Source Code Independent Reverse Engineering of Dynamic Web Sites

Technical Report B-04-10

Dirk Draheim, Christof Lutteroth
Institute of Computer Science
Freie Universität Berlin
Takustr.9, 14195 Berlin, Germany
draheim@acm.org lutterot@inf.fu-berlin.de

Gerald Weber
Department of Computer Science
The University of Auckland
38 Princes Street, Auckland 1020, New Zealand
g.weber@cs.auckland.ac.nz

June 2004

Abstract

This paper describes source code independent reverse engineering of dynamic web sites. The tool Revangie builds a form-oriented analysis model solely from the usage of a web application. The recovered models can be, for example, exploited for the purpose of requirements engineering and load test development. Revangie can explore a given web application fully automatically or can passively record its usages. The collected data, i.e., data about screens, server-side programs, and system responsiveness, are analyzed in order to build a user interface model. The paper presents several adequate screen classifications, which are utilized to yield significant models.

Contents

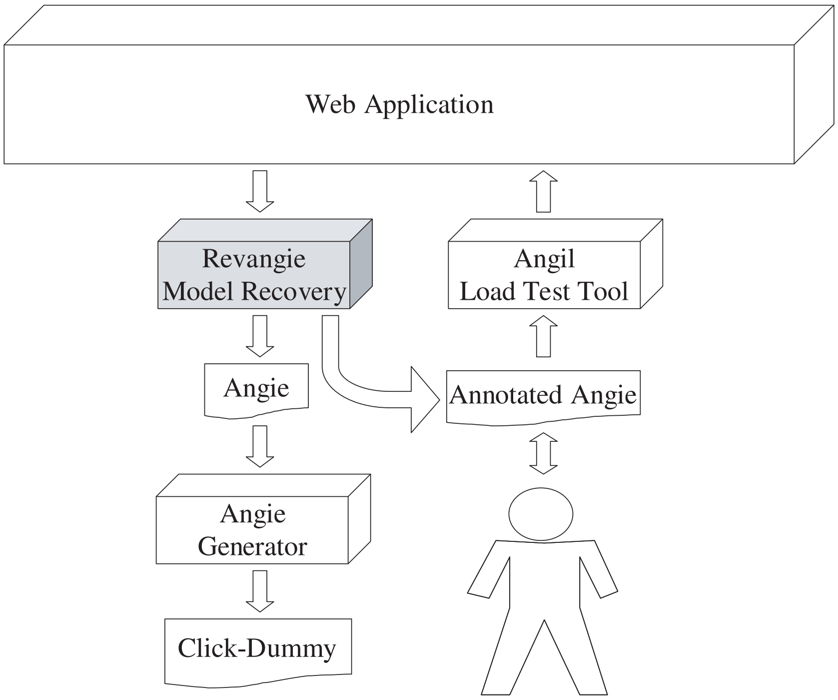


Figure 1: The Angie language related tool suite.

1 Introduction

This paper presents Revangie – a tool that is able to recover the model of a dynamic web interface of a web site without looking at the source code. We chose to recover models independent of source code because of the manifold of languages, platforms and architectures a dynamic web site can be implemented in. Tools that perform source code dependent reverse engineering are usually restricted to a certain language, platform etc., whereby Revangie is independent of all those. Furthermore, the source code independent approach is a straightforward one, because it is much easier to analyze HTML code than the generating code, whereas it is an essential claim that the analysis of the generated HTML is sufficient to recover sophisticated models. It is not only easier to analyze HTML code than the generating code, but more convenient, too, because the HTML code can be explored through the single point of access of an HTTP port, whereas the generating code can have a complex deployment structure. In the case that the source code is inaccessible, analysis must be source code independent anyway, as it is the case in typical product benchmarking efforts.

We explain the motivation for Revangie by describing its role in the Angie tool suite. Please consider Fig. 1. Revangie is used to recover form-oriented models from a dynamic web site. The textual description of this model in the language Angie [8] can be subsequently used for forward engineering of click dummies, as they are conveniently used for requirements engineering, or customizable systems, which can help in migrating to model-driven architecture. In addition to this, Revangie can collect data about user behavior that can be used for load testing. It is further work to provide the load test tool Angil that simulates real users on the basis of an annotated version of the Angie language.

In Sect. 2, we provide a brief overview of form-oriented analysis [7, 5]. In Sect. 3, we describe the different modes of operation of Revangie. Section 4 explores the screen classification problem. The paper finishes with a discussion of related work, further directions and a conclusion in Sects. 5,6 and 7.

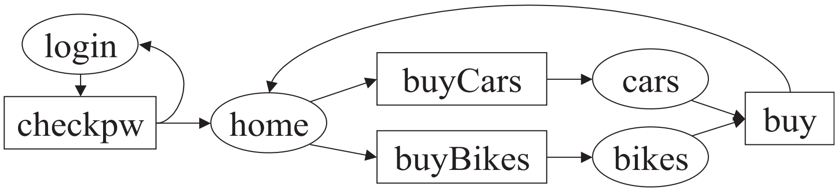


Figure 2: Form chart example.

2 Using the Form-Oriented User Interface Model

In order to perform coherent analysis, we need a model that describes the user interface of web applications adequately. The form-oriented user interface model uses typed, bipartite state machines, in which one set of states denotes client pages and the other set server actions that generate the pages. These graphs contain all the information of page diagrams, but in addition to this, they model the relationship between server-side actions and pages. It is obvious, which form or link on a page uses which server action and which server action generates which page. Note that the pages in the model do not represent individual pages as seen by the user, but classes of pages of which many instances may be generated by different server actions. We call the possible instances of a page *screens*.

An important property of the model is its type system: actions as well as pages have a signature which specifies a type for the data accepted by an action and a type for the data displayed on the screens of a page. These signatures help us in determining which forms and links invoke which actions. Furthermore, they help us in identifying similar actions and similar screens. An action is uniquely identified by the program that is invoked, which is referenced in the URI, plus the set of labels with which parameters are passed to it through the CGI. Both kinds of information are contained in forms and links that use CGI. Therefore, an invocation of a server action can be thought of as a method call, the URI of the CGI program being the method name and the set of labels being the method signature. Following this metaphor, using a CGI program with different sets of labels corresponds to method overloading.

