterministic EFSM.

ProCrawl tries to automatically learn such guards by creating a classification problem over the provided input data in the path history (data trace) of a nondeterministic transition set. This allows us to use well established classification algorithms.

Definition 5. *The **path history** $h(t)$ of a transition t returns the set of transition sequences $\{\langle (s, \alpha, \varsigma, s')_n \rangle\}$, the crawler took before traversing t, where each sequence starts on the execution of the test fixture, i.e. in state s_0.*

Definition 6. *The **data trace** $d(h(t))$ returns the set of data input sequences $\{\langle (variable, value)_m \rangle\}$ from $h(t)$.*

Script commands in $h(t)$ with type select, including checkboxes, selects, radio- and submit buttons, become nominal variables in $d(h(t))$; text input (type) commands become string variables.

The goal is to find a classifier that correctly predicts the target state of a nondeterministic transition based on its data trace. In order to build such a classifier we need to construct a training set from the data traces.

Training set construction. Learning from data traces, i.e. temporal sequence data including discrete, unordered elements requires us to convert the data to an atemporal representation. Table 2 shows an excerpt from the training set $\mathcal{T}$ generated by ProCrawl for OpenConf. Each row contains a

6. http://www.w3.org/TR/wai-aria/states_and_properties

TABLE 2
Data for Learning the Guard Conditions of the Nondeterministic Restore Submissions Transition in Fig. 1

	Input Variables (attributes)				Target State
	scores_boxselect info gain: 0.311	subs_subaction 0	scores_asubmit 0.311	scores_subaction 1.0	class
1	all submissions	Withdraw Submissions	Go	Reject	**05**
2	all submissions	Withdraw Submissions	Go	Accept	
3	default	Withdraw Submissions	default	default	**02**
4	all submissions	Withdraw Submissions	Go	Pending	

feature vector ν with four variable assignments from the data trace of the Restore Submissions transition and the target state of the transition, i.e. the class label. For instance ν_1 is constructed from the data trace of a path from state 01 to 06 that leads to state 05 on Restore Submissions. In case a variable is assigned multiple times in the data trace, only the last assignment is added to ν. $\mathcal{T}$ is then used to build the classifier from which we derive the decision rules that become the guard conditions shown in Fig. 1. More formally (cf. [39]):

Definition 7. *A **data trace classifier** $\phi : D_N \rightarrow \mathcal{S}$ for a nondet. transition set N is a function that maps a data trace $d(t) \in D_N$ to an abstract multi-DOM state $s \in \mathcal{S}$. Given a function space $\mathcal{H}$, the **learning problem** can be described as selecting a function ϕ from $\mathcal{H}$ that minimizes the empirical risk of misclassification, which is computed by averaging a fixed loss function on the training set $\mathcal{T}$. To prevent overfitting $\mathcal{T}$, regularization methods penalizing a high complexity of ϕ are used.*

The more complex ϕ, the more difficult to interpret the classifier, which is particularly important considering the fact that we need to derive decision rules from the classifier. By default, ProCrawl uses decision tree classifiers [31], which are easy to interpret. Decision trees can be directly transformed into guard conditions. We first rewrite the decision tree to decision rules of the form `IF` $c_1 \wedge \ldots \wedge c_n$ `THEN` t where c_i are conditions on the path from the root of the tree to the leaf node t which is the transition that is taken. The guard condition for transition t is created by connecting all decision rules for t via disjunction.

Active learning. If the number of samples (feature vectors in $\mathcal{T}$) is too low given the number of different attributes, the risk of overfitting the training data is high; this results in a classifier that performs well on the training data but poorly on unseen samples. For instance, at the first occurrence of the nondeterministic Restore Submissions transition $6 \rightarrow 2$, Table 2 only contains rows 1-3 and classification can be correctly done based on the value of the first attribute. However, in fact the value of scores_boxselect has no effect on the final state (class) and would perform poorly on unseen samples. To decrease the risk of overfitting the training data, ProCrawl iteratively generates runs to produce additional samples with different values for the attributes. The attributes are ranked based on their *information gain*[7](cf. Table 2), i.e. ProCrawl starts generating values for attributes with a high information gain. In our example, ProCrawl generates a path to state 06 with scores_subaction set to the unobserved value Pending, executes the Restore Submissions action and determines the state, yielding row 4. Technically, the path computation corresponds to finding a counter example to the LTL formula

$$\Box(state = 06 \Rightarrow scores\_subaction \neq Pending),$$

which is computed using a model checker on the ABM.

Limitations. While this approach worked well in our case studies, learning from a limited number of observations is not guaranteed to find correct and concise guard conditions. If a variation in behavior that results in a nondeterministic transition does not depend on the provided input data but for instance on a specific sequence of states, learning from data traces fails. In this case an alternative is to mine discriminating LTL formulas over state sequences, such as $\Box(state = 01 \rightarrow \Diamond state = 02)$, which we did in an earlier approach. However, these are harder to interpret and use as transition guards.

4 Implementation

ProCrawl is implemented in Java, using Selenium [8] for web browser automation and provides a plug-in architecture for writing extensions, which is used for ABM and script serialization, as well as transition guard learning. For each configured actor one or multiple web browsers are started, on which the UI commands are executed. Each time ProCrawl detects a new (nonreflexive) transition, the executed UI commands are exported as a Selenium script and the behavior model is exported in GraphML[9] format. Additionally, ProCrawl exports states ($\mathcal{A} \times \mathcal{V} \times \mathcal{E}$) in text format, making them easily comparable using a diff tool.

Algorithm 1 shows the initialization of the exploration procedure for mining the behavior model. First, a global driver pool is created that is used to interact with the SUT's web UI (Line 3). Then, the initial state s_0 of the SUT is determined (Line 4) and an ABM containing s_0 is created (Lines 2, 5). As described before, ProCrawl keeps a set of *pending actions* $A(s)$ for each state s, which are executed to explore the SUT's behavior; $A(s_0)$ is initialized with the start action defined in the configuration (Line 6). The plug-ins are initialized (Line 7) and the exploration procedure is called on s_0 (Line 8). After the exploration finished, the plug-ins are notified (Line 9).

7. $InfoGain(class, attribute) = H(class) - H(class|attribute)$, where H denotes the information entropy.

8. http://seleniumhq.org
9. http://graphml.graphdrawing.org

Algorithm 1. MAIN

```
    Input: config
1   global A ← new HashMultimap();
2   global ABM_k ← new AppBehaviorModel();
3   global driver ← initDriverPool(config);
4   s_0 ← determineState();
5   ABM_k.initialState ← s_0;
6   A(s_0) ← {config.startAction};
7   plugins.explorationStarted(ABM_k, config);
8   explore(s_0);
9   plugins.explorationFinished();
```

The state of the SUT is determined as shown in Algorithm 2. As described in Section 3.3, a state is defined by the set of elements extracted from the DOM trees of multiple actor/view relations. For each actor α, the DOM trees of the views defined in the actor's exploration scope are retrieved in parallel (Line 3). After removing certain nodes such as invisible HTML elements from the DOM tree (Line 4), elements matching the selection criterion are added to the state (Line 6). Two states s, s' are considered equal, iff they have the same set of distinct elements.

Algorithm 2. DETERMINESTATE

```
    Data: config
    Output: current state s of the SUT
1   s ← {};
2   foreach (α, ς_v) ∈ config.exploration_scope do
3     dom ← driver.getDom(α, ς_v);
4     f_dom ← filter(α, dom);
5     foreach e ∈ select(α, f_dom) do
6       s ← s ∪ f(α, ς_v, e);
7     end
8   end
9   return s;
```

Algorithm 3 shows the exploration procedure recursively building up the behavior model. In each recursion the current state s of the SUT is checked for pending actions. If $A(s)$ is not empty, a pending $action$ is removed (Lines 2–3) and executed (Line 4), returning DOM elements (ΔE) that have been added to the DOM tree as a result of the execution. After that, the current state s' of the SUT is determined (Line 5). If $s' \neq s$, the ABM is updated (Lines 7–17): First, if $s' \notin \text{ABM}_k.\mathcal{S}$, s' is added to the ABM, the plug-ins are notified, and for each tuple $(\alpha, \varsigma_v, \varepsilon) \in s' \setminus s_0$, a new action $(\alpha, \langle \varsigma_v + (\text{click}, l_\varepsilon, \text{null}) \rangle)$ is added to $A(s')$ (Lines 8–13). Second, a new transition t is added to the ABM and the plug-ins are notified (Lines 15–17).

If the state did not change, ProCrawl considers the executed $action$ as not workflow relevant, i.e. the ABM is not updated. However, the $action$ may enable workflow-relevant actions; therefore, provided that a configurable click depth k is not reached (Line 18), for each DOM element $e \in \Delta$E, a new action $(\alpha, \langle \varsigma + (\text{click}, l_e, \text{null}) \rangle)$ is added to $A(s)$ (Lines 19-22). Finally, the exploration procedure is called recursively on the current state s' (Line 24).

If there are no pending actions in $A(s)$, but the set of states with pending actions is not empty (Line 25), Algorithm 4 is called to compute and execute a path to a pending state s_p (Line 26), and the exploration procedure is recursively called on s_p (Line 27). This procedure continues until A is empty.

Algorithm 3. EXPLORE

```
    Input: current state s of the SUT
1   if A(s) ≠ ∅ then
2     action ← (α, ς) ∈ A(s);
3     A(s) ← A(s) \ {action};
4     ΔE ← driver.execute(action);
5     s' ← determineState();
6     if s' ≠ s then
7       if s' ∉ ABM_k.S then
8         ABM_k.S ← ABM_k.S ∪ {s'};
9         plugins.stateAdded(s');
10        foreach (α, ς_v, ε) ∈ s' \ s_0 do
11          ς_ε ← ⟨ς_v + (click, l_ε, null)⟩;
12          A(s') ← A(s') ∪ {(α, ς_ε)};
13        end
14      end
15      t ← (s, action, s');
16      ABM_k.trans ← ABM_k.trans ∪ {t};
17      plugins.transition(t);
18    else if clicks(ς) < κ then
19      foreach e ∈ ΔE do
20        ς_e ← ⟨ς + (click, l_e, null)⟩;
21        A(s) ← A(s) ∪ {(α, ς_e)};
22      end
23    end
24    explore(s');
25  else if {s ∈ ABM_k.S | A(s) ≠ ∅} ≠ ∅ then
26    s_p ← gotoPendingState(s);
27    explore(s_p);
28  end
```

Algorithm 4. GOTOPENDINGSTATE

```
    Data: config
    Input: current state s of the SUT
    Output: pending state s_p
1   S_p ← {s ∈ ABM_k.S | A(s) ≠ ∅};
2   Transitions ← nearestNeighbor(s, S_p, ABM_k);
3   if Transitions = ∅ then
4     executeFixture(config.testFixture);
5     s_0 ← ABM_k.initialState;
6     return gotoPendingState(s_0);
7   end
8   foreach (s, action, s') ∈ Transitions do
9     driver.execute(action);
10  end
11  return determineState();
```

Algorithm 4 computes the shortest path from the current state s of the SUT to the nearest neighboring pending state $s_p \in S_p$ (Line 2). If no path was found, the SUT is set to the initial state s_0 by executing a configured test fixture, and the procedure is called recursively on s_0 (Lines 4–6), otherwise the actions in the path are executed and s_p is returned (Lines 8-11).

Algorithm 5. LEARNTRANSITIONGUARDS

```
Input: transition t_0 ← (s, a, s')
1  N ← {t ∈ Transitions | t.s = s ∧ t.a = a ∧ t.s' ≠ s'};
2  if N ≠ ∅ then
3    T ← CONSTRUCTTRAININGSET(N ∪ {t_0});
4    foreach a ∈ RANKATTRIBUTES(T) do
5      {(v, s)} ← EXPLOREPATHS(ABM_k, a);
6      T ← T ∪ {(v, s)};
7    end
8    foreach (t, r) ∈ LEARNDECISIONRULES(T) do
9      ABM_k.ADDGUARD(t, r);
10   end
11 end
```

Transition guard learning (Algorithm 5) is implemented as a plug-in listening to ABM change events. First, the set of nondeterministic transitions N is determined (Line 1), containing all transitions with the same source state and action as t_0, but different target states. Second, the training set is constructed from the data trace of the nondeterministic transitions (Line 3), as described in Section 3.7. Each tuple of the training set consists of a feature vector v with the variable assignments extracted from the data trace, and the resulting target state s' of $t \in N$, which is the class label in the classification problem. Third, the input variables (attributes) are ranked based on the *information gain* (Line 4) and for each attribute a additional paths are computed and executed to produce additional training data for building the classifier (Lines 5-6). Therefor we transform the ABM into an *abstract machine* in *B notation* [1], i.e. a set of states and operations modifying the state, and use the ProB [43] model checker to generate unexplored paths with predefined data input as described in Section 3.7. For building the classifier used to learn transition guards, we use a variation of the C.4.5 algorithm [31] provided by Weka,[10] derive decision rules and add them as guards to the ABM (Lines 8-10).

5 Evaluation

To assess the effectiveness of ProCrawl in mining workflow models, we conducted case studies (cf. [44]) on a commercial SAP and several popular open-source web applications with 16-259K LOC.

5.1 Evaluation Methodology

This section reports the target applications (evaluation subjects), formulates the research questions and describes the evaluation process and environment.

5.1.1 Evaluation Subjects

OpenConf[11] v5.30 *Community Edition* is an open-source peer-review and conference management system with about 16K lines of PHP and 384 lines of JavaScript code.

Oxid eShop[12] v4.8.5 *Community Edition* is an open-source e-commerce platform with about 244K lines of PHP and 8K lines of JavaScript code using the jQuery[13] library. The web UI consists of a front-end interface for ordering products, and a back-end interface for shop administration and order processing.

OrangeHRM[14] v3.1.1 is an open-source HR management system with about 222K lines of PHP code and 37K lines of JavaScript code.

MantisBT[15] v1.2.17 is an open-source issue tracker with about 117K lines of PHP and 462 lines of JavaScript.

In our original paper [33] we applied an earlier version of ProCrawl, without input data variation and transition guard learning, on OpenConf, Oxid and an SAP OpenUI5 application (93K SLOC/Java) for exchanging product compliance information.

5.1.2 Research Questions and Metrics

The main focus of the case studies is to assess the accuracy of the workflow models inferred by ProCrawl. The extracted ABMs may be under and/or over approximating the actual workflow of the target application. In an under-approximating ABM, workflow-relevant states and/or transitions are missing. An over-approximating or nondeterministic ABM allows traversal paths that are not feasible in the target application, which causes false positives when using the model for test generation.

