PlayBug: Discovery by Software-knowledge Rules

Iaakov Exman and Shuri Hazani

Software Engineering Department, Jerusalem College of Engineering
POB 3566, 91035, Jerusalem, Israel

Abstract. Regression testing of software packages is useful to eliminate stubborn remaining bugs. But one often obtains relatively long sequences of commands needed to reproduce software failures and pinpoint bugs. A systematic approach to reduce these command sequences is essential for efficient bug discovery. This work proposes the use of rules expressing specific knowledge about the given software application. Rules are grouped in rule classes, which enable their application by a generic engine. The approach was validated by design and actual implementation of a PlayBug engine and its extensive testing on application families of software dealing with interactive GUI commands.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

1 Introduction

Regression testing is commonly used to discover new or recurring bugs when a software package is revised during development, leading to a new package version.

Usually when a bug is spotted, analyzed and fixed, a sequence of test commands that exposes the bug is recorded and retested after subsequent program changes.

An important problem is that the command sequences may be quite long, in fact longer than needed to expose a certain bug. Thus regression may be very inefficient.

This paper proposes reduction of long command sequences by using a priori knowledge about the software application. This knowledge is expressed as application specific rules which are input to a generic sequence reduction engine. In this work we refer to testing of interactive applications, such as usage of a text editing program. Often these applications are used by non-deterministic sequences of commands, thus different testing sequences of varying lengths may lead to the same bugs.

In this introduction we shortly review regression testing concepts and related work dealing with the command sequence reduction problem.

In the remaining of the paper we introduce command sequence reduction techniques (section 2), present the software architecture of a command sequence reduction module in the PlayBug engine (section 3), deal in detail with the software knowledge embedded in command sequence reduction rules (section 4), describe the validation experiments for our approach with PlayBug (section 5) and conclude with a discussion (section 6).

Exman I. and Hazani S..
PlayBug: Discovery by Software-knowledge Rules.
DOI: 10.5220/0003699500480056
In *Proceedings of the 2nd International Workshop on Software Knowledge* (SKY-2011), pages 48-56
ISBN: 978-989-8425-82-9
Copyright © 2011 SCITEPRESS (Science and Technology Publications, Lda.)

1.1 Software Regression Testing

Software regression testing is a modality of testing used again and again along the development process of a software package.

Whenever one adds code to software under development, or a bug is spotted and a patch added to fix the bug, the package must be regression tested to check whether the bug was indeed fixed and new bugs were not introduced. This is typically done automatically, for several bugs in the same run. Commonly the regression testing suite is run every night or once a week.

Some general references on regression testing are [1], [9] and [10].

1.2 Related Work on Command Sequence Reductions

In the technical literature the problem of the efficiency of regression testing has been attacked from several points of view.

An example is the granularity of regression testing suites, which can be reduced by various techniques (see e.g. Rothermel et al. [8]). Granularity in this sense usually does not refer to isolated commands, as in the present work.

A more closely related topic is the choice of path in programs – either to isolate bugs or to maximize bug occurrence – as in Lal et al. [2].

There are a variety of works on testing of GUI (Graphical User Interface) applications. Marchetto et al. [3] deal with state-based testing of AJAX Web applications. They refer to the semantic implications of command sequences. Among others, the issue of whether semantically interacting commands can be commutative.

Memon and collaborators published papers on regression testing of GUIs. Ref. [4] deals with test sequences as tasks in an Artificial Intelligence planning context. In Ref. [5] a DART environment for automated regression testing is described.

Orso et al. [6] deal with regression testing of components, using their metacontent.

2 Command Sequence Reduction

In this section we deal in detail with the command sequence reduction problem and some possible solutions.

2.1 The Command Sequence Reduction Problem

Whenever a change is made in a package with the intent to fix a bug that was previously located, various testing outcomes may occur:

a) either the bug is fixed or not;

b) either new bugs are introduced or not, with/without eliminating the old bug.

These possibilities may occur in distinct software paths taken by different runs. These may fail at the same bug, but in command sequences of various lengths.