For textual representation of such models, form-oriented analysis proposes the language Angie [8]. Small models can be visualized conveniently in so called *form charts*, in which pages are represented as bubbles and actions as rectangles.

The web application depicted in the form chart in Fig. 2 starts at client page "login" which should contain a form for submitting a user name and password to server action "checkpw". If the submitted login information is invalid, the user is shown another instance of the "login" page, otherwise the user is forwarded to page "home". Here, options "buyCars" and "buyBikes" can be chosen and the respective pages subsume all the possible screens showing lists of cars and bikes, respectively. According to the form chart, both pages use the same action "buy" to process submitted data. This means that the same CGI program is used with the same parameter labels, so the buying of a car is distinguished from the buying of a bike merely by the actual parameters.

In the context of reverse engineering, we can, as mentioned, identify the signature of an action in HTML code and therefore the set of actions in our model could be accurately determined by an exhaustive analysis of a system's HTML screens. The signature of pages, however, cannot generally be recovered from the HTML code. The difference between pages may be a purely semantic one like in our example, where the pages "cars" and "bikes" show lists that may be structurally identical. In order to distinguish pages precisely, we would have to look into the CGI programs that generate them as, for example, it is done by the tool JSpick [4]. Consequently, a main problem of client-side reverse engineering without the source code of the CGI programs is the classification of screens into pages.

3 The Reverse Engineering Process

Revangie can work in different modes: the crawl mode, which works on the client side, and the snoop mode, which works at any point in the communication line between client and server. When Revangie is started, it must be told which mode it should use.

3.1 The Crawl Mode

The crawl mode works like an automated web browser: It uses an HTTP client to request pages, submit values and analyze the trace of submitted values and visited pages. It starts at a specified URL, retrieves the corresponding HTML screen and scans it for links, forms and some other tags. As we have discussed in section 2, we can infer an action for each link and form. As we will see in section 4, each screen, in turn, must be classified as being an instance of a certain page. In this way, we can obtain information about which pages and actions are in the model, which action can be invoked from which page and which page is generated by which actions.

3.1.1 Problems of Fully Automatic System Exploration

It would be naive to think that the analysis model of every non-trivial web application could be reconstructed automatically. The problematic point is the data submitted on a page, which usually has to suffice certain constraints and therefore, cannot be generated generically. Sometimes, this data depends on the context or its history, e.g., email addresses, credit card numbers or passwords. But still, there are web applications for which the crawl mode, which generates input data randomly, can yield satisfactory results. Think, for example, of a search engine. It is conceivable that the range of web-applications feasible for the crawler can be widened by the use of simple heuristics which, for example, exploit the fact that certain field labels, e.g., "surname" or "email", correspond to well-known data types. Note that we must not generate data for the hidden parameters of forms and links. Here, we have to use the values provided by the web application in order to guarantee consistent operation.

We cannot simply use a breadth-first or depth-first search of the bipartite graph that we are constructing because most of the more sophisticated web applications keep track of sessions, i.e., the history of action invocations triggered by a single user. The permitted protocol of a session is given by the sequence of screens as provided by the web server, and any deviation might cause an error. Most people know that using the back- and reload-buttons of a browser in a form-oriented system can cause errors due to the expiry of a page.

An important question for the crawl mode is how the choice for the next action in the automated browser is made. Of course, we want to make the choice that is most likely to help us in the analysis, i.e., that leads us to the discovery of a maximum of new pages and actions. But we cannot predict which choice this will be, so we have to use heuristics like a usage counter for each form and link. The "least used" heuristic says that the form or link with the minimum usage counter is a good choice.

3.1.2 The Crawling Algorithm

Putting it all together, this yields Algorithm 1 as the operational semantics of the crawl mode. Procedure crawlAll simulates many single-user sessions successively by invoking procedure crawl on page *entry* as long as a significant extension of the reconstructed model takes place. *entry* is an artificial page which contains forms for all the actions that may serve as entry points into the web application. Although page *entry* does not appear in the actual web application, it is represented internally like any other page, only that it is marked in the model.

In order to stop exploration when no new parts of the web application are discovered any more, we calculate the value α as a ratio of the number of iterations done in crawlAll to the number of actions found in *model*, the global variable containing the reconstructed model. This gives us the average number of actions discovered in a single simulated session; and once α drops below a threshold Θ, we assume that the application has been explored sufficiently. We use the number of actions and not the number of actions and pages because there is – in contrast to screens – a limited number of actions in a web application, usually proportional to the number of conceptual pages. The number of pages in the model depends on the way we classify screens and may grow enormously, while the number of actions is well-defined. Think, for example, of a search engine: the search action may lead to innumerable screens that may be hard to classify.

Algorithm 1 Crawl Mode

1: **procedure** CRAWLALL
2:     **while** $\alpha < \Theta$ **do**
3:         crawl($entry$)
4:     **end while**
5: **end procedure**

6: **procedure** CRAWL(Page p)
7:     **while** $|p.\text{forms}| > 0 \wedge \alpha < \Theta$ **do**
8:         **for all** Form $f \in p.\text{forms}$ **do**
9:             $model \leftarrow model \cup \{(p, \text{inferAction}(f))\}$
10:         **end for**
11:         $f \in p.\text{forms} \wedge f.\text{usages} = \min_{\substack{f' \in p.\text{forms} \\ \wedge \text{feasible}(f')}} f'.\text{usages}$
12:         **if** $f = null$ **then**
13:             break
14:         **end if**
15:         $f.\text{usages} \leftarrow f.\text{usages}+1$
16:         Action $a \leftarrow$inferAction(f)
17:         Set $data \leftarrow \emptyset$
18:         **for all** $field \in f.\text{fields}$ **do**
19:             $value \leftarrow$generateValue($field$)
20:             $data \leftarrow data \cup \{(field.\text{label}, value)\}$
21:         **end for**
22:         $s \leftarrow$invoke($a, data$)
23:         $p' \leftarrow$classify(s)
24:         $p'.\text{screens} \leftarrow p'.\text{screens} \cup \{s\}$
25:         $model \leftarrow model \cup \{(a, p')\}$
26:         $p \leftarrow p'$
27:     **end while**
28: **end procedure**

Procedure crawl simulates a single session which runs into a terminal set of states, i.e., a set of pages and actions that cannot be left. The existence of terminal sets of states in a web application is the reason that simulation of a single session is insufficient; a single session can only explore one terminal set. We try to cover all the terminal sets by simulating multiple sessions.