We address the following research questions:

- **RQ1** *What is the **precision** of the extracted ABMs? Do they contain navigational or other actions that are not workflow-relevant?*
- **RQ2** *What is the level of **under-approximation** of the extracted ABMs? Are there workflow-relevant actions that are not represented in the model?*
- **RQ3** *Does **transition guard learning** successfully resolve nondeterminism in the models? What is the runtime overhead of transition guard learning?*
- **RQ4** *Are the extracted ABMs and scripts usable as input to **model-based testing**?*
- **RQ5** *What is the level of **over-approximation** of the extracted ABMs? Do they correctly model the target workflow? How many paths in the ABM are infeasible?*
- **RQ6** *Are the test cases generated from the ABMs useful for **system regression testing**?*

We used ProCrawl to extract workflow models for the core processes of the target applications and manually compared the models with the applications and available documentation (Section 5.2). To answer **RQ1**, we manually investigated the *target workflow* identifying the workflow steps and relevant actions, i.e. the sequences of UI commands necessary to execute the workflow and computed the *precision*:

$$\frac{\left|\{relevant\ ABM\ transitions\}\right|}{\left|\{ABM\ transitions\}\right|}.$$

To evaluate the level of under-approximation (**RQ2**), we checked if the target workflow can be replayed using the

10. http://www.cs.waikato.ac.nz/ml/weka
11. http://www.openconf.com
12. https://www.oxid-esales.com
13. http://jquery.com
14. http://www.orangehrm.com/OpenSource
15. http://www.mantisbt.org

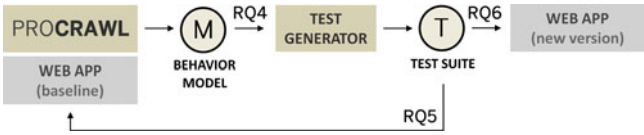

Fig. 5. **Test generation**. We extract a behavior model from the baseline application, use the model as input for a test generator (**RQ4**), run the generated test cases on the baseline application to ensure they succeed (**RQ5**) and on the new version of the web application to detect changes (**RQ6**).

extracted ABM. We counted the number of transitions that need to be added to the ABM in order to replay the target workflow and computed the *recall*:

$$\frac{\left|\{relevant\ ABM\ transitions\}\right|}{\left|\{relevant\ ABM\ transitions\}\right| + \left|\{missing\ transitions\}\right|}.$$

Furthermore we computed the harmonic mean of precision and recall (F_1-score), as well as the $F_{0.5}$- and F_2-score, which put a higher weight on precision or recall, respectively.

To answer **RQ3**, we conducted runs with and without transition guard learning, counted the number of nondeterministic transitions in the extracted ABMs and manually checked if the inferred transition guards are correct and resolve the nondeterminism.

Fig. 5 illustrates the second part of the evaluation (Section 5.3): Based on the extracted ABMs we generated tests using a model-based test generator (**RQ4**, Section 5.3.1). To answer **RQ5**, we executed the tests on the target application (Section 5.3.2) and counted the number of failing tests, which indicate infeasible paths. Furthermore, we executed the tests on different versions of the target application (**RQ6**, Section 5.3.3) and investigated the root cause of failing tests.

All generated models, scripts and log files are publicly available (cf. Section 7).

5.1.3 Evaluation Process and Environment

The evaluation was performed on an Intel i5-3360M 2.8 GHz with 16 GB RAM, running Windows 7 x64 with JDK 8u25 and Selenium/Java 2.43.1.

For each of the case studies we:

1) downloaded and installed the target application on a local web server;
2) manually navigated through the application using a web browser to identify the user roles and views for setting the exploration scope and the start action of the target workflow;
3) configured ProCrawl with the identified actors, views, start action, a test fixture and the click depth # (default: 2), and started ProCrawl with (DG#) and without (D#) transition guard learning;
4) manually investigated the *target workflow* and identified workflow-relevant actions;
5) compared the extracted ABMs with the target workflow to answer the research questions.

TABLE 4
Executions Generated by ProCrawl

App. (D# | DG#)	Traces	Script Exec.	Time (*hh:mm*)
OpenConf (#:2)	36 | 42	700 | 728	00:58 | 01:05
Oxid (#:4)	28 | 30	642 | 651	01:11 | 01:17
OrangeHRM (#:2)	17 | 17	130 | 130	00:22 | 00:22
MantisBT (#:2)	47 | 74	1,784 | 1,891	02:02 | 02:25

5.2 Model Accuracy

In this section we evaluate *precision* and *recall* of the models extracted by ProCrawl (cf. Table 3). We configured ProCrawl to run with Chrome v38 (64-bit). Table 4 summarizes the two runs generated for each of the four case studies. ProCrawl analyzed 294-7,860 DOM trees and performed between 130 and 1,891 script executions, with each script comprising 1-16 UI commands. The script executions were grouped into 17-74 traces; each trace starting after executing the configured test fixture setting the SUT to the initial state. Over all runs the exploration time was 22-145 minutes, depending on the configuration of ProCrawl and the complexity and responsiveness of the SUT.

Table 5 reports statistics about the length of the generated traces for the four D# configurations. The overall avg. trace length is 122.4 UI commands, with a minimum of 2 and a maximum of 753 commands.

5.2.1 Study 1: OpenConf Peer-Review

Target workflow. The OpenConf peer-review process involves an *author*, a *reviewer* and a *chair*. A typical usage scenario looks like this:

TABLE 3
Precision, Recall and F_β Score of ProCrawl in Terms of Process-Relevant Actions Covered in the ABMs

Subject	SLOC	Config *click depth*	States # (DOM trees)	Transitions # (*merged* | *nondet.*)	Scripts #	Precision (*transitions*)	Recall (*transitions*)	F_β-score β: 0.5 / 1 / 2
OpenConf	PHP+JS 16K/384	D2	8 (3680)	117 (31 | 2)	31	**1.0** (31/31)	**0.84** (31/37)	**0.96 / 0.91 / 0.87**
		DG2	8 (3850)	117 (31 | 0)	31			
Oxid	PHP+JS 244K/8K	D2	5 (330)	18 (13 | 0)	15	**1.0** (13/13)	**0.22** (13/60)	**0.58 / 0.36 / 0.26**
		D4	12 (2010)	95 (49 | 2)	56	**1.0** (49/49)	**0.82** (49/60)	**0.96 / 0.90 / 0.85**
		DG4	12 (2043)	95 (49 | 0)	56	**1.0** (49/49)	**0.82** (49/60)	**0.96 / 0.90 / 0.85**
OrangeHRM	PHP+JS 222K/37K	D2	5 (294)	16 (06 | 0)	14	**1.0** (06/06)	**1.00** (06/06)	**1.00 / 1.00 / 1.00**
		DG2	5 (294)	16 (06 | 0)	14			
MantisBT	PHP+JS 117K/462	D2	8 (7324)	136 (37 | 4)	35	**1.0** (37/37)	**0.82** (37/45)	**0.96 / 0.90 / 0.85**
		DG2	8 (7860)	136 (38 | 0)	35	**1.0** (38/38)	**0.84** (38/45)	**0.96 / 0.92 / 0.87**

(*Config:* D = default, C = custom, G = transition guard learning, #= click depth).

TABLE 5
Number of UI Commands in Generated Traces

Trace length	OpenConf	Oxid	OrangeHRM	MantisBT
min.	23	2	8	33
avg.	113.9	146.3	26.2	203.1
max.	474	753	101	703

(1) The *author* completes the submission form and clicks 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 completing the review form and clicking the Submit Review button.
(4) The *chair* changes the submission status (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. Also the *chair* can withdraw a submission; and additionally delete submissions and restore withdrawn submissions. After Step 2, the *chair* can unassign reviewers from a submission.

Setup. We conducted two runs: Default configuration (D2) without and (DG2) with transition guard learning. The configuration was provided as described in Section 3.1 (1), with the default click depth of 2.

Findings. With both configurations, ProCrawl generated 31 Selenium scripts and mined the ABM_2 shown in Fig. 1; the transition guards were only included in the DG2 model. As described in Section 1, the ABM consists of 8 states and 117 transitions which were merged to 31 transitions.

With DG2, ProCrawl learned concise guard conditions resolving nondeterminism in the Restore Submissions transitions. To learn the transition guards, ProCrawl additionally executed 28 scripts causing a runtime overhead of 7 minutes (12 percent, overall 65 min).

All of the 31 transitions represent workflow-relevant actions, resulting in a *precision* of 1.0. However, the extracted ABM is not able to replay the Author : Withdraw Submission[16] action of the target workflow, which should be present in state 02-05, 07 and 08, i.e. six missing transitions, resulting in a *recall* of 0.84 (31/37).

5.2.2 Study 2: OXID eShop Ordering

Target workflow. The Oxid ordering process involves a *customer* and a *retailer*. The customer interacts via the shop front end, whereas the retailer operates via a separate backend 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 the Next Step button or 4. Order link.
3) The *customer* orders the product by clicking the Order nowbutton 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

- click the add link in the shopping cart, select a gift wrap and a greeting card via radio button and click the Apply button.
- select an article in the shopping cart and click 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 order status and prevents shipment (Step 4).[17]
- 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 actions related to that order.
- download an invoice by clicking the Create PDF button in the orders view.

After Step 4 the *retailer* may click the Reset Shipping Date button in the orders view.

Setup. We conducted three runs: Default configuration (D2) with click depth 2, (D4) with depth 4, and (DG4) with transition guard learning. In all runs we configured two actors representing a *customer* and a *retailer*, provided login credentials and a start script adding a product to the shopping cart. Furthermore, we added the URLs to the shopping cart and the order history of the customer, as well as a script to access the orders list for the retailer to the *exploration scope*.

Findings. With D2, ProCrawl did not pass through the ordering steps in the shopping cart and finished exploration within 9 minutes without submitting the order, resulting in a low recall of 0.22 (13/60). With D4 and DG4, ProCrawl mined an ABM_4 with 12 states and 95 transitions (merged: 49) and generated 56 Selenium scripts. With DG4, ProCrawl executed nine more scripts and learned concise transition guards with a low runtime overhead of 8 percent (overall: 77 min).

Since the DG4-ABM is too large to include it in this paper, we conducted another run with a custom configuration (CG4) removing the gift wrap and greeting card selection by adding a CSS DOM filter (`.wrappingTrigger,.wrapping`). With CG4, ProCrawl mined the ABM_4 shown in Fig. 6, consisting of six states and 43 transitions (merged: 15), in 45 minutes, generating 38 scripts.

All transitions covered in the ABMs represent workflow-relevant actions, resulting in a *precision* of 1.0. Concerning the recall, none of the ABMs includes the Submit Coupon, as well as the Create PDF action. Submit Coupon can be executed in each of the eight states

16. To execute the action, ProCrawl would need to provide the submission id displayed after submitting the paper, as well as the password provided in the submission form.

17. However, the Ship Now button is not deactivated.

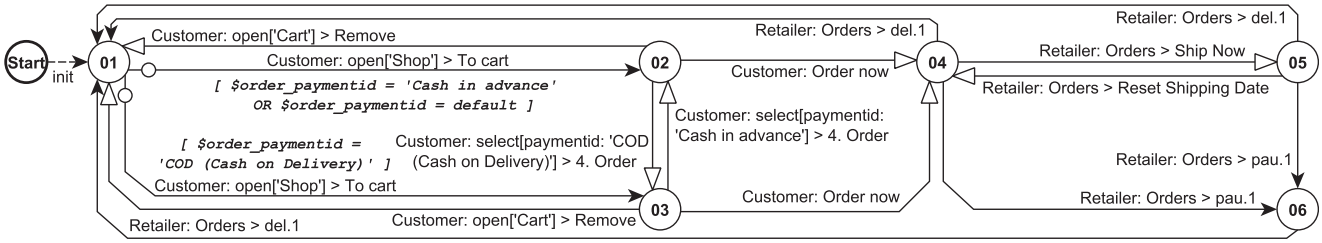

Fig. 6. The ProCrawl ABM_4 (CG4) of the ordering process in Oxid eShop (del.1 = delete; pau.1 = cancel).

between adding an article to the shopping cart and ordering; it was executed in all four runs, but ProCrawl was not able to generate a valid coupon code. Create PDF can be executed in each of the three states after ordering and was clicked in the three depth four runs. However, it did not change any elements considered for state abstraction. This results in 11 missing transitions summing up to 60 workflow-relevant transitions from which 49 are in the ABMs, resulting in a *recall* of 0.82.

5.2.3 Study 3: OrangeHRM Vacation Request

Target workflow. The vacation request process in OrangeHRM involves an *employee* and a *manager*:

(1) The *employee* submits a vacation request by selecting the leave type Vacation, specifying the date and clicking the Apply button.
(2) The *manager* handles the request by selecting an action (Reject, Approve) on the leave list and clicking the Save button.

As long as the vacation request has not been rejected, it can be canceled by the *employee* or the *manager*.

Setup. We configured ProCrawl with the two aforementioned actors, provided login credentials and a start action submitting a vacation request, and added the URL to the leave list to the *exploration scope*.

Findings. ProCrawl mined the ABM_2 shown in Fig. 7 in 22 minutes, generating 14 Selenium scripts. Transition guard learning did not intervene, since the ABM_2 does not have nondeterministic transitions. The ABM consists of five states and 16 transitions (merged: 6). All of the six transitions represent actions that are relevant for the leave request process and all relevant actions are covered by the ABM_2, resulting in a *precision* and *recall* of 1.0.

5.2.4 Study 4: Mantis Bug Tracker

Target workflow. The issue tracking process in MantisBT involves a *reporter* and a *developer*. According to the documentation, an issue can be in one of three stages: opened (status: new, acknowledged, confirmed, assigned, or feedback), resolved and closed.

The issue workflow is as follows:

(1) The *reporter* completes the form on the bug report view and clicks the Submit Report button. (new)
(2) The *developer* assigns the issue by opening the bug report, selecting a developer (or none to unassign) and clicking the Assign To button.
(3) The *developer* changes the issue status by selecting the report, setting the action-selection box to Update Status, clicking the OK button, selecting the new status and clicking the Update Status button.

The *developer* can change the issue status and assignee or delete an issue in any stage. In stage opened, the *reporter* may add notes by opening the bug report, typing a note and clicking the Add Note button. In stage resolved and closed, both actors may reopen the issue by opening the report, clicking the Reopen button, adding a note and clicking the Request Feedback button in which case the issue status is changed to feedback.

Setup. We conducted two runs: (D2) Default configuration without, and (DG2) with transition guard learning. In both cases we configured two actors representing a *reporter* and a *developer*, provided login credentials and a start action clicking Submit Report; input data for the report form was not provided. The *exploration scope* was defined by providing the URLs to the issues list and the home view for both actors.

Findings. With D2, ProCrawl mined an ABM_2 consisting of eight states and 136 transitions (merged: 37) in 122 minutes, generating 35 Selenium scripts. With DG2, ProCrawl executed 107 more scripts and finished the exploration process in 145 minutes, which is an overhead of 19 percent. All transitions in both ABMs represent process-relevant actions, resulting in a *precision* of 1.0. Due to space restrictions, we cannot show the ABMs here and have to refer to the annex (Section 7).

Besides the initial state (01) with no issue submitted, the ABMs consist of a state for status new (08) and assigned (03),

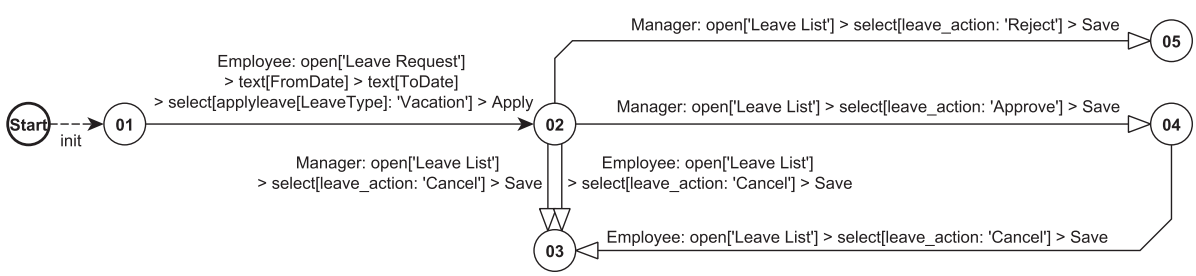

Fig. 7. The ProCrawl ABM_2 (D2|DG2) of the leave request process in OrangeHRM.

and a single state for acknowledged, confirmed and feedback (06). For each of the three latter states there is an additional state (02, 04, 07) with the same status but no assignee. The resolved and closed stages are merged in a single state (05). Merging closed with resolved might result in a precision loss, however comparing the ABMs with the actual system behavior revealed that the set of enabled actions is the same in all three stages, i.e. the issue status can be changed in any state. In contrast to the ABM states for stage opened, there is no distinction between issues with and without an assignee in state 05 (resolved/closed). Changing the status of an issue from resolved or closed to one of the five statuses from the opened stage, reveals the assignee, resulting in a nondeterministic model. ProCrawl explored this behavior when changing the status to new and acknowledged and learned the following transition guards for the new transitions:

$$
\begin{aligned}
&5 \rightarrow 2 \ \texttt{resolve = default} \wedge \texttt{handler\_id = default} \\
\vee\ &\texttt{resolve = default} \wedge \texttt{handler\_id = null} \\
&5 \rightarrow 8 \ \texttt{resolve = ResolveIssues} \\
\vee\ &\texttt{resolve = default} \wedge \texttt{handler\_id = dev@test.com} \\
\vee\ &\texttt{resolve = default} \wedge \texttt{handler\_id = [Myself]}
\end{aligned}
$$

Resolving an issue via the Resolve Issue button automatically assigns the issue, otherwise a developer has to be explicitly set as a handler. The first transition guard represents the issue state without an assignee, the second guard represents the state with assignee. The guards for the acknowledged transitions are similar. However, ProCrawl did not explore the guards for the confirmed, assigned and feedback actions, which show similar behavior. This is because the behavior is only discovered when taking a different path to state 05 and then changing the issue status by executing a nonpending action. ProCrawl executes nonpending actions in order to reach a pending state and tries to resolve nondeterminism if a behavior change is detected. However, in general, ProCrawl cannot systematically explore all paths through an ABM, since for realistically sized applications the number of different paths is usually infinite. Furthermore, ProCrawl merged the issue statuses acknowledged, confirmed and feedback into a single state 06 (07 in case of an assignee). While this seems valid, since all three statuses are from the opened stage, MantisBT shows a variation in behavior when executing the Add note action: only if the issue status is feedback, the status changes to assigned (state 03), or new (state 02) in case the issue was not assigned before. This behavior only appears when entering state 06 or 07 via the feedback transition and then executing the Add note action and was not detected by ProCrawl. Considering the three missing alternative transitions for the confirmed, assigned and feedback actions, as well as the four transitions for the Add note behavior, results in a *recall* of 0.82 (37/45) for D2 and 0.84 (38/45) for DG2.

5.2.5 Summary and Discussion

ProCrawl mined behavior models with *full precision* for all case studies, i.e. all of the actions in the extracted models are workflow-relevant (**RQ1**). Concerning the *recall* (**RQ2**), 82-100 percent of the actions described in the target workflows are covered by the extracted models. However, for Oxid we had to increase the maximum click depth to 4 in order to pass through the ordering steps in the shopping cart. With click depth 2, the recall was only 22 percent. The ProCrawl default configuration proved to be effective and for the other case studies also the default click depth of 2 provided good results. Actions that were not covered in the extracted models either required an input data configuration that was not provided in our case study, such as the author password for withdrawing a paper in OpenConf, or did not change the state of the SUT, such as downloading the invoice pdf in Oxid.

While the results are promising, the ability of a crawler in automatically exploring a web application in general depends on the complexity of the required input data and UI interactions; ProCrawl is no exception (cf. Section 5.4). More complex applications with a low crawlability [2] may require a manual specification of the input data and the recall may be lower.

Transition guard learning produced correct guard conditions that successfully resolved nondeterminism in the ABMs (**RQ3**). The runtime overhead was 8-19 percent for three of the case studies; for OrangeHRM, no guards were necessary and transition guard learning did not intervene.

In our case studies, ProCrawl produced reasonable accurate and concise models without requiring much configuration effort. Merging similar transitions in the ABM diagrams reduced the number of edges between 35-73 percent, which improved the readability of the models. The largest model (Oxid DG4) has 12 states and 95 transitions that were merged into 49 transitions, a size that can reasonably be validated manually.

The time required to configure ProCrawl depends on the familiarity of the user with the tool and the SUT. For our case study applications that were initially unknown to us, the configuration time was below 30 minutes each.

5.3 Behavior Models as Input for Test Generation

The release frequency in today's industrial practice of engineering web applications is very high. Therefore, testing needs to be highly efficient and automated.

Testing levels. The testing of (web) applications usually takes place on several levels [5]: *Unit tests* ensure that software components are functionally correct in isolation. The goal is to cover a large percentage of the component's source code, to increase the confidence in its correctness. *Integration tests* ensure the fulfillment of contractual obligations of a component and correct communication between coupled components. Here we are focusing on the top most level in this hierarchy where the complete system undergoes a cross module *system test*. On this level we are not concerned with fully covering the system's source code through the tests but rather in covering the steps of the workflow implemented by the system.

Model-based testing. With the aim of increasing test automation, model-based testing [37] 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,

TABLE 6
Using ProCrawl Behavior Models for Detecting Logical Changes (LC) and Structural UI Changes (SC)

Subject	Model	Scripts	Test Cases		SUT	# Failures			Execution Time
	SUT version	#	#	*min / avg / max length*	*version*	Scripts	LC	SC	(*fixed version*) *mm:ss*
OpenConf	DG2 *5.30*	31	4	4 / 51 / 74	5.30	0	0	0	03:27
					6.10	1	**1**	0	(03:55) | 17:43
Oxid	DG4 *4.8.5*	56	7	8 / 54.3 / 80	4.7.0	6	0	**1**	(57:05) | 50:13
					4.8.5	0	0	0	63:21
					4.8.7	0	0	0	58:19
OrangeHRM	DG2 *3.1.1*	14	4	2	2.7.1	1	**1**	0	(00:49) | 01:21
					3.1.1	0	0	0	00:55
					3.1.2	0	0	0	00:56
MantisBT	DG2 *1.2.17*	35	7	2 / 29.1 / 58	1.2.17	0	0	0	26:41
					1.3.0dev	4 | 19	**1**	**1**	(16:10) | 21:59

industrial application has shown positive effects of MBT on the development productivity (e.g., [8], [18]). However, wide-scale adoption suffers from the absence of explicit models that are complete enough to be executable by test frameworks.

ProCrawl can automatically extract such models from running web applications. Given that the extracted models correctly model the observed behavior (Section 5.3.2), they may serve as a "*gold standard*" oracle [7] for system regression testing [29] (Section 5.3.3); using a model-based test generator [9], one can generate an executable test suite (Section 5.3.1) that detects regressions which eliminate behavior from one revision of the application to the other.

While the general idea of extracting behavior models for testing web applications is not new (cf. [23], [27]), the models extracted by ProCrawl are particularly interesting with regards to former approaches due to the applied level of abstraction modeling workflows with multiple interacting users and including complex UI command sequences over multiple views.

5.3.1 Model-Based Test Case Generation

When using the ABMs for test generation, a model-based test generator traverses the given ABM according to a defined coverage criteria, such as transition coverage and returns the generated transition sequences as test cases. In ABMs each transition refers to an actor and a generated script, which makes the abstract transition sequences executable. Table 7 reports some statistics about the length of the scripts generated by ProCrawl. The overall avg. script length is 6.1 UI commands, with a minimum of 2 and a maximum of 15 commands.

For the test case generation we defined the accepting states of the ABMs and used a test generator [43] for state transition systems. Table 6 reports the results.

TABLE 7
Number of UI Commands in Generated Scripts

Script length	OpenConf	Oxid	OrangeHRM	MantisBT
min.	3	2	3	4
avg.	6.5	6.8	4.1	7.2
max.	15	9	5	15

OpenConf We selected state 08 (accepted/rejected) of the ABM_2 as accepting, i.e. all test cases end in state 08, resulting in four test cases with 51 script or 331.5 UI command executions on average.

Oxid We selected state 05 (shipped) and 10 (canceled) as accepting, resulting in seven test cases with 54.3 script or 369.2 UI command executions on average for the DG4 ABM_4.

OrangeHRM We selected state 03 (canceled), 04 (approved) and 05 (rejected) of the ABM_2 as accepting, resulting in four test cases with two script or 8.2 UI command executions on average.

MantisBT We selected state 05 (resolved/closed) of the ABM_2 as accepting, resulting in seven test cases with 29.1 script or 209.5 UI command executions on average.

Discussion. Based on the generated ABMs and scripts we could automatically generate executable test cases with minimal effort (**RQ4**). We just had to select the accepting ABM states for the test generator.

In [22], Marchetto and Tonella did an experiment on the fault exposing potential of 12K generated test cases with 2-12 UI commands each, based on two web applications. Considering their result that *longer interaction sequences have higher fault exposing potential*, the test cases generated from the ABMs are promising.

5.3.2 Model Over-Approximation

An over-approximating ABM allows traversal paths that are not feasible in the target application, which causes false positives when using the ABM for testing. Before using the generated test cases for regression testing, we therefore have to ensure they do not fail when executing them on the unmodified system.

Except for OrangeHRM, all the extracted ABMs contain cycles, i.e. the number of different paths through these ABMs is unlimited and exhaustive testing is impossible. Therefore, test generators have to do test selection. This means we cannot prove the absence of infeasible paths through testing. Nevertheless, we can significantly increase the confidence in the correctness of an extracted ABM by executing the test cases that were selected by the

test generator (cf. Section 5.3.1) on the version of the application from which we extracted the ABM (cf. Table 6). Failing test cases indicate infeasible paths. We executed the generated test cases using Firefox v33 with SELENIUM IDE 2.8.0.

OPENCONF The four test cases with overall 204 script executions completed successfully in 3:27 minutes.

OXID The seven test cases with overall 380 script executions completed successfully in 63:21 minutes. However, we had to reduce the execution speed to medium, otherwise some of the scripts failed.

ORANGEHRM The two test cases with overall eight script executions completed successfully in 0:55 minutes.

MANTISBT The seven test cases with overall 204 script executions completed successfully in 26:41 minutes.

Discussion. None of the generated test cases failed. While this does not prove the absence of infeasible paths, we have high confidence that the extracted ABMs correctly model the actual application behavior (**RQ5**). Since all generated test cases passed on the unmodified system, we can use them for regression testing.

5.3.3 Model-Based Regression Testing

As software evolves and components get added or modified, these may cause failures (called software regressions) in unchanged components. A common method to increase the confidence in the correctness of the modified system is *regression testing* [7]. Usually this involves using existing tests that have been successfully executed on the unmodified system (baseline) for *retesting* the modified system (delta build) to ensure that unchanged parts of the system are not adversely affected by the modifications.

By generating tests from the ABM of the baseline application and executing these tests on the delta build, we are able to detect

- *logical changes* to the underlying workflow, e.g., removing workflow steps or introducing additional mandatory steps.
- *structural changes* to the web UI that cause the generated scripts to fail, e.g., removing, replacing or changing attributes of UI elements that are used in the scripts to locate these elements.

We executed the test cases generated from the baseline ABMs on several delta builds (cf. Table 6):

OPENCONF The paper submission failed in v6.10, since no topic area was selected in the generated script. This is because in v5.30 the topic area was preselected whereas in v6.10 it was not (LC).

OXID Running the tests generated from the v4.8.5 DG4 ABM on v4.8.7, there were no failures; for v4.7.0 we had to manually update 6 scripts, since the text of a button in the shopping cart has been changed (SC).

ORANGEHRM There were no failures during test execution on v3.1.2. Running the test suite on v2.7.1 revealed that the leave request form has been changed; in v2.7.1 the number of hours has to be entered when requesting a single day of vacation (LC).

MANTISBT We had to manually update 19 out of 35 scripts, since the name of the issue selection checkbox has been changed in v1.3.0dev (SC). Running the updated tests resulted in four failing scripts; in v1.3.0dev it was not possible to unassign an issue by selecting `null` as handler (LC).

5.3.4 Summary and Discussion

Based on the extracted ABMs we could automatically generate test cases with minimal effort (**RQ4**). All of the generated tests could be successfully executed on the baseline applications. There were no infeasible paths generated from the extracted ABMs, which increased our confidence in the correctness of the models (**RQ5**).

By using the ABMs for model-based regression testing (**RQ6**), we detected structural changes on the UI, such as changed button labels, as well as several logical changes. After revising the changes, the models and scripts can be recreated by running PROCRAWL on the new version of the applications.

Furthermore, test models enable model-based test suite minimization with respect to a set of selected requirements [19].

5.4 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 five diverse, nontrivial web applications, and found that the resulting models represent the underlying processes in a reasonably accurate manner. However, the assumption underlying PROCRAWL is that most process behavior can be explored through a sequence of simple GUI 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. In future work we want to leverage existing test cases (Section 7).

We have chosen applications that support a multi-user workflow, and provide an HTML UI. For applications that are similar to the ones we studied, including web shops, submission systems, issue trackers and the like, we are confident that the PROCRAWL results will be similarly accurate as in our 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 and making the generated models, scripts and log files available (cf. Section 7), such that readers can compare them with the applications.

The final concern is *construct validity*—the adequacy of our measures for capturing dependent variables. As it comes to extracting behavior models from programs, comparing them to ground truth established through static analysis or testing is common practice; 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 understandable by humans; however, we found that the resulting models, 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 and processes.

6 Related Work

There exist many approaches for reverse engineering various properties of software components (cf. [32]). ProCrawl is a dynamic technique mining behavior models of web applications that generalize upon multiple runs (traces) of the application, which makes it applicable where source code is not accessible or amenable to static analysis. The resulting models capture the interaction of multiple users, as well as the interplay between input data and event sequencing affecting application behavior.

ProCrawl is related to dynamic techniques mining (extended) finite state machines, experimental techniques that systematically generate program runs to learn program behavior, techniques extracting models from GUI applications, and web crawling approaches.