SCIENCE AND TECHNOLOGY PUBLICATIONS

Thus one can define the command sequence reduction problem as follows:

- Given command sequences of various lengths, find a command sequence of minimal length that exposes a given bug.

2.2 Command Sequence Reduction Techniques

One can broadly classify command sequence reduction techniques as follows:

a- *probabilistic* – one selects by probabilistic criteria a certain fraction of sequences of given lengths – among all the command sequences that fail in the chosen bug – for further reduction;

b- *knowledge-based* – reduction is done according to knowledge-based rules.

In our work we have used techniques belonging to both classes. In this paper we focus on the knowledge-based rules.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

3 Command Sequence Reduction within PlayBug

A software regression engine containing a command sequence reduction sub-system – PlayBug – was designed and implemented to enable validation and actual utilization of our approach. A software regression engine has various functions, viz. running tests, analyzing run outcomes and recording command sequences. Again we focus on the command sequence reduction capabilities.

Here we describe the command sequence reduction sub-system within PlayBug.

3.1 Reduction Sub-system Architecture

A schematic diagram of the PlayBug reduction sub-system software architecture is seen in figure 1.

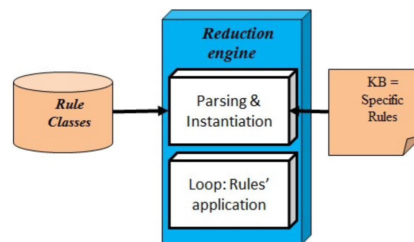

Fig. 1. PlayBug Architecture with ***Reduction Engine*** – schematic module diagram showing the command sequence reduction sub-system. Generic ***Rule classes*** are embedded in the system. The KB (Knowledge Base) containing Specific Rules is read as input for each kind of SUT (Software Under Test).

The PlayBug command sequence reduction sub-system contains two modules:

a) ***Reduction Engine*** – the engine actually performs reduction on the input com

mand sequences;

b) ***Rule Classes*** – this is an interface containing generic classes of rules embedded in the sub-system;

The reduction sub-system also receives as input a knowledge base (KB) containing specific rules that are only relevant to a given kind of SUT (Software Under Test).

3.2 Reduction Sub-system Implementation

The PlayBug reduction sub-system was actually implemented in the Java Language.

The ***Rule Classes*** is a Java interface implemented by Java classes, each one standing for a generic rule class. Classes have double meaning: a Java class for a rule class.

The KB (Knowledge Base) is a CSV file – comma separated value – file format, containing one textual specific rule in each row. The number of rows in the file is the number of specific rules for a kind of SUT. It is read as input into the "Parsing & Instantiation" sub-module of the Reduction Engine.

The overall functionality of the Reduction Engine is as follows:

1- The "Parsing & Instantiation" sub-module parses each specific rule of the KB and instantiates a corresponding object of the ***Rule Classes***. This object will perform the necessary reduction actions for the given rule. The number of objects equals the number of rules in the KB.

2- The "Loop: Rules' Application" sub-module sorts rules by priority and actually applies the specific rules to the test command sequence. Its logic is quite sophisticated: for instance, it does not immediately delete superfluous commands; it rather marks them for additional passes of the Loop. This will be described in detail elsewhere.

4 Command Sequence Reduction Rules

We have grouped command sequence reduction rules into generic classes, to enable easy reuse of these generic classes.

We first describe the generic classes and then give examples of specific rules for each class.

4.1 Generic Rule Classes

We currently use four generic rule classes:

1) *Anchor* – this class indicates that all commands preceding the “anchor” command can be deleted;

2) *PreSequence* – this class indicates a sequence of repeated commands, without other intermingled commands; all the commands in the sequence, except the last one, may be removed.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

3) *Itself* – this class indicates that a given command itself may be removed, because it has no semantic consequences;
4) *Ignored Preparation* – this class indicates a command type that may have semantic consequences, but in the given sequence it was ignored, i.e. remained without semantic consequences;

4.2 Specific Rules

We give specific rules in the context of an interactive GUI (Graphical User Interface) application.