In the main loop of procedure crawl, we iterate over page-action-page transitions, starting on the page p given as argument. In the condition of the while-loop in line 7, we check if the page offers any possibility of interaction at all; if not, crawl terminates. An HTML link can be represented as a special case of an HTML form; therefore, the class that represents a link is internally a subclass of class Form and if p contains no forms it does not contain links either. The other while-condition, $\alpha < \Theta$, is similar to the one in procedure crawlAll and will be discussed later on. Next, we infer the signature of an action for each of the forms on page p and add a corresponding page-action transition to *model*. We have seen in section 2 that the signature of an action can be inferred from any form; *model* represents the reconstructed model as a set of transitions between actions and pages.

In line 11, we choose one form f out of all feasible ones that has a minimal usage counter, thereby trying to maximize the likelihood of efficient exploration. A form is feasible if its action does not lead us out of the application, i.e., onto other web sites. If there is no feasible form, that is if $f = null$, the while-loop is left and crawl terminates. The actions of non-feasible forms in the model are later on connected to another artificial page named *exit*. After incrementing the usage-counter of the chosen form, we determine which action is invoked by f. In lines 17-21, we generate a label-value pair for each field of f, so that we can submit this data and receive a new screen s. We classify s into a page p', which can be an existing page or a new one, and add s to its screen set. We add the reconstructed action-page transition (a, p') to *model* and prepare the next iteration that uses p' as starting point.

The second condition of the while-loop, $\alpha < \Theta$, is there to ensure that crawl terminates when the terminal set of states it operates in seems to be sufficiently explored. Hereby, α is a value that drops below a threshold Θ when crawl does not significantly extend the model any more, e.g.,

$$\alpha := \frac{\text{number of new actions}}{\sqrt{\text{number of while iterations}}}.$$

Similar to procedure crawlAll, α is the ratio of the number of new actions found in the session to a term that grows with the number of iterations. Here, we use the square root of the number of iterations done in the while-loop because we iterate over the edges of the model and the number of states in a dense graph grows typically like the square root of its number of edges. In an application with many forms, this prevents crawl from terminating too early.

3.1.3 The Quality of Crawling

The quality of the constructed model depends heavily on functions generateValue and classify. It depends on generateValue how effectively user input is simulated and therefore how well the actual set of pages and actions in the web application is covered; classify determines if the granularity of the reconstructed model during exploration is adequate. This, in turn, also influences the efficacy of the crawling process because it affects the heuristic for the selection of the next form to use: A classification that is too fine leads to too much pages and thus too much forms in the model; most usage counters will be zero for a long time, so the least-used heuristic does not help much. A classification that is too coarse leads to too few forms and most usage counters will reach a high value very soon. In both the extreme cases, the least-used heuristic is not much better than random choice.

Classification of pages is not only, like depicted in the pseudo code, possible online, i.e., during the process of exploration. We can also perform a new classification offline, i.e., in a separate stage after the process of exploration. Therefor, we would record the exploration history as a set of linear lists of alternating screens and actions. Then, the classification algorithm could operate on the whole set of actions and screens a-posteriori and consequently use a wider range of analysis techniques.

3.2 The Snoop Mode

The snoop mode collects data of actual sessions of a web application in order to analyze it and reconstruct a model afterwards. It can either monitor the HTML communication of one or more users by taking the role of a proxy server, or monitor the communication of all users by taking the role of a facade to the web server. In the first case, the user has to configure his web browser accordingly, in the latter case, the web server has to be reconfigured.

This mode allows us to collect realistic session data of one or more users. In the case of many users, it can be utilized to determine certain user model parameters statistically. An analysis model with this additional data, like distributions for the turnaround time, navigation probabilities or sample input, is ideally fit for performing fully-automated realistic tests, especially load-testing.

3.3 The Guide Mode

The guide mode tries to combine the advantages of crawl and single-user snoop mode: automation and the possibility to enter form data manually. The algorithm is like in the crawl mode, but at certain points, the user is asked to choose the form to use next and enter appropriate values. The tool can ask for user input at every step, or only at steps that are hard to handle automatically, like forms with input fields.

Besides for the selection of a form and the generation of input data, user input can also be useful for screen classification. If classification seems ambiguous according to some measure, the user can select to which page a screen should be added. If automatic classification produced unsatisfactory results, screens can be reclassified by the user.

The guide mode uses an ordinary web browser for interaction and offers additional controls for computer aided reverse engineering. The tool acts as a proxy for the web browser and displays session histories, usage counters for forms, screen classifications and a form chart of the current model. It also suggests automatically generated data for input fields, thus making reverse engineering faster.

4 The Screen Classification Problem

We need to classify each received screen to a page in order to build our model properly. As depicted in Sect. 2, a page is characterized by the signatures of its targets, i.e., the actions that can be invoked on it by the user, and the signature of the data presented on its screens. The signature of an action is the URL of the CGI program and the set of labels used for transmitting values, and since the code of the HTML screens contains all the information about the signatures of the targets of corresponding pages, we can infer at least a partial key of the page for each screen. But there is a wide variety of possibilities for screen classification. If, for example, a target's URL has the suffix `.html` and no parameters, we can conclude that it leads to a static page of which all screens have to be equal by string equality. On the other hand, if a target action has parameters, it is likely to generate screens with varying content that require more sophisticated techniques to classify.

4.1 The Lattice of Classifications

Let us first look at the formal basis of classification. Formally, a classification is an equivalence relation, i.e., a reflexive, transitive and symmetric binary relation which partitions a set into disjoint subsets called equivalence classes. We denote the set of all equivalence relations over a given set S by EQ_S.

We call a classification resp. equivalence relation $A \in EQ_S$ a refinement of equivalence relation $B \in EQ_S$, denoted by $A \sqsubseteq B$, *iff*

$$\forall x, y \in S : x \sim_A y \Rightarrow x \sim_B y$$

The above definition of refinement is natural because a refining classification is conservative with respect to the refined classification, i.e., a refinement only further subdivides the classes of a given classification. This is also neatly expressed by the fact that the defined refinement is equal to the subset relationship between equivalence relations, i.e., the following holds

$$A \sqsubseteq B \textit{ iff } A \subseteq B$$

For example, given a set of screens $S = \{a, b, c\}$, the Hasse diagram in Fig. 3 visualizes the partial order $(EQ_S, \sqsubseteq)$ of equivalence relations over S.

Similarly we can define true refinement $\sqsubset$, coarsening $\sqsupseteq$, and true coarsening $\sqsupset$ by:

$$A \sqsubset B \textit{ iff } A \subset B,$$

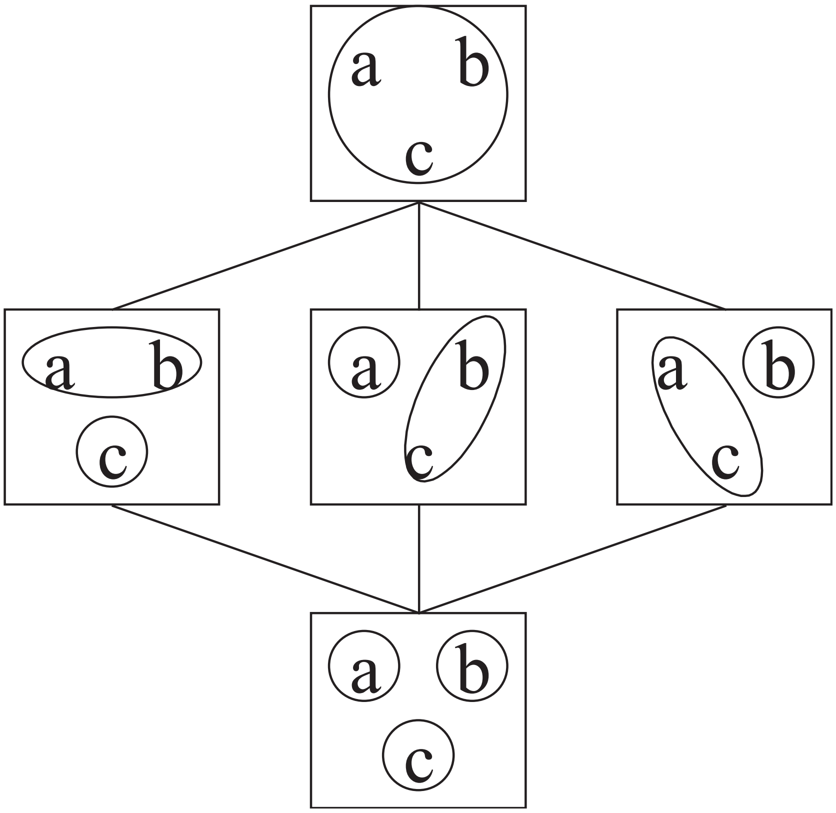


Figure 3: Cpo of equivalence relations.

$$A \sqsupseteq B \textit{ iff } A \supseteq B,$$
$$A \sqsupset B \textit{ iff } A \supset B.$$

We now define conjunction $\wedge$ and disjunction $\vee$ of classifications. A classification identifies elements of the set for which it is defined. Informally, a conjunction $A \wedge B$ identifies those elements that are identified by both A and B, whereas a disjunction $A \vee B$ identifies those elements that can be identified by means of A or B. Conjunction is defined by set intersection on equivalence relations; however, disjunction cannot be defined analogously just by set union, but must be defined as transitive closure of set union.

$$\begin{aligned} A \wedge B &=_{def} \quad A \cap B \\ A \vee B &=_{def} \quad \text{transitive-closure}(A \cup B) \end{aligned}$$

The partial order EQ_S together with conjunction and disjunction, i.e., $(EQ_S, \wedge, \vee)$, form a complete lattice.

A notion of classification assigns to each set a classification of this set, i.e., an equivalence relation on this set. Typically, only sets that adhere to certain criteria are considered – we can formalize this by considering subsets S of a given base set B only. The given context criteria can be exploited to argue about the refinement relation between notions of classifications defined below. All this means, that a notion of classification is a dependent product with respect to a family of sets and equivalence relations over these sets.

$$C = (C_S : EQ_S)_{S \subseteq B}$$

We now define refinement, disjunction and conjunction for notions of classification:

$$(C_S)_{S \subseteq B} \sqsubseteq (D_S)_{S \subseteq B} \quad \textit{iff} \quad \forall S \subseteq B . C_S \sqsubseteq D_S,$$

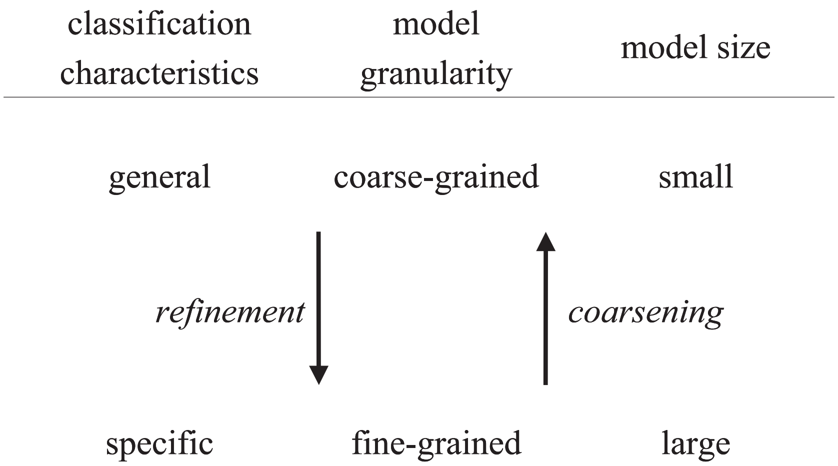


Figure 4: Terminology for classification.