Mining finite state machines. There are various approaches that mine FSMs capturing program behavior as sequencing of events (cf. [40]), many of them building upon the k-tails algorithm [6]. However, program behavior usually depends not only on the sequencing of events, but also on the provided input data, which led to approaches combining FSM inference with data rule inference. jMiner [20] mines parametric specifications capturing multi-object behavior of Java classes by preprocessing parametric execution traces and using an extension of k-tails to extract an FSM with data parameters. Pradel et al. [30] also mine parametric multi-object specifications (API usage protocols) from Java classes. While multi-object specifications capture method calls performed on multiple objects, ProCrawl captures UI interactions performed by multiple actors.

One pioneering work was developed by Lorenzoli et al. [21], combining k-tails with the Daikon [15] data invariant miner, inferring extended finite state machines. The inferred data constraints summarize the data for individual transitions. However, they do not capture the logical relationship between data and control and therefore may fail to resolve nondeterminism. While nondeterministic models may be acceptable for program comprehension, they are usually too imprecise for test case generation.

The transition guard learning approach implemented in ProCrawl is similar to the approach of Walkinshaw et al. [41], extracting global data rules that resolve nondeterminism via data classifier inference. Whereas they use classification to predict the next method to be called based on the current method and data input, ProCrawl builds classifiers predicting the target state of nondeterministic transitions based on the data trace to the transition. Furthermore, ProCrawl does not rely on a given set of training data (traces), but applies active learning, i.e. it systematically generates runs to extend the training set for building the classifier. In contrast to previous approaches to EFSM inference, ABMs inferred by ProCrawl include update functions.

Mining behavior from web applications. Techniques for mining FSMs have also been applied to the domain of web applications, most of them (e.g., [3], [24]) based on dynamic analysis and relying on a given set of execution traces (inductive analysis [47]), but also experimenting with web crawling to extend the set of available traces [4].

Francescomarino et al. [14] presents an approach for reverse engineering business processes exposed in web applications by inferring an FSM from execution traces and transforming it to BPMN. ReBPMN [35] also extracts process models in BPMN using a genetic algorithm operating on a given set of execution traces that are annotated with tags from a business domain ontology.

An overview of static and dynamic analysis approaches is given in [36].

Experimental analysis. The quality of models extracted by dynamic techniques closely depends on the choice of program runs (traces) these techniques generalize upon. Experimental approaches (cf. [47]) such as ProCrawl tackle this problem by systematically generating runs to explore program behavior; this allows ProCrawl to verify its hypotheses and explicitly control the exploration scope, which would otherwise be implicitly induced by a given set of traces. Such techniques have successfully been implemented for mining behavior models of Java classes [13], [46]. The underlying problem ProCrawl has to solve is an instance of online exploration of a directed multi-graph by repeatingly selecting an outgoing edge from the current vertex and traversing it, which is a fundamental problem in robotics and has been extensively studied for strongly connected graphs (cf. [16]). However, ABMs are usually not strongly connected, i.e. ProCrawl might need to reset the SUT to the initial state to continue graph exploration.

Extracting GUI models. ProCrawl mines program behavior by observing changes on the application's web UI, which is related to GUI Ripping developed by Memon et al. [25]; GUITAR [28] reverse engineers event-flow graphs (EFGs) of Java desktop, web and Android applications, which are used for model-based GUI testing. While in ABMs extracted by ProCrawl user actions are modeled as transitions between nodes that represent abstract GUI states of multiple users, in EFGs they are modeled as nodes with transitions representing the event flow.

Web crawling. To explore a web application, ProCrawl applies techniques similar to web crawling. Choudhary et al. [10] gives an overview on the state of the art.

A prominent representative and actively developed crawler for AJAX applications is CRAWLJAX [26], which automatically creates a state-flow graph (SFG) of the dynamic DOM states and the event-based transitions between them. SFGs depict the various navigational paths and UI states within an AJAX application. Although ABMs look similar to SFGs, the underlying abstraction in SFGs is much closer to the SUT's UI.

Limiting the state abstraction scope to a single DOM tree of a single user, limits the number of actions that can be detected by the crawler and often leads to a nondeterministic FSM, prohibiting effective model-based testing. In ABMs, a node represents an abstraction over multiple DOM trees (views) of multiple users and transitions refer to generated scripts encapsulating sequences of UI commands. Furthermore nondeterminism is effectively eliminated by learning transition guards over the input data.

Web testing. In our evaluation we apply the behavior models inferred by PROCRAWL for state-based web application testing, similar to [23] and [27]. However, due to the wider exploration scope and higher level of abstraction supporting multiple users and views, the test cases generated from PROCRAWL ABMs are more than pure UI tests and suitable for testing workflows. Garousi et al. [17] reports the state of the art in web testing.

7 CONCLUSION & FUTURE WORK

PROCRAWL is a fully automatic, though configurable, tool to mine workflow models from web applications. It implements an iterative approach of executing actions through the UI, observing changes to the UI, and enhancing the model. Its state abstraction is specific enough to capture essential workflow steps, yet sufficiently generic to be applicable to a diverse range of web applications. Through active learning, PROCRAWL increases model accuracy by inferring transition guard conditions from the input data.

As our evaluation on several real-world web applications shows, the models mined by PROCRAWL are adequate in size, accurate, cover all to almost all workflow-relevant actions, and are a suitable input to model-based test generation. The generated models, scripts, and log files can be downloaded from

https://www.st.cs.uni-saarland.de/models/procrawl

In our experience, the models mined by PROCRAWL provide an excellent starting point for manual refinement; as demonstrated in this paper, they can even be used without any human adaptation. With this, PROCRAWL lays the path for future model mining tools, extracting models that not only can be processed by computers, but that will be well amenable for human understanding, too.

Our future work will focus on the following topics:

Integrating existing tests. Many web applications come with existing unit and system tests. We are exploring means to integrate and adapt these tests into automated crawling, using their data for input provisioning, and their interaction flows for even better coverage.

Richer models. At this point, the base of our model is purely state-based. We are thinking about leveraging context-free and context-sensitive grammars, that would allow to express much more complex interactions and dependencies.

Alternative platforms. Besides web applications, the PROCRAWL techniques could just as well be applied on generic GUI-driven applications, providing model extraction and subsequent model-based testing on a wide range of platforms and programs.

ACKNOWLEDGMENTS

The authors thank Vitaly Kozyura, Sebastian Wieczorek and the anonymous reviewers for helpful comments on earlier revisions of this paper. The work presented herein has been partially funded by the German Federal Ministry of Education and Research (BMBF) under grant no. 01IC12S01 as well as by an European Research Council (ERC) Advanced Grant "SPECMATE – Specification Mining and Testing".