Sample rules are given for each class:

1) *Anchor* – a "reset" operation is an anchor: all the commands preceding it may be deleted; other *anchor* examples are: "load", "reload", "unload";

2) *PreSequence* – a series of consecutive "clicking on a table row" operations is an example of a sequence; one may delete all clicks except the last one, since the previous unused clicks do not make any difference.

3) *Itself* – an "increase window size" operation may be removed; it has no semantic consequences on the window contents; other *itself* examples are: "help", "change layout";

4) *Ignored Preparation* – a "sorting" operation on a table, without using the resulting table is an ignored preparation; similarly "filtering", "grouping" and "search" are types of preparations.

It is important to stress that "semantic consequences" are application dependent. The same operation can have semantic consequences for a given application and no consequences for a different application.

SCIENCE AND TECHNOLOGY PUBLICATIONS

5 Validation

In this section we describe the validation technique and experimental results.

5.1 Experimental Technique

The experimental technique to validate our approach is to generate a large variety of command sequences for the same set of bugs – the source sequences – and apply the command sequence reduction to obtain shorter ones – the target sequences. These were actually performed with PlayBug.

Given the sets of source and target sequences, we plot them to check the efficiency of the knowledge-based approach.

5.2 Experimental Results

A typical graph of command sequences of a variety of sizes for a given exposed bug

is seen in Figure 2.

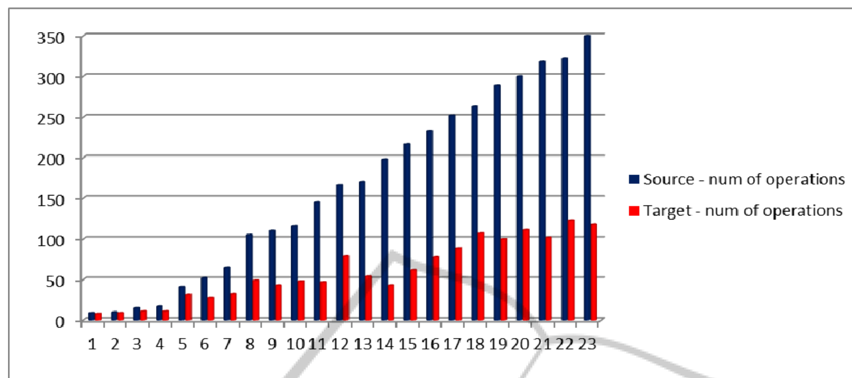

Fig. 2. Experimental Relative Reduction Graph – reductions of command sequences for a variety of sequence sizes. The vertical axis shows the sequence sizes in numbers of operations (commands). The horizontal axis is sorted by source sizes.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

In the next figure one can see the relative percentage change of the reduction for source sizes.

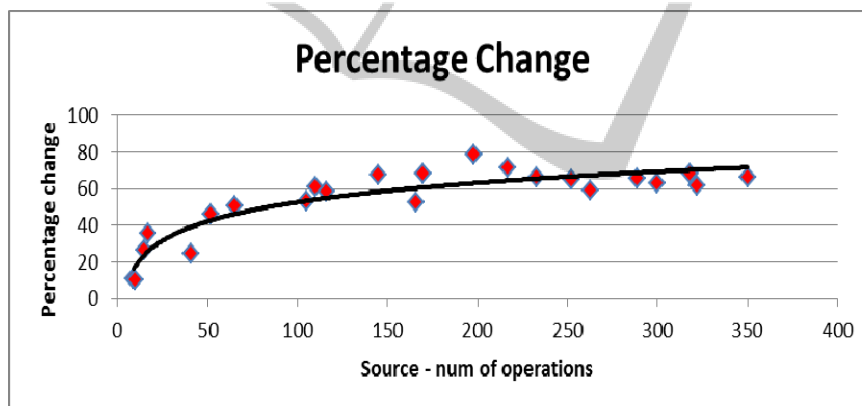

Fig. 3. Relative Percentage Reduction Graph – relative reductions of command sequences for various sequence sizes. The vertical axis shows the percentage changes.