$$
\begin{aligned}
(C_S)_{S\subseteq B} \wedge (D_S)_{S\subseteq B} \quad &=_{def} \quad (C_S \wedge D_S)_{S\subseteq B},\\
(C_S)_{S\subseteq B} \vee (D_S)_{S\subseteq B} \quad &=_{def} \quad (C_S \vee D_S)_{S\subseteq B}.
\end{aligned}
$$

Also the set NoC_B of notions of classification with respect to a base set B together with disjunction and conjunction, i.e., $(NoC_B, \wedge, \vee)$, form a complete lattice.

We now define the concept $SNoC_B$ of *standard notions of classification* with respect to base set B. Informally, a standard notion of classification is conservative with respect to set operations on the underlying set of the equivalence relations.

$$C \in SNoC_B \quad \textit{iff} \quad \forall S,T \subseteq B. \forall x,y \in S \cap T. x \sim_{C_S} y \Leftrightarrow x \sim_{C_T} y$$

For standard notions of classification, refinement can be defined more directly as the refinement on the base set classification because the following holds:

$$\forall S \subseteq B. C_S \sqsubseteq D_S \quad \Leftrightarrow \quad C_B \sqsubseteq D_B.$$

In the sequel, we use the terms *notion of classification* and *classification* synonymously. The terminology for classification used in our paper is defined in Fig. 4. It shows the effect of refinement and coarsening on classification characteristics, model granularity, and model size.

4.2 Possible Screen Classifications

Now, let us consider different notions of screen classification:

Trivial identity is the coarsest possible equivalence relation. All screens are equivalent. It is practically unimportant, but forms the top ($\top$) element of our lattice.

Screen identity is the finest possible equivalence relation for screen classification and consequently forms the bottom ($\perp$) element. Each screen that is received by a web client gets its own page, even when two screens have the same HTML code.

Textual identity groups screens with the same HTML code into the same page.

Source identity groups screens into the same page that were generated by the same action.

Targets identity groups screens with identical *targets signature*, i.e., the same set of signatures of the server actions targeted by a screen. Targets identity can be coarsened to **form targets identity** by excluding the signatures of links from the targets signature, or orthogonally, coarsened to **internal targets identity** by excluding forms and links that target external actions, like links to other web sites.

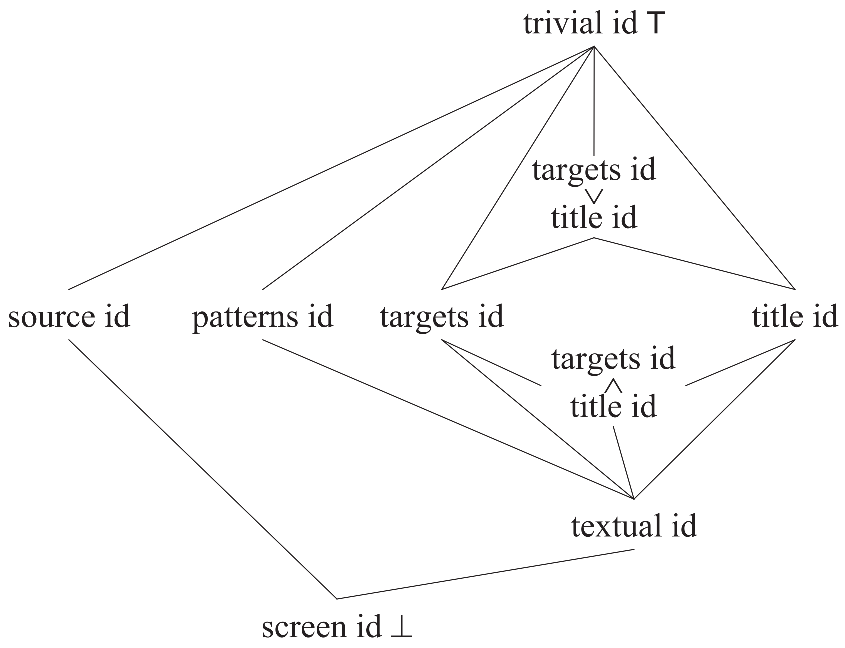


Figure 5: Lattice of notions of screen classification.

Title identity groups screens with identical HTML titles.

Pattern identity groups screens that match a user-defined pattern. This may be a textual pattern, a purely syntactical pattern or a mixture of both; regular or, at most, context-free patterns are usually sufficient.

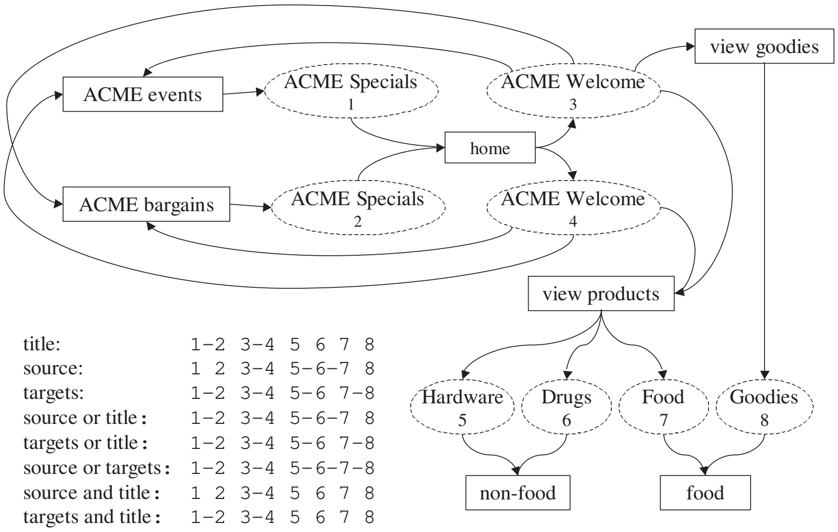
Similarity or dissimilarity of screens according to some textual or structural distance metric can be used to cluster them into pages (e.g., [17]). A similarity measure suitable for clustering can also be created by combination of multiple similarity indicators, like any of the discussed ones, which can be combined, for example, by a weighted sum.

The relationship between the different notions of classifications is illustrated in Fig. 5. Note that Fig. 5 is not a complete visualization of the lattice of notions of screen classification because only exemplary disjunctions resp. conjunctions are included.

By combining and configuring classification techniques appropriately, it is possible to reconstruct an expressive model with adequate page granularity. For a maximum of flexibility, Revangie offers a system of configurable plug-ins for screen classification, with a generic default plug-in. Note that it depends on the intended usage of a model which model granularity is adequate, i.e., a given model granularity must be judged with respect to an external criterion. For example, targets identity can be considered to fit best in load testing scenarios, whereas textual identity is the classification of choice for static HTML pages. Targets identity is an important basic classification because it yields the smallest possible form-oriented analysis models without enabling conditions for targets, i.e., it yields a model where all the screens of a page offer the same server actions for interaction. We discuss this topic further in Sect. 4.6.

Our classification apparatus is especially expressive because of the notions of conjunction and disjunction. Potential candidates for conjunction are, for example, targets identity and title, or targets identity and pattern identity. Please consider the classifications resulting from conjunction and disjunction in the ACME webshop example of Fig. 6[1]. The example, with its many different

[1] The figure shows a modified form chart diagram where single, completely unclassified screens are drawn instead of client pages. Therefore, dashed lines are used instead of solid ones for the bubbles.



title:	1−2 3−4 5 6 7 8
source:	1 2 3−4 5−6−7 8
targets:	1−2 3−4 5−6 7−8
source or title :	1−2 3−4 5−6−7 8
targets or title :	1−2 3−4 5−6 7−8
source or targets :	1−2 3−4 5−6−7−8
source and title :	1 2 3−4 5 6 7 8
targets and title :	1−2 3−4 5 6 7 8
source and targets :	1 2 3−4 5−6 7 8

Figure 6: Example conjoined and disjoined classifications.

classifications, aims to clarify that there is no single notion of correct model granularity. All the classifications correctly group screens 3 and 4, but not all separate 1 and 2 although they are semantically different. All the screens 5 to 8 display different kinds of products, so it would be thinkable to separate them, like title identity does. It might also be more appropriate to classify them according to their targets identity, so that only food and non-food products are distinguished.

4.3 Classification and Clustering

Every classification can be reformulated as a clustering according to some metric δ, with δ being a binary function on screens that evaluates to a positive real number called distance. For purely logic predicates like the identity of targets or titles, corresponding metrics would produce a distance of zero iff the predicate is true and some other fixed value if not. If the metric has a range that encompasses more than two distance values, as is the case for dissimilarity measures, i.e., if the answer of δ whether two screens belong to the same page is fuzzy, we need to use a clustering method. Such a method partitions a set of objects – in our case, the screens – into clusters which correspond to the equivalence classes of an equivalence relation – the pages, in our case. A large variety of such methods exists; see, for example, [12].

Most clustering methods allow us to choose the number of clusters, which means in our case, the granularity of the model. This allows us also to let the user set a preferred model size and calculate a model that fits that size best. But there are also useful heuristics for determining an optimal model size: it is reasonable to assume that the fan-out of an action has a small maximum, i.e., that an action generates only a few conceptually different pages, e.g., a page that displays and requests some information and a page that shows some error message if its input was invalid. If the fan-out of an action exceeds the maximum, we can assume that the clustering of its generated screens is too fine and coarsen the clustering, e.g., by merging clusters. If we assume that there is some typical fan-out value for an action, e.g., that it generates either a proper result or an error, we can also steer the clustering in a way that this value is approximated best. In the same way, we can also assume that a page has a limited fan-in, i.e., a limit on the actions that generate it. Remember that a link is represented by an action, not a page, so that if a link is on many screens of a web application, the corresponding action has a high fan-in, but not the page. That two different actions generate the same conceptual page is rare but occurs; so a maximum on the fan-in of a page can be used to steer the clustering, too.

4.4 Statistical Testing of Refinements

We introduce statistical methods in order to refine an equivalence relation on screens. After running Revangie in the snoop mode, we have a set S of screens. Furthermore, we may find a certain criterion suitable for distinguishing conceptually different pages. A good criterion is, for example, the target usage frequency distribution of a page, i.e., the number of screens for each target in which the respective target was invoked by a user. It is a natural assumption that pages with different usage patterns are conceptually different, and therefore, we look for classifications that partition S into pages with different usage patterns well.

We might want to know, for example, if classification by screen titles helps to distinguish pages that are used differently. In terms of statistics, every screen has random variables *target* and *title*, with *target* being the action that was invoked from that screen. In order to determine if *title* helps us to distinguish between different usages, we test if *target* depends on *title*. If so, we refine our classification by intersecting with title identity. A standard method used for testing dependency of random variables is Pearson's chi-square [13].

4.5 Clustering with Significant Refinements

We introduce a clustering method that uses Pearson's chi-square test as described in Sect. 4.4. Algorithm 2 works the following way: procedure cluster gets as input a classification, i.e., an equivalence relation, on the set S of all screens. It iterates over different other equivalence relations B on S, like target identity or title identity, and conjoins the current classification A with B using operator $\wedge_*$.

Algorithm 2 Statistical clustering of screens.

1: **procedure** CLUSTER(EQ_S A)
2:     **for all** $B \in \{$targets id, title id, ...$\}$ **do**
3:         $A \leftarrow A \wedge_* B$
4:     **end for**
5: **end procedure**

6: **function** $\wedge_*(EQ_S\ A,\ EQ_S\ B)$
7:     Set $P \leftarrow S/(A \wedge B) - S/A$
8:     **for all** $p_1, p_2 \in P, p_1 \neq p_2$ **do**
9:         **if** $\neg\chi^2(p_1, p_2)^*$ **then**
10:             $P \leftarrow (P - \{p_1, p_2\}) \cup \{p_1 \cup p_2\}$
11:         **end if**
12:     **end for**
13:     $S/A' = (S/(A \wedge B) \cap S/A) \cup P$
14:     return A'
15: **end function**

The conjunction $\wedge_*$ does not work like ordinary conjunction as defined in Sect. 4. Before splitting up a page cluster in A, it tests if the split leads to conceptually different subclusters. That is, it may preserve certain parts of A that the ordinary conjunction would take away:

$$A \supseteq (A \wedge_* B) \supseteq (A \wedge B).$$

Let us consider now how $\wedge_*$ works. In the following, we will use the terms equivalence class and cluster synonymously. In line 7, $S/(A \wedge B)$ denotes the quotient of set S of all screens with equivalence relation $(A \wedge B)$, i.e., the set of clusters that S is partitioned into by $(A \wedge B)$. In order to find out which splits are new, i.e., done by B, we remove the clusters already produced by A, S/A. Now, P contains the new clusters that were created by conjoining A with B.

In the loop, we choose two different clusters p_1 and p_2 in P and test for all screens in $p_1 \cup p_2$ if the random variable "cluster of the screen" with possible values $\{p_1, p_2\}$ depends on a random variable that can be reasonably assumed to distinguish conceptually different screens, like usage (see Sect. 4.4). If the χ^2-test does not signify dependence, $\neg\chi^2(p_1, p_2)^*$, we reunite the two clusters in P. In that way, we iterate over all pairs of clusters. If equivalence relation B proves useless for distinguishing conceptually different screens, all clusters are reunited and A is not refined. In line 13, we reconstruct the desired equivalence relation A' from the cluster set P united with the clusters that are in A as well as $A \wedge B$, i.e., $(S/(A \wedge B) \cap S/A) \cup P$.

4.6 Examples for Refinement and Coarsening

We identify two main motivations for reverse engineering a web application with Revangie: the creation of a model of the application itself for the purpose of product benchmarking [21] or re-engineering and the creation of a model of its users for the purpose of testing the application, esp. load-testing. In the following, we will, for each of the motivations, look at cases where refinement or coarsening of the model is desirable.

4.6.1 Reconstruction of an Application Model

For the reconstruction of an application model, there is not a single recipe for screen classification. Generally, classification by targets identity is a good starting point but usually either too fine or too coarse. Consider, for example, the dynamic pages of a content-managed information system like the portal of an online newspaper. The start page of the newspaper may contain summaries of the latest articles with links to them, and although many different screens are instances of this conceptual page, they may differ heavily in the links and therefore in their targets signature. In this case, a good choice for coarsening the classification might be form targets identity. On the other hand, there may be conceptually different pages with the same targets signature, e.g., a page in a web shop to view articles and a conceptually different one, although the same in targets identity, to view special offers. In such cases, it can be useful to conjoin another classification like, for example, title identity. For some systems, it may be useful to readjust the classification for certain pages a-posteriori.

4.6.2 Reconstruction of a User Model

Reconstruction of a user model requires that a sufficient amount of session data has been collected in the snoop mode. The first natural classification is that by targets identity because, in a user model, we want to distinguish pages by the way they are used. Furthermore, we can use the clustering technique described in 4.5 with target choice or user response time as second random variable in order to determine a clustering that distinguishes screens with different user behavior well. Consequently, if screens are equivalent with respect to user behavior, they can be grouped in a common page, but classifications that group screens with different user behavior have to be refined.

5 Related Work

There are a couple of elaborated approaches [15, 11, 10, 9, 14, 1] that analyze source code to recover [3] a design or analysis model of the considered system. The motivation for these systems may range from pure system navigation visualization over maintainability support to the recovery of high-level architectural descriptions. Consider the tool presented in [11, 10, 9] as a representative example for this technology class. The tool analyses source code and pages of a web application and generates an architecture diagram that visualizes the interactions between static pages, active ASP or JSP pages and other software components by arrows. A similar support is offered by the tool WARE (Web Application Reverse Engineering) [14] for ASP and PHP based systems. Thereby, flat information about actual form parameters is recovered by this tool, too. A technique for recovering navigational structure and a conceptual model from a web application without tool support is described in [1].

VAQUISTA [20, 19] offers support for the static analysis of HTML pages. However, its goal is not to reverse engineer a whole web site but to reverse engineer the user interface from individual HTML pages in order to make them accessible from other contexts, i.e., device independency is targeted. An example for a tool that can track the change history of a web site is given in [2].

A tool for source-code independent analysis of web applications is ReWeb [16, 18]. An early version of ReWeb was able to recover a navigational model of a completely static web site. However, already in this early contribution the possibility to recover models for dynamic web pages solely by analyzing the generated HTML code has been envisioned. In [18], there is an outline of the new ReWeb features for the analysis of dynamic web sites. ReWeb is different from Revangie with respect to motivation, input simulation, server action classification, and notions of screen classifications. The motivation for the dynamic features of ReWeb is regression testing in web site evolution scenarios. ReWeb works similar to our crawl mode, but input values must be given before it is started. This means that an initial knowledge of the system is required already before ReWeb can be fully utilized. However, the choice of values can be challenging because inappropriate values can lead to partial models or duplicate pages in the model. ReWeb draws a distinction between implicit and explicit state models. The explicit state model regards each invocation of a CGI program with new values as a new server action. The Revangie state model is rather an implicit state model in the sense of ReWeb; however, the actions are distinguished with regard to their inferred types. The explicit type model has a couple of flaws: interaction state cannot necessarily be distinguished through hidden parameters because recovered hidden parameter values may loose their validity over time. When ReWeb is used on a web application, hidden parameters that are generated by the system are not propagated properly but overwritten by the ones in the explicit model, which may cause dynamic errors. The distinction of actions according to hidden parameters will lead to bloated models if hidden parameters are used for session management. ReWeb offers three notions of classification for grouping screens: textual identity, syntactical identity, and identity with respect to similarity metrics. Syntactical identity groups screens that have identical abstract syntax trees, i.e., equal up to textual content of tags. However, we feel syntactical identity is not the appropriate generalization of textual identity: consider the insertion of a new row into a table, which causes the resulting screens to be different with respect to syntactical identity, although they might very well be instances of the same conceptual page. In this important example, syntactical identity is no improvement on textual identity. However, the example can be dealt with very elegantly by means of syntactical pattern identity.

6 Further Directions

Once an adequate model for a web site is reconstructed, it may be desirable to extract certain features in order to make it clearer or facilitate subsequent redesign. Like many other software systems, also web sites usually contain aspects, i.e., parts that cross-cut the general structure. A common example for aspects in web applications are menus, which are important for the navigation on a web site, and it is possible to extract such features from the recovered model automatically. For example, forms and links that occur on several pages can be extracted and modeled separately with so called state sets, thereby making changes much easier and allowing users to view parts of the system independently of the whole without disturbing its general structure. For an account on how this can be done using form-oriented analysis, see [6].

Furthermore, we are about to implement Angil – a tool that uses the models created by Revangie in order to perform realistic load testing of web applications. The models used by Angil are annotated by additional data about typical user behavior, like transition probabilities and user response times. This data can be collected by Revangie in the snoop mode.

7 Conclusion

We introduced Revangie – a tool for reverse engineering web applications that reconstructs analysis models based on the concepts of form-oriented analysis. Revangie can operate in three different modes: the crawl-mode, which works automatically, the snoop-mode, which is user-driven, and the guide-mode, which works semi-automatically, combining the advantages of the other two modes. We described how these modes work, what problems come up when reverse engineering a web application, and what solutions we found. Especially, the problem of screen classification was discussed and its theoretical background described. We introduced a statistical test for determining if a classification leads to significant differences in the transition probability distributions of pages or other random variables of interest and described how this test can be exploited for screen clustering. Revangie is built on solid theoretical grounds and offers robust solutions to common problems.

References

[1] G. Antoniol, G. Canfora, G. Casazza, and A. D. Lucia. Web site reengineering using RMM. In *International Workshop on Web Site Evolution*, pages 9–16, March 2000.

[2] D. Budgen and S. Burgees. A simple tool for temporal indexing of hypertext documents. *Computer*, 31:52–53, December 1998.

[3] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, January 1990.

[4] D. Draheim, E. Fehr, and G. Weber. JSPick - a server pages design recovery. In *7th European Conference on Software Maintenance and Reengineering*, LNCS. IEEE Press, March 2003.

[5] D. Draheim and G. Weber. Modeling submit/response style systems with form charts and dialogue constraints. In *Workshop on Human Computer Interface for Semantic Web and Web Applications*, LNCS 2889. Springer, 2003.

[6] D. Draheim and G. Weber. Storyboarding form-based interfaces. In *INTERACT 2003 - Ninth IFIP TC13 International Conference on Human-Computer Interaction*. IOS Press, 2003.

[7] D. Draheim and G. Weber. *Form-Oriented Analysis - A New Methodology to Model Form-Based Applications*. Springer, September 2004.

[8] D. Draheim and G. Weber. Specification and generation of model 2 web interfaces. In *APCHI 2004 - 6th Asia-Pacific Conference on Computer-Human Interaction*, LNCS. Springer, June 2004.

[9] A. E. Hassan and R. C. Holt. Towards a better understanding of web applications. In *WSE 2001: International Workshop on Web Site Evolution*, November 2001.

[10] A. E. Hassan and R. C. Holt. Architecture recovery of web applications. In *ICSE 2002: International Conference on Software Engineering*, May 2002.

[11] A. E. Hassan and R. C. Holt. A visual architectural approach to maintaining web applications. *Annals of Software Engineering*, 16, 2003.

[12] K. Jajuga, A. Sokoowski, and H. H. Bock. *Classification, Clustering and Data Analysis*. Springer, August 2002.

[13] E. L. Lehmann. *Testing Statistical Hypotheses*. Springer, March 1997. 2nd Reprint edition.

[14] G. Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. de Carlini. Ware: A tool for the reverse engineering of web applications. In *Sixth European Conference on Software Maintenance and Reengineering*. IEEE, 2002.

[15] S. Mancoridis, T. S. Souder, Y.-F. Chen, E. R. Gansner, and J. L. Korn. Reportal: A web-based portal site for reverse engineering. In *8th Working Conference on Reverse Engineering*, pages 221–230. IEEE Press, November 2001.

[16] F. Ricca and P. Tonella. Analysis and testing of web applications. In *ICSE'2001 - International Conference on Software Engineering*, pages 25–34. IEEE Press, May 2001.

[17] F. Ricca and P. Tonella. Using clustering to support the migration from static to dynamic web pages. In *11th International Workshop on Program Comprehension*, pages 207–216, May 2003.

[18] P. Tonella and F. Ricca. Statistical testing of web applications. *Software Maintenance and Evolution*, 16(1-2):103–127, April 2004.

[19] J. Vanderdonckt and L. Bouillon. Retargeting of web pages to other computing platforms with VAQUISTA. In *9th Working Conference on Reverse Engineering*, pages 339–338. IEEE Press, November 2002.

[20] J. Vanderdonckt, L. Bouillon, and N. Souchon. Flexible reverse engineering of web pages with VAQUISTA. In *8th Working Conference on Reverse Engineering*, pages 241–248. IEEE Press, November 2001.

[21] G. H. Watson. *Strategic benchmarking - How to rate your company's performance against the world's best.* John Wiley, 1993.