REFERENCES

[1] J.-R. Abrial, *The B-Book: Assigning Programs to Meanings*. New York, NY, USA: Cambridge Univ. Press, 1996.
[2] N. Alshahwan, M. Harman, A. Marchetto, R. Tiella, and P. Tonella, "Crawlability metrics for web applications," in *Proc. IEEE 5th Int. Conf. Softw. Testing, Verification Validation*, 2012, pp. 151–160.
[3] D. Amalfitano, A. Fasolino, and P. Tramontana, "Reverse engineering finite state machines from rich internet applications," in *Proc. 15th Working Conf. Reverse Eng.*, 2008, pp. 69–73.
[4] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "Rich internet application testing using execution trace data," in *Proc. 3rd Int. Conf. Softw. Testing, Verification, Validation Workshops*, 2010, pp. 274–283.
[5] P. Ammann and J. Offutt, *Introduction to Software Testing*. New York, NY, USA: Cambridge Univ. Press, 2008.
[6] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Trans. Comput.*, vol. C-21, no. 6, pp. 592–597, Jun. 1972.
[7] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Object Technology Series. Reading, MA, USA: Addison-Wesley, 1999.
[8] R. V. Binder, B. Legeard, and A. Kramer, "Model-based testing: Where does it stand?" *Queue*, vol. 13, no. 1, pp. 40:40–40:48, Dec. 2014.
[9] K. T. Cheng and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *Proc. 30th Int. Des. Autom. Conf.*, New York, NY, USA, 1993, pp. 86–91.
[10] S. Choudhary, M. E. Dincturk, S. M. Mirtaheri, A. Moosavi, G. von Bochmann, G.-V. Jourdan, and I. V. Onut, "Crawling rich internet applications: The state of the art," in *Proc. Conf. Center Adv. Stud. Collaborative Res.*, Riverton, NJ, USA, 2012, pp. 146–160.
[11] 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 *Proc. Int. Conf. Softw. Eng.*, New York, NY, USA, 2000, pp. 439–448.
[12] V. Dallmeier, M. Burger, T. Orth, and A. Zeller, "WebMate: A tool for testing web 2.0 applications," in *Proc. Workshop JavaScript Tools*, New York, NY, USA, 2012, pp. 11–15.
[13] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller, "Generating test cases for specification mining," in *Proc. 19th Int. Symp. Softw. Testing Anal.*, New York, NY, USA, 2010, pp. 85–96.
[14] C. Di Francescomarino, A. Marchetto, and P. Tonella, "Reverse engineering of business processes exposed as web applications," in *Proc. Eur. Conf. Softw. Maintenance Reeng.*, 2009, pp. 139–148.
[15] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 99–123, Feb. 2001.
[16] K.-T. Förster and R. Wattenhofer, "Directed graph exploration," in *Proc. 16th Int. Conf. Principles Distrib. Syst.*, 2012, pp. 151–165.
[17] V. Garousi, A. Mesbah, A. Betin-Can, and S. Mirshokraie, "A systematic mapping study of web application testing," *Inf. Softw. Technol.*, vol. 55, no. 8, pp. 1374–1396, Aug. 2013.

[18] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman, "Model-based quality assurance of protocol documentation: Tools and methodology," *Softw. Testing, Verification Rel.*, vol. 21, no. 1, pp. 55–71, Mar. 2011.
[19] B. Korel, L. H. Tahat, and B. Vaysburg, "Model based regression test reduction using dependence analysis," in *Proc. Int. Conf. Softw. Maintenance*, 2002, pp. 214–223.
[20] C. Lee, F. Chen, and G. Roşu, "Mining parametric specifications," in *Proc. 33rd Int. Conf. Softw. Eng.*, New York, NY, USA, 2011, pp. 591–600.
[21] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proc. 30th Int. Conf. Softw. Eng.*, New York, NY, USA, 2008, pp. 501–510.
[22] A. Marchetto and P. Tonella, "Search-based testing of Ajax web applications," in *Proc. 1st Int. Symp. Search Based Softw. Eng.*, 2009, pp. 3–12.
[23] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of Ajax web applications," in *Proc. Int. Conf. Softw. Testing, Verification, Validation*, 2008, pp. 121–130.
[24] A. Marchetto, P. Tonella, and F. Ricca, "ReAjax: A reverse engineering tool for Ajax web applications," *IET Softw.*, vol. 6, no. 1, pp. 33–49, Feb. 2012.
[25] A. M. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *Proc. 10th Working Conf. Reverse Eng.*, 2003, pp. 260–269.
[26] A. Mesbah, A. Van Deursen, and S. Lenselink, "Crawling Ajax-based web applications through dynamic analysis of user interface state changes," *ACM Trans. Web*, vol. 6, no. 1, pp. 1–30, Mar. 2012.
[27] A. Mesbah, A. Van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 35–53, Jan. 2012.
[28] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: An innovative tool for automated testing of GUI-driven software," *Automated Softw. Eng.*, vol. 21, no. 1, pp. 65–105, Mar. 2014.
[29] M. Pezzè and M. Young, *Software Testing and Analysis: Process, Principles and Techniques*. New York, NY, USA: Wiley, 2007.
[30] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking API protocol conformance with mined multi-object specifications," in *Proc. 34th Int. Conf. Softw. Eng.*, Piscataway, NJ, USA, 2012, pp. 925–935.
[31] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann, 1993.
[32] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Trans. Softw. Eng.*, vol. 39, no. 5, pp. 613–637, May 2013.
[33] M. Schur, A. Roth, and A. Zeller, "Mining behavior models from enterprise web applications," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, 2013, pp. 422–432.
[34] M. Schur, A. Roth, and A. Zeller, "ProCrawl: Mining test models from multi-user web applications," in *Proc. Int. Symp. Softw. Testing Anal.*, San Jose, CA, USA, Jul. 21–26, 2014, pp. 413–416.
[35] A. Tomasi, A. Marchetto, C. D. Francescomarino, and A. Susi, "reBPMN: Recovering and reducing Business Processes," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance*, 2012, pp. 666–669.
[36] P. Tramontana, D. Amalfitano, A. R. Fasolino, and N. Federico, "Reverse engineering techniques: From web applications to rich internet applications," in *Proc. 15th IEEE Int. Symp. Web Syst. Evolution*, 2013, pp. 83–86.
[37] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann, 2007.
[38] A. Valmari, "The state explosion problem," in *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*. London, U.K.: Springer-Verlag, 1998, pp. 429–528.
[39] V. N. Vapnik, "An overview of statistical learning theory," *Trans. Neur. Netw.*, vol. 10, no. 5, pp. 988–999, Sept. 1999.
[40] N. Walkinshaw, B. Lambeau, C. Damas, K. Bogdanov, and P. Dupont, "Stamina: A competition to encourage the development and assessment of software model inference techniques," *Empirical Softw. Eng.*, vol. 18, no. 4, pp. 791–824, Aug. 2013.
[41] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions," in *Proc. 20th Working Conf. Reverse Eng.*, 2013, pp. 301–310.
[42] E. J. Weyuker, "On testing non-testable programs," *Comput. J.*, vol. 25, no. 4, pp. 465–470, 1982.
[43] S. Wieczorek, V. Kozyura, A. Roth, M. Leuschel, J. Bendisposto, D. Plagge, and I. Schieferdecker, "Applying model checking to generate model-based integration tests from choreography models," in *Proc. 21st IFIP WG 6.1 Int. Conf. Testing Softw. Commun. Syst.*, 2009, pp. 179–194.
[44] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. New York, NY, USA: Springer, 2012.
[45] Q. Xie and A. Memon, "Using a pilot study to derive a GUI model for automated testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 2, pp. 1–35, 2008.
[46] T. Xie and D. Notkin, "Automatic extraction of object-oriented observer abstractions from unit-test executions," in *Proc. Int. Conf. Formal Eng. Methods*, 2004, pp. 290–305.
[47] A. Zeller, "Program analysis: A hierarchy," in *Proc. ICSE Workshop Dyn. Anal.*, 2003, pp. 6–9.

Matthias Schur is a software engineer in the cloud platform team at SAP SE, Germany, and a PhD candidate at Saarland University-Chair for Software Engineering. His research concerns automated analysis and testing of enterprise web applications.

Andreas Roth is a development architect at SAP SE, Germany, and a visting professor of practice at Newcastle University, United Kingdom. He works on tools for the development of enterprise web applications.

Andreas Zeller is a full professor for Software Engineering at Saarland University in Saarbrücken, Germany. His research concerns the analysis of complex software systems, their security properties, and their development process. In 2010, Zeller was inducted as Fellow of the ACM for his contributions to automated debugging and mining software archives. He is a member of the IEEE Computer Society.