5.3 Comparison with Expectations

For each set of command sequences that expose a certain bug, one expects the existence of a minimal finite sequence that can be theoretically achieved. Thus, for any command sequence length, there is a bound to reduction, i.e. reductions of command sequences are bounded below 100% reduction.

The next Fig. 4 displays the theoretical expected graph based on the previous argument, depicting the percentage change as a function of the source command sequence.

The explanation of this graph is simple to understand. For short source command sequences, the target after reduction cannot be much shorter than the source itself since it is very close to the minimal sequence size. With the growth of the source command sequences, reduction grows approximately linearly with the source size. When the source command sequences are very large the reduction is at most of the size of the source, viz. it is bounded below a 100% reduction.

In the same figure one sees that the experimental functional behavior closely follows the theoretical expectations. Moreover, the experimental results are bounded by 70% – only 30% below the theoretical bound – showing that PlayBug's efficiency is very high.

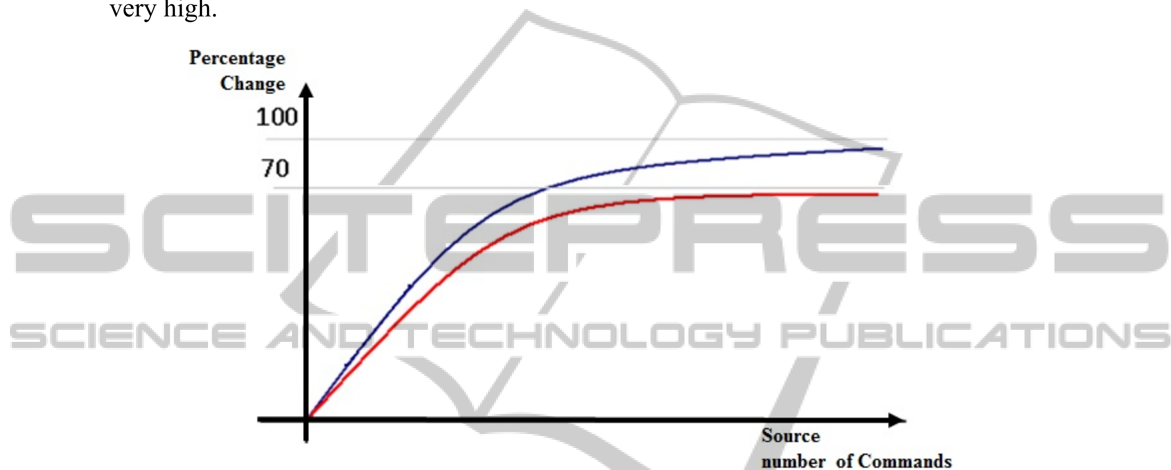

Fig. 4. Theoretical vs. Experimental Percentage Reduction Graph – relative reductions of command sequences for varying sourcesequence sizes. The vertical axis shows the percentage changes. The theoretical (blue) graph is bounded by 100%. The experimental (red) graph is bounded by 70%.

6 Discussion

A knowledge-based rules approach to command sequence reduction within regression testing has been described. Experimental results show that this is a promising approach – due to its relatively high efficiency – for interactive GUI applications.

Figure 5 maps in tabular form the generic rule classes already in use, with the intention to suggest degrees of freedom for possible additional generic classes. They are mapped against two class features of importance: scope and semantic interactions. Scope refers to the range of commands relevant to the class: preceding commands, the command itself and succeeding commands. Generic semantic interactions can be none, cancellation, preparation, and possibly others.

An example of a new class not used yet is "***Sequence***". It was suggested by such mapping, considering the symmetry relative to PreSequence.

One can conceivably have more complex rule classes. An example of such a rule class is "***Complementary***" (row 6 in Fig. 5) which refers to pairs of mutually canceling commands – e.g. expand and collapse.

#	Class Name	Scope			Semantic Interactions
		preceding	self	succeeding	
1	Anchor	*V*	*V*		Cancels preceding
2	PreSequence	*V*	*V*		Meaningful operation cancels preceding
3	Itself		*V*		None
4	Sequence		*V*	*V*	Meaningful operation acts on succeeding
5	Ignored Preparation			*V*	Preparation for succeeding
6	Complementary	*V*	*V*		Mutual

Fig. 5. Generic Rule Classes mapped in Tabular Form – The class scope is marked by a bold-italic ***V*** on a dark (orange) background. Classes are ordered from *preceding* towards *succeeding* scope, except for the "Complementary" class in the lower row.

6.1 Future Work

An open theoretical question is whether the minimal command sequence is unique.

Another theoretical issue is the number of generic rule classes. All the rule classes used in this work referred to commands independently of their arguments. Rule classes that are argument dependent are more computationally expensive. Thus, to reduce the 30% gap shown in Fig. 4 may not be cost effective.

A deeper question is the explanatory or discovery power of command sequences exposing a given bug. Are different sequences equivalent in this sense? Even if they have the same length?

Practical issues include the approach usability to other families of software applications, besides those inherently interactive. Extensive application of PlayBug should also contribute to a deeper understanding of the reduction upper bounds.

6.2 Main Contribution

This work uses knowledge-based rules for command sequence reduction within regression testing. Its main contribution is the grouping of rules into generic classes – a higher abstraction level in this context – similar to software design patterns for reuse.

Acknowledgements

Shuri Hazani wishes to express her gratitude to Noam Garber and Tal Tabakman from Cadence, Israel, for their help and participation in this project.

References

1. Chen, Y. Probert, R. L. and Sims, D. P.: Specification-based Regression Test Selection with Risk Analysis, in CASCON '02 Proc. Conference of the Centre for Advanced Studies on Collaborative research, IBM Press (2002).
2. Lal, A., Lim, J., Polishchuk, M. and Liblit, B.: Path Optimization in Programs and it's

Application to Debugging, LNCS Volume 3924/2006, pp. 246-263, Springer (2006).
3. Marchetto, A., Tonella, P. and Ricca, F.: State-Based Testing of Ajax Web Applications, in 1st Int. Conf. Software Testing, Verification and Validation, pp. 121-130 April (2008).
4. Memon, A. M.: Using Tasks to Automate Regression Testing of GUIs, in IASTED Int. Conf. on Artificial Intelligence and Applications – AIA, ACTA Press, 477-482 (2004).
5. Memon, A. M., Banerjee, I., Nada Hashmi, N. and Nagarajan, A.: DART: A Framework for Regression Testing “Nightly/daily Builds” of GUI Applications, in Proc. Int. Conf. on Software Maintenance, pp. 410-419 (2003).
6. Orso, A., Harrold, M. J., Rosenblum, D., Rothermel, G., Soffa, M. L. and Do, H.: Using Component Metacontents to Support the Regression Testing of Component-Based Software, in Proc. IEEE Int. Conf. on Software Maintenance, pp. 716-725 (2001).
7. Pan, K.: Bug Classification Using Program Slicing Metrics, Proc. Source Code Analysis and Manipulation Conference, SCAM '06, pages 31-42, (2006).
8. Rothermel, G., Elbaum, S., Malishevsky, A., Kallakuri, P. and Davia, B.: The Impact of Test Suite Granularity on the Cost Effectiveness of Regression Testing, in ICSE’02, 24th International Conference on Software Engineering pp.130, (2002).
9. Rothermel, G. and Harrold, M. J.: A Safe, Efficient Algorithm for Regression Test Selection, ACM Transactions on Software Engineering and Methodology (TOSEM) Volume 6 Issue 2, (1997).
10. Sen, A. and Srivastava, M.: Regression analysis - theory, methods and applications, Springer, New York , 1990.
11. Takao, S.: Bug Localization Based on Error-cause Chasing Methods, J. Information Processing 15(Extra), pp. 53-64, Japan, (1993).
12. Wahbe, R., Lucco, S., Anderson, T. and Graham, S.: Efficient Software-based Fault Isolation, ACM SIGOPS Operating Systems Review, ACM, New York (1993).

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS