4th Symposium on Search-Based Software Engineering
Riva del Garda | Trento | Italy
September 28th - 30th 2012

Fast Abstracts
4\textsuperscript{th} Symposium on
Search Based-Software Engineering
Riva del Garda | Trento | Italy
September 28\textsuperscript{th} – 30\textsuperscript{th} 2012

Fast Abstracts
Credits: Fototeca Ingarda Trentino S.p.A., Photo By: G. Paolo Calzà

Technical Editing and Book Design: FBK-Press, Trento, Italy

Please, remember that this file is provided for personal use in compliance with the legal terms and conditions of Italian copyright law. It cannot be uploaded on internet sites without authorization

# Table of Contents

Message from the SSBSE 2012 Chairs ........................................................................................................ IV

Conference Organization ......................................................................................................................... V

Sponsors ................................................................................................................................................ VII

Gergő Balogh, Ádám Zoltán Végh and Árpád Beszédes  
*Prediction of Software Development Modification Effort Enhanced by Genetic Algorithm* .................................................. 1

Sebastian Bauersfeld and Tanja Vos  
*A Reinforcement Learning Approach to Automated GUI Robustness Testing* ................................. 7

Mustafa Bozkurt and Mark Harman  
*Cost-aware Test Suite Minimisation for Service-centric Systems* ..................................................... 13

Mathew Hall and Phil McMinn  
*An Analysis of the Performance of the Bunch Modularisation Algorithm’s Hierarchy Generation Approach* ........................................................................................................ 19

Guanzhou Lu, Jinlong Li and Xin Yao  
*Embrace the New Trend in SBSE with Fitness-Landscape Based Adaptive Evolutionary Algorithm* .................................................................................................................. 25

Sophia Nobrega, Sergio Ricardo de Souza and Marcone Jamilson Freiast Sousa  
*Development of Software Project Schedule Using Heuristic Optimization* .................................... 31

Christopher Simons and James Smith  
*A Comparison of Evolutionary Algorithms and Ant Colony Optimization for Interactive Software Design* ......................................................................................................................... 37

Jerry Swan, Mark Harman, Gabriela Ochoa and Edmund Burke  
*Generic Software Subgraph Isomorphism* ............................................................................................. 43

Benjamin Wilmes  
*Toward a Tool for Search-Based Testing of Simulink/TargetLink Models* .................................. 49
Message from the SSBSE 2012 Chairs

Welcome to the Fast Abstracts collection of the 4th Symposium on Search Based Software Engineering, SSBSE 2012, held in Riva del Garda, in the Province of Trento, in Italy. Riva del Garda is a small city located at the north-western corner of the Garda Lake, in the middle of the Alps, surrounded by Mediterranean vegetation with olive and lemon trees. The symposium was co-located with the 28th IEEE International Conference on Software Maintenance, the premiere international venue in software maintenance and evolution, to promote the cross-fertilization between the two research communities.

Fast abstracts offer an opportunity to present novel ideas that are not yet fully developed or validated to the community, and they are an established tradition of SSBSE. Fast abstracts are subject to a lighter reviewing process, where each paper was reviewed by at least two reviewers. The collection of fast abstracts demonstrates that search-based software engineering increasingly influences all phases of the software engineering process from requirements analysis to software testing and maintenance.

Many people contributed in different ways to the organization of the symposium. In particular, we would like to thank the members of the Program Committee, of the Organizing Committee and of the Steering Committee for their valuable help in the scientific and organizational aspects.

Thanks also to FBK, University of Sheffield, Univerity do Cearà, UCL CREST department, Berner & Mattner, IBM, Springer and FITTEST project for their support to the Symposium.

We hope this symposium was an occasion to create new fruitful synergies between research groups from around the world, as well as between different disciplines and cultures.

See you next year at SSBSE 2013 in St. Petersburg, Russia.

August 2012

Gordon Fraser, Angelo Susi, Jerffeson Teixeira de Souza
Conference Organization

General Chair
Angelo Susi
Fondazione Bruno Kessler, Italy

Program Chairs
Gordon Fraser
University of Sheffield, UK
Jerffeson Souza
Universidade Estadual do Ceara, Brazil

Graduate Students Track Chair
Shin Yoo
University College London, UK

Organization Committee
Alessandro Marchetto
Fondazione Bruno Kessler, Italy
Matthieu Vergne
Fondazione Bruno Kessler, Italy
Moira Osti
Fondazione Bruno Kessler, Italy

Program Committee
Enrique Alba
University of Málaga, Spain
Giulio Antoniol
Ecole Polytechnique de Montreal, Canada
Andrea Arcuri
Schlumberger and Simula Research Laboratory, Norway
Marcio Barros
Universidade Federal do Estado do Rio de Janeiro, Brazil
Leonardo Bottaci
University of Hull, UK
Lionel Briand
University of Luxembourg, Luxembourg
Francisco Chicano
University of Málaga, Spain
Mel Ó Cinnéide
University College Dublin, Ireland
John Clark
University of York, UK
Myra Cohen
University of Nebraska-Lincoln, USA
Massimiliano Di Penta
RCOST - University of Sannio, Italy
Robert Feldt
University of Blekinge, Chalmers University of Technology, Sweden
Mark Harman
University College London, UK
Rob Hierons
Brunel University, UK
Colin Johnson
University of Kent, UK
Gregory Kapfhammer
Allegheny College, UK
Yvan Labiche
Spiros Mancoridis
Phil McMinn
Pasqualina Potena
Simon Poulding
Xiao Qu
Marek Reformat
Marc Roper
Guenther Ruhe
Paolo Tonella
Silvia Vergilio
Tanja Vos
Westley Weimer
Yuanyuan Zhang

Carleton University, Canada
Drexel University, USA
University of Sheffield, UK
Universita degli Studi di Bergamo, Italy
University of York, UK
ABB Corporate Research, USA
University of Alberta, Canada
University of Strathclyde, UK
University of Calgary, Canada
Fondazione Bruno Kessler, Italy
Universidade Federal do Paraná, Brazil
Universidad Politecnica de Valencia, Spain
University of Virginia, USA
University College London, UK

External Reviewers

Nadia Alshahwan
Nesa Asoudeh
Arthur Baars
S. M. Didar-Al-Alam
Iman Hemati Moghadam
Reza Karimpour
Sebastian Maurice
Shiva Nejati
Arash Niknafs
Mehrdad Sabetzadeh
Pingyu Zhang

University College London, UK
University of Luxembourg, Luxembourg
Carleton University, Canada
Universidad Politecnica de Valencia, Spain
University of Calgary, Canada
University College Dublin, Ireland
University of Calgary, Canada
Simula Research Laboratory, Norway
University of Calgary, Canada
Simula Research Laboratory, Norway
University of Nebraska-Lincoln, USA

Steering Committee

Mark Harman (chair)
Giulio Antoniol
Lionel Briand
Myra Cohen
Massimiliano Di Penta
Phil McMinn
Mel Ó Cinnéide
Jeffreson Souza
Joachim Wegener

University College London, UK
Ecole Polytechnique de Montreal, Canada
Universite du Luxembourg, Luxembourg
University of Nebraska Lincoln, USA
University of Sannio, Italy
University of Sheffield, UK
University College Dublin, Ireland
Universidade Estadual do Ceara, Brazil
Berner and Mattner, Germany
Sponsors

"The IBM logo is a registered trademark of International Business Machines Corporation (IBM) in the United States and other countries"
Prediction of Software Development Modification Effort Enhanced by a Genetic Algorithm

Gergő Balogh, Ádám Zoltán Végh, and Árpád Beszédes

Department of Software Engineering University of Szeged, Szeged, Hungary
{geryxyz,azvegh,beszedes}@inf.u-szeged.hu

Abstract. During the planning, development, and maintenance of software projects one of the main challenges is to accurately predict the modification cost of a particular piece of code. Several methods are traditionally applied for this purpose and many of them are based on static code investigation. We experimented with a combined use of product and process attributes (metrics) to improve cost prediction, and we applied machine learning to this end. The method depends on several important parameters which can significantly influence the success of the learning model. In the present work, we overview the usage of search based methods (one genetic algorithm in particular) to calibrate these parameters. For the first set of experiments four industrial projects were analysed, and the accuracy of the predictions was compared to previous results. We found that by calibrating the parameters using search based methods we could achieve significant improvement in the overall efficiency of the prediction, from about 50% to 70% (F-measure).

Keywords: software development, effort estimation, modification effort, genetic algorithm

1 Introduction

One of the tasks in software cost estimation, especially in the evolution phase, is to predict the cost (required effort) for modifying a piece of code. A possible approach for such modification effort prediction is to use various software attributes from historical development data and from the current version of the software. The attributes can be expressed in the form of software metrics, both product and process. Product metrics are calculated by performing the static analysis of the software (a simple example is the logical lines of code), while process metrics can represent time-related quantities collected during project implementation (for example, the Net Development Time of the modifications). The metrics can then be used, with the help of a model, to predict the modification costs.

Numerous articles were written about experiences using process or product metrics [1,2], but researches using both are rare. Our previous works tried to predict the cost of future modifications by applying both product metrics calculated from the source code and process metrics collected during the development [3]. We compared the results when using only one type of metrics with the case when both kinds of metrics had been used. We applied machine learning to create the model for the prediction. In an advanced model we used a more complex metric for expressing the modification effort, which was the aggregation of different kinds of modifications like creation, deletion, and type change. To express the modification effort in a single value, we used different parameters (weights) for the different kinds of modifications. The choice of these parameters was crucial for the accuracy of prediction and their calibration was not simple. In the present work, we
report on our early experiences in applying search based methods to determine these parameters (a basic genetic algorithm (GA) was used for this purpose). A typical improvement of as much as 20 percentage points was achieved in the combined prediction accuracy (F-measure on the precision and recall) when comparing the model with initial parameters to the one obtained after running the search method.

We summarize the experiments with the following research questions: RQ. 1 Is it possible to define a better model for software development effort prediction than the one used in paper [3,4]? In particular, we seek for a better way to express modification effort. RQ. 2 Can the GA be used to improve the precision of the estimation of software development effort compared to the results in paper [3,4]? In particular, what rate of improvement can be obtained after calibrating the crucial parameters of the cost metric and re-applying them to the model?

To achieve the mentioned goals we implemented a framework which is capable to collect and aggregate product and process metrics from various sources including the source code and the integrated development environment. The framework detects the modifications between revisions, and tries to predict the effort of further changes.

2 Overview

The overall process is shown in Fig. 1. The experiment starts with a measurement phase where the used data is collected from various sources: the metrics about the evolution of the software, the source code from the SVN version controlling system, and the metrics estimations which were given by the project manager. This phase has two main tasks; to collect and calculate the process and product metrics, and to detect and group the modifications of the source code between the revisions. The metrics and the modification groups are sent to the GA which prepares a population of individual entities. During the initial set-up of the population and the evolution steps, two metrics are calculated: Typed Modification (TMod), which was defined as the weighted count of modifications between two revisions; and Modification Effort (Meff), the ratio of TMod and the net development time of these modifications. Afterwards the prediction model is tested and its weighted F-measure value is used as fitness to rank the individuals in the population and select the best entities for breeding. As the final step of the evolution cycle, the new weights of the modification are calculated and the model is updated. When the precision reaches an appropriate value, the GA stops and a new, enhanced model is built using the weights of the best entity in the final population. This Meff prediction model is the output of the execution of the framework.

Data was collected during the experiment from about 800 revision, about 75 days long. Both R&D and industrial projects were analysed and the source code was written in Java language using the Java EE 6 virtual machine and the Seam 2 framework. Altogether 2200 records were collected as learning set.

3 Modification Effort Prediction

Process and product metrics were used together per source file basis as separated entities. Then, the appropriate cost function was calculated from this data. The metrics used as predictors were the followings: (i) Logical Lines Of Code (LLOC) (ii) C&K metric suit defined by Chidamber and Kemerer[2] (iii) Estimated development time of a task, aggregated into 3 groups: short, medium,
To extend our previous framework, a new metric called Modification Effort (Meff) was defined which was calculated as the following: first, the modifications were grouped, based on the target entity (e.g.: method) and the performed action (e.g.: creation) [3]. The weighted count of the previously defined modifications was called Typed Modification (TMod), this expresses the different amount of developer's effort used in the modifications. The ratio of the net development time with the TMod is the Meff metric.

The weight was defined on the basis of the per groups of modification. We established a single rule for each disjunct group which determined the contained relation.

The following modification groups were measured:
- class creation deletion accessibility change
- method creation deletion accessibility change prototype change return type change size change
- data member creation deletion accessibility change type change

Instead of the Level of Modification Complexity (LMC) – defined as the ratio of the Effectively Changed Lines Of Code (ECLOC) for the next change of the file/class and DT, the net development time between two revisions [3,4] – Meff was used as the target function, which was equally labelled with: low, medium and high values.

The Weka framework [6] machine learning and the 10-fold cross-validation utility were used to evaluate our model. The weight sum of the F-measure was chosen as the fitness value which is the harmonic mean of precision and recall.

4 Genetic Algorithm

SBSE was used to fine tune the weights of the modification. The initial weight-vector was set by our developer experience. The algorithm was iterated on the basis of previously set scenario. Our assumption was that the genetic algorithm should converge to the suitable weights, which should provide a more accurate estimation [7].

---

1 detailed in paper [3]
As previously described, a weighted sum of the count of modification group was used as the target function. A genetic algorithm (GA) was used to fine tune the weights of each group.

The individuals identified by its chromosome, which is a vector over the real numbers with the same dimension. In the model each chromosome represents a weight-vector, and every element determines the weight of a single modification group.

The fitness value is calculated for each individual by evaluating the model with the weights defined in that particular individual. The final goal of the GA is to improve the precision of the model. For classification problems, the F-measure value can give a reliable approximation of the accuracy. The base model was not enhanced with the GA, but it was evaluated using the F-measure. Thus the F-measure was chosen to be the fitness value of the GA.

An evolution step starts with the breeding process which consists of two steps, first, the GA selects the two best entities with its fitness value. The crossover operator will apply only this pair. Every call of the crossover operator produces exactly one child. The algorithm repeats the operation to produce more than one child.

We used a uniform crossover logic. During the crossover the algorithm iterates via the elements of the chromosome (vector) and randomly chooses an element from one of the two parents. Every element has the same chance to be copied into the child’s chromosome [8].

The chromosome of the children is subject to mutation. A lower limit and an upper limit were determined for the weights of the groups. During the mutation some elements (weights) of the chromosome change. The algorithm gets the half of the distance between the limits and the current selected weight and sets the current value either to the lower or to the upper half point. This way, the two limits are never exceeded. Then, the child is included in the population.

The individuals with the worst fitness value are killed (removed from the population) to maintain the size of the population, this way the current evolution step is completed and the algorithm proceeds to the next generation [9,10].

The above mentioned GA parameters and their values are shown in Tab. 1.

<table>
<thead>
<tr>
<th>Table 1. GA parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial mutation rate</td>
</tr>
<tr>
<td>mutation rate</td>
</tr>
<tr>
<td>mutation lower limit</td>
</tr>
<tr>
<td>mutation upper limit</td>
</tr>
<tr>
<td>birth count</td>
</tr>
<tr>
<td>crossover rate</td>
</tr>
<tr>
<td>population size</td>
</tr>
<tr>
<td>generation count</td>
</tr>
</tbody>
</table>

5 Preliminary Results

As shown in Fig. 2, the fitness value of the prediction grows in every case. Tab. 2 shows the same fitness values. As it can be seen in Tab. 3, the average grows with about 18 percent. It is also relevant that in the worst case our model proves to be better with about 16 percent.

With these pieces of information the two research questions can be answered.
Fig. 2. Fitness values of prediction per project

Table 2. Fitness values of prediction per project

<table>
<thead>
<tr>
<th>Project 1</th>
<th>Project 2</th>
<th>Project 3</th>
<th>Project 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>base model</td>
<td>59.8000%</td>
<td>47.0000%</td>
<td>44.2000%</td>
</tr>
<tr>
<td>enhanced model</td>
<td>75.2630%</td>
<td>66.6425%</td>
<td>67.4835%</td>
</tr>
</tbody>
</table>

Table 3. Comparison of models

<table>
<thead>
<tr>
<th></th>
<th>worst</th>
<th>best</th>
<th>average</th>
<th>median</th>
</tr>
</thead>
<tbody>
<tr>
<td>base experiment</td>
<td>44.2000%</td>
<td>59.8000%</td>
<td>49.0750%</td>
<td>46.1500%</td>
</tr>
<tr>
<td>enhanced model</td>
<td>60.2791%</td>
<td>75.2630%</td>
<td>67.4170%</td>
<td>67.0630%</td>
</tr>
<tr>
<td>difference</td>
<td>16.0791%</td>
<td>15.4630%</td>
<td>18.3420%</td>
<td>20.9130%</td>
</tr>
</tbody>
</table>

RQ. 1 Yes, it is possible. Our model gives a better estimation from the beginning of the evolution and the population average fitness value is higher in every generation.

RQ. 2 Yes, the use of the GA can improve precision. With this simple GA implementation a significant improvement was reached.

The weight of groups was aggregated from all four projects and weighted with the size of the learning set. Two aspects were created to examine the validity of the weights calculated by the GA.

Fig. 3. Aggregated weights of groups
These aspects are shown in Fig. 5. The values can be interpreted as the importance of a modification, i.e. how much gain will be achieved by applying the modification. The diagram on the left shows a by action aggregation. As it can be seen, the creation and deletion are more important than the type and visibility changes. On the right side, a subject based aggregation can be seen. The most important modification was applied on the method elements which included the method body modifications as well.

6 Conclusion and Plans

In this paper, we present two new metrics and a new procedure with which we can increase the effectiveness of our modification cost prediction method based on product and process metrics and machine learning. Our previous results have been improved with the introduction of new metrics \((\text{TMod}, \text{Meff})\) used as target function, and by using a genetic algorithm to calibrate certain crucial parameters required by the model. We were able to increase the success of the prediction model significantly. We manually investigated the final parameter values produced by the \(\text{GA}\). These parameters seem to be valid based on our own developer experience, but further analysis will be needed to validate the results.

In the future, we plan to prepare a new set of parameters by collecting developer experience within our team. We plan to use a questionnaire and compare the results with the automatically calculated ones, which do not use the bias of developer experiment. We also plan to repeat the experiment on a bigger dataset and apply the method to achieve different goals like comparing projects or developer productivity. We would like to fine tune our \(\text{GA}\) with the testing of some other crossover operator or fitness value.

References

A Reinforcement Learning Approach to Automated GUI Robustness Testing

Sebastian Bauersfeld and Tanja E. J. Vos

Universitat Politècnica de València,
Camino de Vera s/n, 46022, Valencia, Spain
{sbauersfeld,tvos}@pros.upv.es

http://www.upv.es

Abstract. Graphical User Interfaces (GUIs) can be found in almost all modern desktop, tablet and smartphone applications. Since they are the glue between an application’s components, they lend themselves to system level testing. Unfortunately, although many tools promise automation, GUI testing is still an inherently difficult task and involves great manual labor. However, tests that aim at critical faults, like crashes and excessive response times, are completely automatable and can be very effective. These robustness tests often apply random algorithms to select the actions to be executed on the GUI. This paper proposes a new approach to fully automated robustness testing of complex GUI applications with the goal to enhance the fault finding capabilities. The approach uses a well-known machine learning algorithm called Q-Learning in order to combine the advantages of random and coverage-based testing. We will explain how it operates, how we plan to implement it and provide arguments for its usefulness.

Keywords: gui testing, automated testing, reinforcement learning

1 Introduction

Graphical User Interfaces (GUIs) represent the main connection point between a software’s components and its end users and can be found in almost all modern applications. This makes them attractive for testers, since testing at the GUI level means testing from the user’s perspective and is thus the ultimate way of verifying a program’s correct behavior. However, current GUlIs are large, complex and difficult to access programmatically, which poses great challenges for their testability. In an earlier work [1] we presented a framework called GUITest to cope with these challenges. GUITest allows to generate arbitrary input sequences even for large Systems Under Test (SUT). The basic procedure comprises three steps: 1) Obtain the GUI’s state (i.e. the visible widgets and their properties like position, size, focus ...). 2) Derive a set of sensible actions (clicks, text input, mouse gestures, ...). 3) Select and execute an action. Figure 1 gives an example of how GUITest iteratively applies these three steps in order to drive Microsoft Word. The ultimate goal is to generate sequences that crash the GUI or make it unresponsive.
Fig. 1. Sequence generation by iteratively selecting from the set of currently available actions. The ultimate goal is to crash the SUT. (Not all possible actions are displayed in order to preserve clarity)
In its current form, GUITest selects the actions to be executed at random. We believe that this is a straightforward and effective technique of provoking crashes and reported on its success in [1]. We were able to find 14 crash sequences for Microsoft Word while running GUITest during 48 hours\(^1\). Some of the advantages and disadvantages of random action selection are:

+ No model or instrumentation required: This is an important aspect, since usually there exists no abstract model of the GUI. Generating one often involves manual effort and, due to size constraints, it is either not complete or not entirely accurate [2].
+ Easy to maintain: Tests continue to run in the presence of GUI changes.
+− Unbiased: Each permutation of actions is equally likely to be generated. In the absence of domain knowledge about the SUT this is desirable. However, quite often certain aspects require more thorough testing than others.
− Not every action is equally likely to be executed.
− It can take a considerable amount of time until a sufficiently large percentage of the GUI has been operated.

In this paper we strive to improve on the last three characteristics. In large SUTs with many – potentially deeply nested – dialogs and actions, it is unlikely that a random algorithm will sufficiently exercise most parts of the GUI within a reasonable amount of time. Certain actions are easier to access and will therefore be executed more often, while others might not be executed at all.

The idea is to slightly change the probability distribution over the sequence space. This means that action selection will still be random, but seldom executed actions will be selected with a higher likelihood, in order to favor exploration of the GUI. The straightforward greedy approach of selecting at each state the action which has been executed the least number of times, might not yield the expected results: In a GUI it is often necessary to first execute certain actions in order to reach others. Hence, these need to be executed more often, which requires an algorithm that can “look ahead”. This brought us to consider a reinforcement learning technique called Q-Learning. In the following section we will explain how it works, define the environment that it operates in and specify the reward that it is supposed to maximize.

2 The Algorithm

We assume that our SUT can be modeled as a finite Markov Decision Process (MDP). A finite MDP is a discrete time stochastic control process in an environment with a finite set of states \(S\) and a finite set of actions \(A\) [3]. During each time step \(t\) the environment remains in a state \(s = s_t\) and a decision maker, called agent, executes an action \(a = a_t \in A_s \subseteq A\) from the set of available actions, which causes a transition to state \(s' = s_{t+1}\). In our case, \(A\) will refer

\(^1\) Videos of these crashes are available at https://staq.dsic.upv.es/sbauersfeld/index.html
to the set of possible GUI actions, i.e. clicks on widgets, text input and mouse gestures and $S$ will be the set of observable GUI states.

The state transition probabilities are governed by

$$P(a, s, s') = \Pr\{s_{t+1} = s'|s_t = s, a_t = a\}$$

meaning, that the likelihood of arriving in state $s'$ exclusively depends on $a$ and $s$ and not on any previous actions or states. This property, which is crucial to the application of reinforcement learning algorithms, is called Markov Property. We assume that it approximately holds for the SUT\(^2\).

In an MDP, the agent receives rewards $R(a, s, s')$ after each transition, so that it can learn to distinguish good from bad decisions. Since we want to favor exploration, we set the rewards as follows:

$\begin{align*}
R(a, s, s') &= \begin{cases}
    r_{\text{init}}, & \text{if } x_a = 0 \\
    \frac{1}{x_a}, & \text{else}
\end{cases}
\end{align*}$

Where $x_a$ is the amount of times that action $a$ has been executed and $r_{\text{init}}$ is a large positive number. Hence, the more often an action has been executed, the less desirable it will be for the agent. The ultimate goal is to learn a policy $\pi$ which maximizes the agent’s expected reward. The policy determines for each state $s \in S$ which action $a \in A_s$ should be executed. We will apply the Q-Learning algorithm in order to find $\pi$. Instead of computing it directly, Q-Learning first calculates a value function $V(s, a)$ which assigns a numeric quality value – the expected future reward – to each state action pair $(s, a) \in S \times A$.

This function is essential, since it allows the agent to look ahead when making decisions. Eventually, it becomes trivial to derive the optimal policy: One selects $a^* = \arg\max_a \{V(s, a)|a \in A_s\}$ for each $s \in S$.

Algorithm 1 shows the pseudocode for our approach. The agent starts off completely uninformed, i.e. without any knowledge about the GUI. Step by step it discovers states and actions and learns the value function through the rewards it obtains. The quality value for each new state action pair is initialized to $r_{\text{init}}$. The heart of the algorithm is the value update in line 9: The updated quality value of the executed state action pair is the sum of the received reward plus the maximum value of all subsequent state action pairs multiplied by the discount factor $\gamma$. The more $\gamma$ approaches zero, the more opportunistic and greedy the agent becomes (it considers only immediate rewards). When $\gamma$ approaches 1 it will opt for long term reward. It is worth mentioning that the value function and consequently the policy will constantly vary, due to the fact that the rewards change. This is what we want, since the agent should always put emphasis on the regions of the SUT which it has visited the least amount of times.

Instead of always selecting the best action (line 6), one might also consider a random selection proportional to the values of the available state action pairs. This would introduce more randomness in the sequence generation process.

---

\(^2\) Since we can only observe the GUI states and not the SUT’s true internal states (Hidden Markov Model), one might argue whether the Markov Property holds sufficiently. However, we assume that this has no significant impact on the learning process.
Input: $r_{init}$  
/* reward for unexecuted actions */

Input: $0 < \gamma < 1$  
/* discount factor */

1 begin
2 start SUT
3 $V(s, a) \leftarrow r_{init}$  
$\forall (s, a) \in S \times A$
4 repeat
5 obtain current state $s$ and available actions $A_s$
6 $a^* \leftarrow \arg\max_a \{V(s, a) | a \in A_s\}$  
/* select the best action */
7 execute $a^*$
8 obtain state $s'$ and available actions $A_{s'}$
9 $V(s, a^*) \leftarrow R(a^*, s, s') + \gamma \cdot \max_{a \in A_{s'}} V(s', a)$  
/* value update */
10 until stopping criteria met
11 stop SUT
12 end

Algorithm 1: Sequence generation with Q-Learning

Representation of States and Actions In order to be able to apply the above algorithm, we have to assign a unique and stable identifier to each state and action, so that we are able to recognize them, even across different runs of the SUT. Our testing library GUITest allows us to derive the complete GUI state of the SUT, so that we can inspect the property values of each visible widget. For example: It gives access to the title of a button, its position and size and tells us whether it is enabled. This gives us the means to create a unique identifier for a click on that button: It can be represented as a combination of the button’s property values, like its title, help text or its position in the widget hierarchy (which parents / children does the button have). The properties selected for the identifier should be relatively “stable”: The title of a window, for example, is quite often not a stable value (opening new documents in Word will change the title of the main window) whereas its tool tip or help text is less likely to change. The same approach can be applied to represent GUI states: One simply combines the values of all stable properties of all widgets on the screen. Since this might be a lot of information, we will only save a hash value generated from these values\(^3\). This way we can assign a unique and stable number to each state and action.

3 Related Work

Artzi et al. [4] perform feedback-directed random test case generation for JavaScript web applications. Their objectives are to find test suites with high code coverage as well as sequences that exhibit programming errors, like invalid-html or runtime exceptions. They developed a framework called Artemis, which triggers events by calling the appropriate handler methods and supplying them with the

\(^3\) Of course this could lead to collisions. However, for the sake of simplicity we assume that this is unlikely and does not significantly affect the optimization process.
necessary arguments. To direct their search, they use prioritization functions: They select event handlers at random, but prefer the ones for which they have achieved only low branch coverage during previous sequences.

Marchetto and Tonella [5] generate test suites for AJAX applications using metaheuristic algorithms. They execute the applications to obtain a finite state machine, whose states are instances of the application’s DOM-tree (Document Object Model) and whose transitions are events (messages from the server / user input). From this FSM they calculate the set of semantically interacting events. The goal is to generate test suites with maximally diverse event interaction sequences, where each pair of consecutive events is semantically interacting.

The strength of our approach is, that it works with large, native applications which it can drive using complex actions. The abovementioned approaches either invoke event handlers (which is not applicable to many GUI technologies) or perform only simple actions (clicks). Moreover, our technique does not modify nor require the SUT’s source code, which makes it applicable to a wide range of programs.

4 Future Work

In this paper we proposed a technique for improving the effectivity of GUI robustness testing. We believe in the ability of random testing to provoke crashes in GUI-based applications. However, due to the fact that modern GUIs often have deeply nested dialogs and actions, which are unlikely to be executed by a random algorithm, we think that a more explorative approach could improve the fault finding effectivity. We are therefore integrating an algorithm known as Q-Learning into our test framework GUITest. We think that this algorithm, which was designed to solve problems in the presence of uncertainty and lack of environmental knowledge, is suitable for testing large and complex GUIs without additional information provided by formal models. We expect to finish the implementation soon and look forward to report on the results.

Acknowledgments. This work is supported by EU grant ICT-257574 (FITTEST).

References

Cost-aware Test Suite Minimisation for Service-centric Systems

Mustafa Bozkurt and Mark Harman
{m.bozkurt, m.harman}@cs.ucl.ac.uk
CREST Centre, Department of Computer Science, University College London.
Malet Place, London WC1E 6BT, UK

Abstract. The cost associated with testing is one of the major limitations in testing service-centric systems. Specifically runtime testing cost can be reduced by selecting test cases which provides high coverage while incurring a minimal service invocation cost. We formulated this as an optimisation problem and suggest the use of multi-objective optimisation approaches as a possible solution.

1 Introduction

One of the limitations in testing of Service-centric Systems (ScS) is the cost associated with invoking services [3, 4]. ScS Testing (ScST) costs can include:

1. The monetary cost of invoking services during runtime testing.
2. Disruptions that might be caused by testing.
3. Side effects of testing in transactional systems, such as stock-exchange systems.

Increased test frequency for ScS exacerbates the severity of issues regarding testing cost. There are two levels in ScST; composition level and service level. The cost of invoking services during testing is a major problem at composition level. Solutions aimed at reducing the cost of testing, such as simulated testing (discussed in Section 2) have previously been proposed. However these approaches do not eliminate the need for runtime testing (testing with real services).

One widely studied solution aimed at reducing testing cost is test suite minimisation. The purpose of test suite minimisation and prioritisation is to reduce testing cost through removing redundant test cases [12]. In test suite minimisation there are concerns such as retaining coverage, test suite effectiveness or fault detection capability, execution time and resource usage, such as memory. According to the literature, problems where there are multiple competing and conflicting concerns/objectives can be investigated using pareto optimal optimisation approaches [5]. Previously, a pareto optimal test suite minimisation approach is proposed by Yoo and Harman [13].
In earlier work we addressed ScST’s need for realistic test inputs by introducing the concept of service-centric test data generation. We developed a tool called ATAM [1] that composes services to provide service mashups which enable automated realistic input generation. ATAM is able to generate arbitrary service compositions to find more elaborate realistic test data that fulfills tester-specified constraints. Though ATAM can generate realistic test suite for ScST, there remains the issue of controlling the test cost.

In order to avoid the side effects of extensive runtime testing, a test suite minimisation approach could be used to remove ‘redundant’ test cases; those which merely cover the previously covered features. Thus, we propose to apply to ScST the multi-objective test suite minimisation algorithm proposed by Yoo and Harman with ScST objectives, such as, cost of service invocation, branch coverage and test suite reliability.

The advantages of the proposed application of multi-objective test suite minimisation for ScS are:

1. Reduced cost in runtime testing through test suite minimisation.
2. Its ability to discover trade-offs between cost of test runs and system coverage.
3. Its ability to select a more reliable test suite without increasing the cost and effecting the coverage of the test suite.

The rest of this paper is organised as follows. Section 2 reviews existing test suite minimisation approaches and approaches proposed in testing of service compositions. Section 3 explains the proposed multi-objective test suite optimisation approach for ScST. Section 4 concludes.

2 Background

Test suite minimisation (or test suite reduction) techniques aim to reduce the size of a test suite by eliminating redundant test cases [12]. Due to the test suite reduction problem (which is a minimal set cover problem) being NP-complete, the use of heuristics proposed by many researchers [12]. However, the first multi-objective test suite minimisation approach is proposed by Yoo and Harman [13] where test suite coverage, previous fault detection capability and execution time are selected as objectives.

In the literature, unit testing of ScS is performed in two ways; simulated testing and real-world testing [3]. Real-world testing (or runtime testing) is the testing process in which a fully working instance of the ScS is tested. In simulated testing, testing is performed as a simulation, replacing regular deployment and invocation with ‘stubs’ or ‘mockups’. The existing work [6–10] on automated simulation testing focuses on two issues; automated stub/mock service generation [8, 11] and testbeds for service compositions [6, 7, 9, 10].

3 Multi-Objective Test Suite Minimisation for Service-centric Systems

In this section, we explain our approach and present our objective functions.
Fig. 1. Example service composition simulation and the results from the test suite execution. The table depicts test cases in the suite with their reliability, branch coverage and execution cost calculated. For the given test suite (T1,...,T6) it is expected that test cases T1 and T2 will be eliminated to get the optimal test suite (T3,T4,T5,T6) which achieves 100% coverage with lowest cost and highest reliability.

3.1 Proposed Approach and HNSGA-II

Our approach consists of two stages; test suite artefact calculation and multi-objective minimisation. After test suite generation, our approach requires the calculation of three measurements in order to apply multi-objective approaches. These are coverage, reliability and execution cost of each test case. The reliability score of a test case is based on the reliability of its inputs. Reliability scores for generated inputs are provided by ATAM (details of reliability calculation for test cases are discussed in Section 3.2).

Unlike reliability, execution cost and branch coverage cannot be acquired from an external source. The easiest way of acquiring this information is by executing the whole test suite. Unfortunately, performing a runtime test for a large suite will increase the overall cost of testing and exacerbate the negative impacts of ScST. In order to avoid this cost we propose the use of simulated testing using mock/stub services (as discussed in Section 2). Using stub/mock services allows us to find branch coverage and service invocation information for each test case without incurring additional costs of test execution on the deployed service composition.

Service invocation cost can occur in several ways based on the type of contract agreed between the provider and the integrator. Two of the most prominent payment plans used at present are; pay per-use and invocation quota based plans. As a result, two different cost calculation functions are introduced. Details of these two payment plans and the cost calculation associated with them are discussed in Section 2.
After completion of stub/mock service generation a simulated run for the test suite is performed. An illustration of an example ScS and its test suite with the test case measurement is depicted in Figure 1.

HNSGA-II is a variant of the standard NSGA-II proposed by Yoo and Harman [14]. HNSGA-II may be more effective in multi-objective minimisation problem compared to NSGA-II. HNSGA-II combines the effectiveness of greedy algorithm for set cover problem with NSGA-II’s global search. Results from the execution of additional greedy algorithm are added to the initial population of NSGA-II in order create an initial population with better solutions compared to a ‘random only’ population.

### 3.2 Objective Functions

Branch coverage is calculated as the percentage of branches of ScS under test covered by the given test suite. In our approach we considered coverage as an objective rather than a constraint in order to allow the tester to explore all the possible solutions on the pareto-optimal front. The following objective function is used for branch coverage since our aim is to maximise the coverage of the test suite.

\[
\text{Maximize} \quad \frac{\text{branches covered by test suite}}{\text{total number of branches}}
\]

The objective function for the cost is not as straightforward as branch coverage because it has multiple options for service invocation cost. As mentioned, there exist several payment plans for service usage. However, the two prominent ones are: pay per-use and quota based payment plans.

**Pay per-use plan:** In this case, the integrator is charged for each service invocation individually. The total cost of executing a test case calculated as the total cost of services invoked by the test case and is formulated as:

\[
cs(t_{cm}) = \sum_{i=1}^{n} X_{S_i} \cdot C_{S_i}
\]

where \(n\) is the number of services invoked by executing the test case \(t_{cm}\), \(S_i\) is the \(i\)th executed service, \(C_{S_i}\) is the cost of invoking service \(S_i\), and \(X_{S_i}\) is the number of times service \(S_i\) invoked by this test case.

The cost for each service can be determined by discovering available services and their prices. Since it is expected to discover multiple alternatives for each service in the composition cost element each service can be determined using one of several criteria, such as, maximum, average and minimum. The tester might choose to change the runtime service selection criteria during the test runs in order to select low-cost services to further reduce the cost of testing. However, the tester might also calculate an average or maximum execution cost for the test suite if a more realistic runtime testing is desired.
The objective function for pay-per-use plan is formulated as:

\[ \text{Minimise } \sum_{i=1}^{k} cs(tc_i) \]  

(2)

where \( k \) is the total number of test cases and \( cs(tc_i) \) is the total cost of executing the \( i \)th test case in the test suite.

Invocation quota based plan: In this case, the integrator pays a subscription fee for a number of service invocations within a period of time (such as monthly or annually). In our scenario, we presume that all the services used in the composition are selected from a single provider and a total invocation quota applies to all services rather than an individual quota for each service. The objective function for this plan is formulated as:

\[ \text{Minimise } \text{number of services invocations.} \]  

(3)

Generating test data using ATAM also enables the use of another test case artefact: reliability. Reliability of a test case is based on the reliability of its inputs. Reliability of a test input is calculated by ATAM as the combined reliability of the data sources used in generation of this input. The reliability calculation and data source selection in ATAM is discussed elsewhere [2].

Each test case might include a combination of inputs generated using ATAM and user generated inputs. In the case of user generated inputs we consider the input as 100% reliable and for ATAM generated inputs the reliability score is provided by ATAM. In the light of these possible cases a reliability function covering these two cases is formulated as:

\[ rf(in_x) = \begin{cases} 
1.0 & \text{if } in_x \text{ is user generated} \\
\text{ATAM score} & \text{if } in_x \text{ is generated using ATAM} 
\end{cases} \]

where \( rf(in_x) \) is the reliability score of the input \( in_x \).

Reliability score of each test case calculated as the average reliability of its inputs, and is formulated as:

\[ rel(tc_m) = \frac{\sum_{i=1}^{y} rf(in_i)}{y} \]

where \( y \) is the number of test inputs and \( rf(in_i) \) is the reliability of the \( i \)th input \( (in_i) \) of the test case \( tc_m \).

Reliability of a test suite is calculated as the average reliability of its test cases. Since our aim is to increase the reliability of the test suite, the objective function for test suite reliability is formulated as:

\[ \text{Maximise } \frac{\sum_{i=1}^{z} rel(tc_i)}{z} \]  

(4)

where \( z \) is the number of test cases in the test suite and \( rel(tc_i) \) is the reliability of the \( i \)th test case \( (tc_i) \) in the test suite.
4 Conclusion

In this paper, we proposed the use of optimisation for runtime testing of ScS. We focused on the cost of runtime testing, branch coverage of the test suite and test suite reliability as our three primary objectives. Future work will consider a wider range of software measurements and explore the application of other multi-objective algorithms to the problem.

References

An Analysis of the Performance of the Bunch Modularisation Algorithm’s Hierarchy Generation Approach

Mathew Hall* and Phil McMinn

Department of Computer Science, University of Sheffield, UK

Abstract. The Bunch modularisation tool can automatically modularise software systems. The level-by-level approach it uses restricts solutions available at deeper levels due to choices made at earlier levels in the optimization process, however. This paper identifies two sources of sub-optimal results with Bunch. Fitness of higher levels in the hierarchy are impacted by lower levels as a result of overfitting. Mean module fitness values are shown to be improved in approximately 30% of cases by generating non-layered hierarchies using a random mutation approach.

1 Introduction

Bunch [5] automatically remodularises software module dependency graphs using a hill climber to improve them for maintenance or comprehension purposes.

While Bunch has been observed to identify modularisations agreeable to developers [7], Glorie et al. [1] found it to be insufficient in an industrial context — parts of the modularisations were acceptable, but others were not, making the overall result unsuitable for the large remodularisation task.

Multiple levels of a hierarchy are generated in Bunch by modularising the result produced by a previous search, building module into larger modules at each step. This approach can only generate layered hierarchies; they must have uniform depth. Subsequent levels are produced by repeating the clustering on the previous level to build up clusters.

This work tests the hypothesis that Bunch produces less fit hierarchical modularisations due to its iterative approach. It does so by measuring the performance of Bunch at two levels to assess the overfitting problem, and examining existing systems and using random mutation to increase mean MQ of Bunch-generated modularisations.

The contributions of this paper are as follows:

– Quantification of the overfitting behaviour resulting from Bunch’s repeated hill climbs
– Analysis of the layered approach used and evidence that an non-layered approach would yield superior results

* Supported by EPSRC grant EP/F065825/1 REGI: Reverse Engineering State Machine Hierarchies by Grammar Inference.
2 Background: Bunch

Module dependency graphs (MDGs) represent the files or classes in a system as nodes in a graph, edges represent references between files. Bunch [7] modularises the files in an MDG using a hill climber to locate the most appropriate assignment of modules. The assignment is evaluated by the “Modularisation Quality” fitness function, which is based on the ratio of edges within a cluster to edges leaving or entering it for each cluster [5].

![Diagram](https://via.placeholder.com/150)

**Fig. 1.** Iterative hierarchy generation in Bunch; each part is a search, from the bottom level in part (a) to the top level in part (f)

Bunch can perform hierarchical clustering to produce a set of modularisations level-by-level from the bottom up. Illustrated in Figure 1, the algorithm clusters its own output (starting with iteration 1 in part (a)): clusters from the previous iteration become nodes in a new graph and the process repeats. This produces a set of levels which can be chosen from based on the abstraction required; the default level used is the median ((c) in the example). The bottom-up, repeated hill climb approach to hierarchy generation makes Bunch subject to two potential pitfalls; overfitting and restriction on balanced hierarchies.

3 Overfitting

The repeated approach can have a compounding effect, multiplying the loss in fitness due to the hill climber reaching sub-optimal solutions. For example, in Figure 1, the search has more freedom to explore solutions in part (a) — subsequent searches ((b) – (f)) depend on the initial modularisation in (a). A
negative effect on fitness at these subsequent levels is likely if the modularisation in (a) overfits or produces a sub-optimal solution. This is true for all subsequent searches; i.e. if (d) is sub-optimal, (e) and (f) will be affected.

The effect of repeated hill climbs from the lowest level is quantified by running Bunch on 30 graphs, consisting of 13 MDGs extracted from various software systems and 17 finite state machines, of which 3 are synthetic.

Each case study is clustered with Bunch a total of 30 times, each producing a hierarchy, each level having an MQ value associated with it. Hierarchies with less than two levels are rejected before the MQ values at the lowest level (level 0) and the level above (level 1) are compared.

![Fig. 2. Violin plot and density plot of normalised MQ values, N = 900](image)

Figure 2 shows the results graphically (data is available at [2]). The normalised MQ values for level 0 and level 1 for each result for each case study, calculated by \( \text{Norm}(x) = \frac{x - \text{Min}}{\text{Max} - \text{Min}} \) are shown in both parts.

Part (a) shows the spread of normalised MQ values for each level. The shape of the distribution for level 1 is different compared to the more ideal shape taken by the data set for level 0. A “perfect” result would be that both sets have the same high density around 1.0 and low density below 1.0.

The density plot in Part (b) shows the dependency of level 1 values on level 0 values. Ideally, this graph should have concentrated darkness in the upper right corner. There are clear “hotspots” in the corners, where one level is maximised while the other is minimised, however the distribution is concentrated around higher level 0 MQ values.

These results identify a clear disparity between normalised fitness values at each layer (Part (a)) and a dependency of fitness value at later levels on previous levels (Part (b)). This confirms the hypothesis that repeated hill climbs may produce suboptimal results.
4 Unbalanced Hierarchies

The repeated hill climb approach restricts the reachable solutions to balanced hierarchies, i.e. the depth of one branch of a node is equal to the depth of the other branches of that node. As will be shown, software hierarchies are rarely balanced in practice, which means the modularisations produced by Bunch are unlikely to be similar to those an expert would produce.

![Diagram of unbalanced hierarchy](image)

**Fig. 3.** (a) Unbalanced vs. (b) balanced hierarchy: some parts of (a) are represented as submodules in (b)

Figure 3 illustrates an unbalanced hierarchy (Part (a)) and the closest possible representation reachable by Bunch (Part (b)). It shows two small modules at a lower level (gdbstub and debug core, audit and file) which Bunch would not be able to produce.

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Mean Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collections</td>
<td>4.71</td>
</tr>
<tr>
<td>epubcheck</td>
<td>4.00</td>
</tr>
<tr>
<td>JDOM</td>
<td>2.63</td>
</tr>
<tr>
<td>WiQuery</td>
<td>5.16</td>
</tr>
<tr>
<td>ZXing</td>
<td>4.79</td>
</tr>
</tbody>
</table>

**Table 1.** Mean depths of classes in the each case study

The mean depth of each class in the hierarchy of 5 diverse Java projects is shown in Table 1. Non-integer values indicate an unbalanced hierarchy, i.e. the total hierarchy is not organised into a set of layers. The majority have unbalanced hierarchies, which Bunch cannot produce.

4.1 Improving Bunch Results with Unbalanced Hierarchies

Each of the subject systems used in Section 4 was clustered in Bunch 30 times. A random module was removed from the hierarchy 100 times for each Bunch result, recording mean MQ values ($\sum MQ_{\text{modules}}$) before and after. Increases in mean MQ indicate the deleted node contributed negatively to the overall quality of the modularisation, implying that the unbalanced hierarchy is superior to the balanced hierarchy.
Figure 4 shows the proportion of mean MQ values increased as a result of the random deletion operation (data is available at [2]). Improvement is observed for all case studies approximately 30% of the time. Considering these results and the findings in Section 4, this makes the case for the use of a modularisation algorithm which is not dependent on layered hierarchies.

A parallel coordinate plot in Figure 5 shows a random sample of 150 mutants from the 15000 produced by the experiment. Lines with an increasing gradient signify an improvement; the plot illustrates the spread of changes in values (improvements or negative changes) for each of the case studies.

5 Related Work

Other studies have focused on single level performance; a comparison of Bunch with the ACDC clustering algorithm [11] on simulated systems found that Bunch produced the most authoritative decompositions for two of the three systems tested [10]. However, ACDC has also been found to produce more authoritative hierarchies than Bunch [9] although Bunch produced partitionings which cut a higher number of edges correctly than those produced by ACDC.

Bunch has been shown to produce modularisations acceptable to domain experts in several cases [7], and reliably produces highly similar modularisations on repeated runs [6] at one level. Using other search techniques has been found to produce higher MQ values at the cost of computation time [4,8].

The use of supervised modularisation has also been suggested as a means to improve results obtained from automated modularisation tools [3].

6 Conclusions and Future Work

This work identifies some problems likely to be encountered when using Bunch to produce hierarchical modularisations. It demonstrates that overfitting occurs
at the lower levels of the hierarchy, impacting subsequent levels, including the level used for presentation (the median).

The layered hierarchy restriction is found to be unsuitable in most of the 5 software systems surveyed, showing that hierarchical modularisation requires un-layered approaches in most cases, which Bunch does not use.

A more directed approach could be used to refine the random deletion approach used in Section 4 to produce improvements in mean MQ more often. Future work is concerned with the possible use of delta debugging to achieve this.

Acknowledgements

The authors would like to thank Steve Swift for providing MDGs used in the study.

References

**Embrace the New Trend in SBSE with Fitness-Landscape Based Adaptive Evolutionary Algorithm**

Guanzhou Lu¹, Jinlong Li², and Xin Yao¹

¹ School of Computer Science, University of Birmingham, UK
² School of Computer Science, University of Science and Technology of China, China

**Abstract.** Search Based Software Engineering (SBSE) has seen its success in the last decade. A new trend has recently emerged to explore the potential of SBSE in adaptivity, and to achieve the grand vision of self-optimised software systems. This paper proposes a Fitness Landscape-Based Adaptive Algorithm, for tackling software engineering problems with complex problem structure or dynamic features.

**1 Introduction**

Search Based Software Engineering (SBSE) successfully links the optimisation techniques to software engineering, it has proved to be a widely effective approach, with applications over the entire software development cycle. With the recent advances in the field of software engineering, an increasing need for adaptive techniques arises in order to deal with complex problems with dynamic features. Furthermore, Harman et al. [1] proposed Dynamic Adaptive SBSE to move the research agenda forward to address the long-standing, grand challenge of self-adaptive systems.

As software engineering problems become more complex and dynamic, so does the task to tune performance- crucial parameters for search heuristics. A core goal of research in Dynamic Adaptive SBSE is to develop a framework algorithm that provides greater adaptivity to cope with problems with complex structure, or in dynamically changing environment. To create an adaptive approach to SBSE, a natural starting point is, therefore, to investigate the applicability of existing adaptive approaches in the literature of optimisation techniques. Apparently the problem of developing an effective adaptive algorithm has long been recognised, in general two classes of adaptation methods have been distinguished [2]:

- Self-adaptation: the values of the operator probabilities are directly encoded in the representation of the individual solutions.
- Adaptive Rules: the search strategy is dependent on the state of the EA run.

Self-adaptation is particularly applied for numerical optimisation. Adaptive rules are more interesting for SBSE as most software engineering problems are
discrete optimisation problems. Looking at the literature it becomes clear that most of the existing adaptive rules belong to the probability matching type [3][4][5][6]. Alternative rules include Adaptive Pursuit Method [2] and the operator selection mechanism using the Multi-armed Bandit Paradigm [7]. Previous work have seen some success along the years like in [8], however, they all work with a static set of candidate operator settings, whilst adapting the probabilities of applying them, therefore limited the adaptivity provided to deal with complex and dynamic problems.

This paper is focused on addressing the above issue, by introducing a variable set of candidate operator settings to allow greater adaptivity. Furthermore, we propose a framework algorithm with a novel mechanism in adapting the operator settings, and apply them in the unconventional evolutionary search. The major contributions of the proposed adaptive algorithm are:

- For the first time, we introduce fitness landscape analysis into adaptive algorithm for wiser decision-making mechanism along the evolutionary progress.
- The variable set of candidate operator settings is introduced, allowing greater adaptivity of the algorithm and offering a wider range of selections to better identify the best suited operator for the local landscape.
- An unconventional strategy of applying genetic operators to search is proposed.

This paper proceeds as follows. Section 2 describes the proposed algorithm in detail. Section 3 discusses the potential applications of the proposed algorithm in SBSE. Section 4 concludes the paper.

2 The Fitness Landscape-Based Adaptive Evolutionary Algorithm

It has long been acknowledged that the choice of operator settings has a significant impact upon evolutionary algorithm (EA) performance. However, finding a good choice is somewhat of a black art. In contrast, there is evidence, both empirical and theoretical, that the most effective operator settings do vary during the course of an EA run [3]. The problem of devising such a schedule to adapt the operator settings along the search progress, is almost certainly as hard as finding a good static set of operator settings. One key problem concerns with the criterion used to judge the performance of candidate operator settings at a given point in the search process.

Most previous mechanisms assume operators that produced better offspring in the past are supposed to perform better in the future, but the emphasis here is the potential ability of operator to improve upon the fitness. This demand brings the notion of fitness landscape to our attention, which reflects the actual situation being searched along the evolutionary progress. An appropriate measure of the fitness landscape can serve as a good metric for evaluating different operator settings during the decision-making process as the search proceeds.
2.1 A Measure of Fitness Landscape

Evolvability is a concept directly linked with problem hardness for EAs, which is defined as the capacity of genetic operators to improve fitness quality for a given problem [9]. Lu et al. [10] proposed a measure of evolvability properties of fitness landscape, which is used as the metric for evaluating different operator settings in our adaptive algorithm. Here we briefly review the definitions of the measure.

**Escape Probability** One of the factors that may influence problem hardness for EAs is the number of steps required to escape particular set of individuals, which might be quantified by Escape Probability. To make the study of Escape Probability applicable in practice, we adopt the idea of transition probability in a Markov chain. Let us partition the search space into $L + 1$ sets according to fitness values, $F = \{f_0, f_1, \ldots, f_L | f_0 < f_1 < \cdots < f_L\}$ denotes all possible fitness values of the entire search space. $S_i$ denotes the average number of steps required to find an improving move starting in an individual of fitness values $f_i$. The escape probability $P(f_i)$ is defined as follows:

$$P(f_i) = \frac{1}{S_i}.$$  

The greater the escape probability for a particular fitness value $f_i$, the easier it is to improve the fitness quality. From this perspective, the escape probability $P(f_i)$ is a good indication of the degree of evolvability for individuals of fitness value $f_i$.

**Fitness-Probability Cloud** We can extend the definition of escape probability to be on a set of fitness values. $P_i$ denotes the average escape probability for individuals of fitness value equal to or above $f_i$ and is defined as:

$$P_i = \frac{\sum_{f_j \in C_i} P(f_j)}{|C_i|}, \text{ where } C_i = \{f_j | j \geq i\}.$$  

If we take into account all the $P_i$ for a given problem, this would be a good indication of the degree of evolvability of the problem. Fitness-Probability Cloud ($fpc$) is defined as:

$$fpc = \{(f_0, P_0), \ldots, (f_L, P_L)\}.$$  

**Accumulated Escape Probability** To quantify the evolvability properties of fitness landscapes, A numerical measure called Accumulated Escape Probability ($aep$) is defined based on the concept of $fpc$:

$$aep = \frac{\sum_{f_i \in F} P_i}{|F|}, \text{ where } F = \{f_0, f_1, \ldots, f_L | f_0 < f_1 < \cdots < f_L\}.$$
In general, $aep$ should classify problem hardness in the following way: the larger the value of $aep$, the higher the evolvability is, and therefore the easier the problem should be for an EA.

### 2.2 The Unconventional Evolutionary Search

**Variable Set of Candidate Operator Settings**

Existing adaptive approaches pre-define a static set of candidate operator settings and only adapt the probabilities of applying them. This is insufficient in that it is not always likely that the best suited operator settings belong to the original set, therefore the limited adaptivity creates obstacles for revealing the best suited operator setting from a wider range.

In the proposed algorithm, we put forward a strategy in updating the initial set of candidate operator settings. At each generation, right after evaluating the operator settings, local search is carried out on selected operators which modifies the associated probabilities, or even the operator. For the sake of revealing best suited operator, local search is carried out on two operators with best performance indicated by the measure of fitness landscape as well as the worst one. This is to achieve the balance between exploration and exploitation, in other words, the search both exploits the area around the best solution to further improve the suitability; and, for the sake of diversity and to prevent premature convergence, it still explores around the worst solution. The set of candidate operator settings is then adapted using the above strategy.

Traditional EAs work in a way that genetic operators are applied once to each individual in the population at each generation, then the new population is generated through selection. As for the Fitness Landscape-based Adaptive Algorithm proposed, we modified the conventional evolutionary search in a way that the computational overhead incurred by the measure of fitness landscape is minimised, whilst retaining the effectiveness of the search heuristic. Basically, the structure of the proposed algorithm is described in Algorithm 1.

### 3 Possible Applications in SBSE

A few early attempts[11][12] have emerged in the SBSE community, developing hyper heuristics for tuning the performance of system by identifying performance critical parameters. The work shows that we can identify and tune performance parameters. If we can do this off-line, why not perform this tuning on-line [1]?

The Fitness Landscape-based Adaptive Algorithm is an immediate response to the above call, and shows some promise in tackling complex and dynamic software engineering problems. To start with, we list a few possible applications in SBSE below.
**Algorithm 1:** Fitness Landscape-Based Adaptive Algorithm

```
begin
    Select an initial set of operators \( C = C_{\text{ini}} \) from the candidate set \( OP \).
    Population \( P \).
    for \( i = 0, i < |C|, i++ \) do
        for \( j = 0, j < |P|, j++ \) do
            counter = 0;
            while \( f_{os} > f_{os}\&\&\text{counter} > \text{threshold} \) do
                Apply operator \( C_i \) to \( \text{Ind}_j \) to obtain offspring \( os \).
            end
            If \( f_{os} > f_j \), replace \( \text{Ind}_j \) with \( os \) in the population.
        end
        Obtain the new sub-population \( \text{SubP}_i \). Compute Measure \( m_i \), without extra neighbourhood expansion evaluation.
    end
    Selection to generate new Population \( P \) from \( |C| \) number of \( \text{SubP} \). Based on the Measures, apply local search on both best and worst operators to generate new operator settings and adapt the set.
end
```

**Dynamic Web Service Composition** Web services are self-described applications which can be invoked across the web. However, a user’s request might not be satisfied by a single Web Service, there should be a possibility to combine existing Web Services to fulfil the request.

To develop an automated method for Web Service composition is highly complex. The complexity comes from the following sources. First, the large and ever-increasing number of Web Services. Second, this method works in a highly dynamic environment where Web Services can be created and updated on the fly, thus the method needs to respond to the change at runtime. Third, user requests can vary widely and keep changing as well.

The dynamic nature of the Web Service Composition makes the application of the proposed Fitness-Landscape Adaptive Algorithm promising.

### 4 Conclusion

There has been few adaptive algorithms proposed in SBSE. This paper shows the Fitness Landscape-based Adaptive Algorithm that aims to address software engineering problems with complex problem structure, or with dynamic features. The next step is to evaluate the algorithm against possible applications.
Acknowledgement

This work was partially supported by the EPSRC grant (No. EP/D052785/1) and the EU FP7 grant (No. 257906).

References

Development of Software Project Schedule Using Heuristic Optimization

Sophia Nóbrega¹, Sérgio Ricardo de Souza¹, Marcone Jamilson Freitas Sousa¹, Geraldo Robson Mateus²

¹Federal Center of Technological Education of Minas Gerais (CEFET-MG)
30510-000, Belo Horizonte-MG, Brazil
sophianob@gmail.com, Sergio@dppg.cefetmg.br,
marcone@iceb.ufop.br

²Federal University of Minas Gerais (UFMG)
31270-901, Belo Horizonte-MG, Brazil
mateus@dcc.ufmg.br

Abstract. One of the main activities of project management is the development schedules, which includes teams and allocate specific tasks to scale software engineering. The development schedule is a complex task that depends on the experience of the project manager, not depending on the process. Trying to solve these issues this project aims to solve the problem of schedule development using metaheuristic MOVNS (Multi-Objective Variable Neighborhood Search).

Keywords: MOVNS, Project Management, Development Schedule

1 Introduction

Project management is directly related to the objective of ensuring the attainment of the objectives of organizations. Experienced project managers understand the intrinsic difficulty of managing complex projects involving large teams. Resource allocation and scheduling tasks with resource constraints are crucial steps in the process of schedule development. The quality of this schedule has a great influence on the success of software development project [1].

Resources Allocation Problem (RAP) [2] and Resource-Constrained Project Scheduling Problem (RCPSP) [3] are well known problems and often studied separately. Both belong to class of problems NP-Complete.

The Project Scheduling Problem for Software Development (PDS) is a problem even more complex because RAP and RCPSP are studied together with the objective of determining the best allocation of resources, and best execution tasks order. PDS complexity justifies use of optimization heuristics for efficient development schedule.
Like other software engineering problems, PDS is highly related to the new field of software engineering, called SBSE (Search-Based Software Engineering) [4].

The RCPSP, one of problems studied as part of the PDS, can be modeled as a scheduling tasks problem. In the literature the algorithm MOVNS (Multi-Objective Variable Neighborhood Search) has been applied as a method of solving several scheduling tasks problems, getting good performance, as in [5].

The use of MOVNS for solving PDS is motivated by its good performance when applied to scheduling task problems. Furthermore, there is no work, at least our knowledge, making PDS solution through the use of MOVNS. The MOVNS algorithm is a metaheuristic based on a simple principle: local search process carried out in a neighborhood structure with systematic changes [5].

2 Related Works

The authors of [6] published the first article using SBSE to solve project management problems. They introduced the SPMNet (Software Project Management Net) approach for solving task scheduling and resource allocation. SPMNet uses an automatic technique based on genetic algorithms to determine optimal resources allocation and calculate total time and project cost. In [7] the authors published a new paper proposing improvements in the genetic algorithm used in SPMNet, dramatically reducing the complexity of search space by providing or optimal solutions or very close to the optimal resource allocation and scheduling tasks.

In [8] the authors propose the use of an SPS (Software Project Simulator) based on evolutionary algorithm that uses a set of management rules for assessing the suitability of a solution and help the project manager in his planning. In [9] genetic algorithms, simulated annealing, random search and hill climbing were used to solve allocation problems. In this study the authors also consider problems of rework and abandonment of projects, important and common aspects of software engineering projects. In [10] the authors propose use of sparse search algorithms to solve project planning software problem, minimizing project duration.

In [13] the authors present the first multiobjective formulation for this kind of problem, in which the strength and time of completion of the project are treated as two competing objectives to solve the problem of project planning software. The proposed algorithm was tested on four real projects, and the results indicate the success of the proposed approach. In [11] the author used a multiobjective genetic algorithm to solve allocation problem of teams and development schedules. The proposed fitness function seeks to minimize total project time, total cost, tasks delay and work overtime.

3 Problem Definition

The proposed algorithm receives as input parameters tasks and human resource. Tasks own as attributes estimated effort, level of importance, start and end dates. Human resources are divided into outsourced employee and employee. The first has as attrib-
utes cost per hours worked and daily dedication. The employee has the attributes salary, daily dedication, work overtime costs and maximum time of work overtime. Both types have an attribute that represents your availability calendar.

Each human resource has a list of individual skills with their respective level of proficiency, and a list of tasks with their respective level of experience. As each task has a list of skills needed for their execution. Teams are defined by allocating resources to tasks. Each resource is allocated to a particular task in percentages ranging from 0 to maximum allowed for resource. This assignment is always made with discrete values which vary by 25%.

To define interdependencies between tasks proposed approach uses four concepts of relationship used by most project management tools: Begin-Begin (BB), Begin-End (BE), End-Begin (EB) and End-End (EE). Once those human resources generate costs, it is desirable that teams definition to minimize project costs.

Team productivity prod,t can be obtained by formula

\[ prod_{r,t} = x_{r,t} \times r_\text{Ef} \times (\prod_{s \in S^r \cap S^t} r_\text{Pr}(s)) \times r_\text{Ex}(t), \]

where \( x_{r,t} \) represents percentage effort from resource \( r \) on task \( t \), \( r_\text{Ef} \) daily dedication of resource \( r \) in hours, \( S^r \) skill set that has resource \( r \), \( S^t \) skill set required by task \( t \), \( r_\text{Pr}(s) \) adjustment factor of resource \( r \) according to their proficiency in skill \( s \), and \( r_\text{Ex}(t) \) adjustment factor of resource \( r \) according to their experience in task \( t \). Task duration in days \( t_{Du} \) can be obtained by formula

\[ t_{Du} = \frac{t_\text{Ef}}{\sum_{r \in R} prod_{r,t}} \times le \times co, \forall t \in T, \]

where \( t_\text{Ef} \) is effort of task in Function Points (FP), \( le \) and \( co \) are multipliers to adjust duration due to learning, and communication overhead, respectively. For more information about multipliers, consult [13].

Estimated each task execution time, you can define project schedule, calculating duration for each task \( t_{Du} \) considering their inter-dependencies.

Functions: (1) minimizes total project time, (2) total cost minimize, (3) minimize makespan. Makespan is the total length of the schedule (that is, when all task are performed); mathematically represent three objective functions proposed. Table 1 presents functions legend.

<table>
<thead>
<tr>
<th>I = set of tasks</th>
<th>J = set of teams</th>
<th>T = set of time periods</th>
</tr>
</thead>
<tbody>
<tr>
<td>( J_i ) = set of teams that can perform task ( i )</td>
<td>( D_i ) = set of tasks that must be performed before task ( i )</td>
<td>( t_{ij} ) = execution time of task ( i ) by team ( j )</td>
</tr>
<tr>
<td>( c_{ij} ) = development cost of task ( i ) by team ( j )</td>
<td>( d_i ) = Deadline for delivery task ( i )</td>
<td>( w_i ) = cost of task ( i ) delay</td>
</tr>
<tr>
<td>( a_i ) = delay of task ( i )</td>
<td>( s_j ) = end tasks of team ( j )</td>
<td>( S = makespan )</td>
</tr>
<tr>
<td>( x_{ij} = 1, ) if task ( i ) is allocated to resource ( j ) in period ( t ). Otherwise 0.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Legend of Objective Function

\[ A = \sum_{i \in I} (w_i \times a_i) \quad (1) \]
\[ C = \sum_{i \in I} \sum_{j \in J} \sum_{t=1}^{|I|-t_{ij}} (c_{ij} \times x_{ij}) \quad (2) \]
\[ S = makespan \quad (3) \]
4 Proposed Approach

4.1 Algorithm Description

MOVNS aims to determine a set $D$ of dominant solutions (an approximation of the Pareto-optimal). Initially, a solution $s$ is randomly generated that will be part of the set of nondominated solutions $D$. Neighborhood structures $\{V_1,V_2,...,V_t\}$ are also defined, being $V_k(s)$ the $k$ neighborhood of the solution $s$, $1 \leq k \leq t$. At each iteration, a solution $s$ of the set $D$ and a neighborhood structure $V$ are chosen. The solution is selected as visited and cannot be selected in the next iteration, if it remains in the set $D$. If the stop condition of the algorithm has not been satisfied and all solutions are selected, then all solutions will be cleared.

Algorithm: MOVNS($t$)

- $s =$ Build a random initial solution
- $D =$ \{s\}; Set of nondominated solutions
- Define a set of $t$ neighborhoods $\{V_1,V_2,...,V_t\}$
- While Stop_Condition = false do
  - $s =$ Choose a solution of $D$;
  - Mark($s$) = true; // solution visited
  - $V =$ Randomly choose a neighborhood;
  - Randomly determining a neighboring solution $s'' \in V(s)$; //shaking
  - For each neighbor $s'' \in V(s')$ do
    - $D =$ Solution set nondominated $D \cup \{s''\}$;
  - End For
- End While
- Return $D$;
- End MOVNS;

Fig. 1. Pseudocode of metaheuristic MOVNS-based

4.2 Solution Representation

Each solution $s$ of the problem is represented by a bidimensional array called allocation matrix and a vector called schedule, which stores the sequence of tasks. One dimension of matrix represents available human resources $\{r_1, r_2,...,r_R\}$, and other dimension represents tasks that must be executed$\{t_1,t_2,...,t_T\}$. Each position of allocation matrix is filled by decision variable $x_{r,t}$, defined as percentage effort from resource $r$ on task $t$. 
The schedule vector has length T, where T is the total number of tasks. The indices of the vector represent the execution order of tasks, and each vector position is filled by an integer that represents one of the project tasks.

4.3 MOVNS Proposed

MOVNS proposed is implemented using as stop condition a pre-established computational time, and two neighborhoods structures (t = 2). Neighborhoods are defined in vector schedule as follows:

V₁: Neighborhood insertion a neighbor of s is generated by inserting a task in position i, in another position j of sequence, 1 ≤ i, j ≤ n, i ≠ j.

V₂: Exchange neighborhood: a neighbor solution of s is generated by making the exchange of two tasks i and j of the sequence, 1 ≤ i, j ≤ n, i ≠ j.

Neighborhoods are defined in allocation matrix as follows:

V₁: Neighborhood insertion a neighbor of s is generated by inserting an allocation that is in position (r,t), in another position (r',t') of allocation matrix.

V₂: Exchange neighborhood: neighbor solution of s is generated by making the exchange between two columns of matrix allocation.

Three solutions called s₁, s₂ and s₃ are initially generated to start MOVNS execution. The generation of each solution is prepared in two steps. In the first step the allocation matrix is generated randomly, and then tasks duration of each solution is calculated. In the second step the vector schedule is generated using a greedy heuristic that performs the sequencing of each solution in order of priority. s₁ and s₂ are built prioritizing tasks that are part of the project’s critical path. s₃ is constructed prioritizing tasks having shorter deadline. The three solutions are evaluated and generated non-dominated solutions are stored in the set D.

The proposed approach allows generation of invalid solutions that are penalties, so that valid solutions are always prioritized. This disadvantage will be made during the evaluation of solution. The application of greedy heuristics for dealing with resource allocation constraints is proposed to reduce invalid solutions. Whenever a solution is changed by one of these heuristics, it is necessary to review it.

The MOVNS builds project schedule defining the beginning and end of tasks, scheduling the selected activity as soon as possible whilst, where possible, restrictions on resource allocation and precedence.

5 Conclusions and Future Work

Our research is focused on solving the PDS using MOVNS algorithm in order to minimize: total project cost, makespan and delay tasks. Currently, we are in the process of implementing MOVNS-based algorithm. After implementation is complete we will evaluate and validate research with empirical studies. Case Study and Experiment will be the empirical research strategies used. The algorithm proposed will be tested using real projects data.
6 Acknowledgments

The authors acknowledge FAPEMIG, CNPq and CEFET-MG for supporting the development of this research.

References

11. F. Colares. (2010). Alocação de equipes e desenvolvimento de cronogramas em projetos de software utilizando otimização. Dissertação de mestrado, UFMG.
A Comparison of Evolutionary Algorithms and Ant Colony Optimization for Interactive Software Design

Christopher Simons and James Smith

Department of Computer Science and Creative Technologies
University of the West of England, Bristol BS16 1QY, United Kingdom
{chris.simons, james.smith}@uwe.ac.uk

Abstract. Evolutionary algorithms have a well-documented history of success in Search-Based Software Engineering, but arguably the preponderance of evolutionary algorithms stems from history rather than as a conscious design choice of meta-heuristic based on the characteristics of the problem at hand. This paper sets out to examine the basis for that assumption, taking as a case study the domain of object-oriented software design. We consider a range of factors that should affect the design choice including representation, scalability, and of course, performance, i.e. that ability to generate good solutions within the limited number of evaluations. We then evaluate Evolutionary Algorithms and Ant Colony Optimization with a variety of representations for candidate solutions. Results show that after suitable parameter tuning, Ant Colony Optimization is highly effective and out-performs Evolutionary Algorithms with respect to increasing numbers of attributes and methods in the software design problem. However, when larger numbers of classes are present in the software design, an evolutionary algorithm using an integer-based representation appears more scalable.

Keywords: Ant Colony Optimization, Evolutionary Algorithms, Software Design

1 Introduction

Search-Based Software Engineering (SBSE) is now a well-established discipline wherein search has been applied across the range of the software development lifecycle [1]. Increasing research focus has been directed recently to the upstream stages of the software design, where meta-heuristic search of design spaces such as the object-oriented modeling of design classes has used metrics relating to coupling and cohesion as fitness functions [2], [3]. However, in the early stages of the software development lifecycle, the software designer has many competing factors to balance. In this situation, the role of search is therefore to enable the exploration of the search
space to discover useful and relevant software designs, and so provide insight into the
design task at hand. Here the precise balance of factors affecting the subjective judg-
ments of the human software designer is less well understood – hence the oft-heard
references to the “art” of software design. Indeed, this is precisely the sort of scenario
in which Interactive Evolutionary Algorithms (IEAs) have been shown to perform
well (see e.g. the survey of Takagi [4]). Our earlier work demonstrates that we can
indeed successfully use meta-heuristics to provide computational support for an inter-
active software design process, evolving object-oriented class models that meet de-
signers’ criteria –both subjective [5] and aesthetic [6].

As with most papers in the field, such interactive design search uses an Evolution-
ary Algorithm (EA) [7] because of their long history of successful applications. How-
ever, as the name of the field of Search-Based Software Engineering suggests, poten-
tially any search algorithm could be used, although in practice research effort has also
tended to concentrate on meta-heuristics, in particular Evolutionary Algorithms. It is
appropriate that we challenge adoption of a technology based on history, and examine
whether other search methods might be better suited to some, if not all, interactive
design search tasks. Indeed the same argument has been made for SBSE in general:
“We must be wary of the unquestioning adoption of evolutionary algorithms merely
because they are popular and widely applicable or because, historically, other re-
searchers have adopted them for SBSE problems; none of these are scientific motiva-
tions for adoption.” [8].

The contribution of this paper is to compare evolutionary algorithms with ant colo-
ny optimization in order to make a more informed choice for an underlying search
engine for interactive search of the object-oriented software design search space. Sec-
tion 2 outlines object-oriented software design, while section 3 describes the search
algorithms investigated. The experimental methodology is described in section 4 and
the results of investigations are revealed in section 5. Finally in section 6, we con-
clude by contrasting the performance characteristics of evolutionary algorithms and
ant colony optimization in object-oriented software design.

2 Object-Oriented Software Design

Software design involves the identification of concepts and information relevant to
the design problem domain under investigation. Using the object-oriented paradigm,
such concepts and information are expressed using the ‘class’ construct, where indi-
vidual instances of classes are known as objects. These classes and objects have cru-
cial relevance to subsequent downstream software implementation and testing. The
Unified Modeling Language (UML) [9] is the standard modeling language of the
object-oriented paradigm, and is widely used by software designers to visualize and
specify classes as well as other aspects of software designs. Using the UML, classes
are groupings of attributes (i.e. data that need to be stored, computed and accessed),
and methods (i.e. units of execution by which objects communicate with other objects
or indeed with human users, other programs etc.) Thus early lifecycle software design
involves finding an appropriate grouping of attributes and methods into classes.
3 Search Algorithms

Two candidate representations have been chosen to investigate search of the software design search space. In the first representation, which we shall call “naïve grouping” (NG), the genotype is a sequence of \( d \) integers from the set \( \{1, \ldots, c\} \), (where \( c \) is the maximum number of classes allowed), with an allele value of grouping \( g_i = j \) being interpreted as putting element \( i \) into class \( j \). The search space is of size \( c^d \), but there is considerable redundancy in the genotype-phenotype mapping since as far as the class model is concerned; the label applied to a class is irrelevant. The second representation is that of a graph inspired by the Travelling Salesman Problem (TSP) and Vehicle Routing Problem (VRP) [10] which we shall call Extended Permutation (XP). In this graph, candidate solutions are represented as permutations of a set of \( (e+c) \) elements, where \( e \) are the attribute and method elements and the extra \( c \) elements are interpreted as “end of class” markers akin to a “return to depot” in a VRP instance.

It is generally understood in software engineering that designers strive for high cohesion in classes (to reflect a clear purpose) and low coupling between objects (to ensure the design is robust yet flexible to change). Therefore, the measure of the structural integrity chosen is inspired by the “Coupling Between Objects” (CBO) measure [11]. Regardless of the representation chosen, each candidate solution is decoded into a set of classes, and the CBO is calculated as the proportion of all uses of attributes by methods that occur across class boundaries. This is expressed as a maximization function \( f_{\text{CBO}} = (1.0 - \text{CBO}) \times 100 \), so that \( f_{\text{CBO}} = 100.0 \) for a completely de-coupled design (all uses occur inside classes) and 0.0 for a completely coupled design.

The evolutionary algorithm chosen for comparison uses deterministic binary tournaments for parent selection and a generational replacement model ensuring the search is comparable to ACO. To the NG representation is applied random uniform mutation with either One-Point or Uniform crossover. For the XP representation, we used Order-based crossover [12] and “Edge Recombination” [13]. The former preserves the relative order of elements (as per scheduling type problems) and the latter preserves adjacency information (as per Travelling Salesman Problem or Vehicle Routing Problem). Following the scheme in [14] a single extra gene is used to encode for one of a set of possible mutation rates. During mutation, first the encoded value is randomly reset with probability 0.1, and then a mutation event occurs in each locus with the encoded probability.

The ant colony optimization has been implemented as described in [15] and uses the XP representation described above. Each ant creates a solution by visiting elements (attributes, methods or “end of class”) in turn, choosing each element probabilistically according to a combination of the attractiveness (\( \alpha \)) of pheromone trails (laid down by previous ants) and heuristics. After the whole population (colony) has created tours, all pheromone trails are subject to evaporation at a constant rate (\( \sigma \)). Finally, for each link traversed in each of the trails, a small amount (\( \mu \)) of additional pheromone is laid down proportional to the fitness of the trail in which it occurred.
4 Methodology

Unfortunately, benchmark software design problems do not appear readily in either the research literature or industrial repositories. Therefore, three real-life software design problems have been selected for use. While it is not possible to precisely assess how representative these might be of the software design field as a whole, both the second and third problems have been drawn from fully enterprise scale industrial software developments, and are decidedly non-trivial in size and complexity. The first test design problem relates to a Cinema Booking System (CBS), and comprises 16 attributes and 15 methods. The second test design problem relates to the Graduate Development Program (GDP) recording system deployed at the University of the West of England, UK, and comprises 43 attributes and 12 methods. Lastly, the third design problem relates to a nautical holiday booking system, Select Cruises (SC), which comprises 52 attributes and 30 methods. Specifications of the three test design problems can be found at [16]. Manual, hand-crafted designs are available for the three test design problems [17], and $f_{CBO}$ values are 84.6%, 70.3% and 54.8% for CBS, GDP and SC respectively.

Parameter values are drawn from the literature (e.g. [7], [15]). All search runs use a fixed number of classes – the same as in the manual design solution to provide comparability. Manual designs comprise 5, 5 and 16 classes for CBS, GDP and SC respectively. To ensure repeatability of results, we made 50 runs for each test, i.e., each combination of algorithm, problem, encoding, and parameter values. Each run is allowed to continue until either one million solutions were evaluated, or a software design with fitness 100.0 was discovered. For each run we recorded fitness (the values of $f_{CBO}$ for best solution found) and speed (the number of solutions evaluated before this best solution is first discovered).

5 Results

5.1 Evolutionary Algorithms

Results showed that for every problem-representation pairing, the use of self-adapting mutation leads to the discovery of solutions with significantly higher fitness than any of the fixed mutation rates, without any significant penalty in terms of the number of evaluations taken. Furthermore, when using self-adaptive mutation neither the choice of tournament size nor of crossover probability (within the range 0.2-0.8) made any significant difference to the $f_{CBO}$ values.

Results reveal and statistical analysis confirms that the NG (One point and Uniform recombination) representation leads to the discovery of better solutions than XP (Edge or Order recombination). For the SC problem, the difference is typically 50%. Factoring out the effect of problem instance, there is not a significant difference between Uniform and One Point recombination for the NG representation. However, with the permutation-based XP, use of the Order crossover discovers higher quality solutions than Edge Recombination.
5.2 Ant Colony Optimization

Results of parameter tuning are as follows:

- $\alpha$: performance increases as $\alpha$ increases from 0 to 1.0 – 1.5 but tails off thereafter;
- $\mu$: performance increases as $\mu$ increases from zero to 3.0;
- $\rho$: little effect for CBS and GDP, but for SC performance increases as $\rho$ increases from 0 to 1.0.

In terms of the time taken to reach the best design solution, analysis shows that some degree of pheromone decay (i.e. $\rho > 0$) is necessary to achieve fast performance (at higher values of $\alpha$ and $\mu$). This suggests that a degree of pheromone decay is crucial in exploiting the search space by making the algorithm able to ‘forget’ design solutions of poor fitness.

5.3 Comparative Analysis

After parameters have been tuned, Table 1 shows a comparison of the $f_{CBO}$ fitness of the three search algorithms while table 2 shows the number of generations / iterations required to achieve the $f_{CBO}$ values stated in table 1. Both tables relate to a population / colony size of 100. In most cases the meta-heuristics create solutions with lower coupling than manual designs. Interestingly, the NG landscape appears to be more amenable to EA search than XP, yet ACO produces better fitness values for CBS and GDP but not SC. This suggests that the EA with the NG landscape is more robust to increased scale in terms of the number of classes in the design.

Table 1. Comparison of Mean Best Coupling ($f_{CBO}$).

<table>
<thead>
<tr>
<th>Design Problem</th>
<th>Manual Design</th>
<th>EA (XP)</th>
<th>EA (NG)</th>
<th>ACO</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBS</td>
<td>84.6%</td>
<td>82.10%</td>
<td>88.80%</td>
<td>90.00%</td>
</tr>
<tr>
<td>GDP</td>
<td>70.3%</td>
<td>77.46%</td>
<td>88.07%</td>
<td>96.26%</td>
</tr>
<tr>
<td>SC</td>
<td>54.8%</td>
<td>42.68%</td>
<td>67.74%</td>
<td>49.76%</td>
</tr>
</tbody>
</table>

Table 2. Comparison of the Number of Generations / Iterations required.

<table>
<thead>
<tr>
<th>Design Problem</th>
<th>EA (XP)</th>
<th>EA (NG)</th>
<th>ACO</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBS</td>
<td>400</td>
<td>350</td>
<td>27</td>
</tr>
<tr>
<td>GDP</td>
<td>350</td>
<td>200</td>
<td>34</td>
</tr>
<tr>
<td>SC</td>
<td>1000</td>
<td>800</td>
<td>90</td>
</tr>
</tbody>
</table>
6 Conclusions

Comparison of evolutionary algorithms and ant colony optimization reveals that with respect to solution fitness, both approaches produce software design solutions of fitness values superior to those of the hand-crafted design solutions, except ACO for the SC problem. The EA with the integer NG representation emerges as the clear favorite in terms of performance for high numbers of classes, and with self-adaptive mutation, is far more robust to parameter settings. However, if a wholly interactive search is required for designs with smaller numbers of classes but higher numbers of attributes and methods and a limited computational budget, then a very different picture emerges. In this case, ACO discovers higher quality solutions, and in less time than the EAs.

References

Generic Software Subgraph Isomorphism

Jerry Swan¹, Mark Harman², Gabriela Ochoa¹ and Edmund Burke¹

¹University of Stirling, Stirling, FK9 4LA, SCOTLAND.
²University College London, London, WC1E 6BT, UK.

Abstract. The omnipresence of software graphs as useful intermediate representations means that the identification of near-match subgraphs (Error-Correcting Subgraph Isomorphism) has diverse and widespread applications in software engineering, such as querying, clone detection and model checking. Each software engineering subarea has developed specific tailored approaches to subgraph isomorphism, thereby reducing comparability and generality, and potentially yielding sub-optimal results. We introduce a generic tabu-search formulation, facilitating the incorporation of software engineering domain knowledge and present initial results for our approach.

1 Introduction

Software graphs are a ubiquitous abstraction, used throughout software engineering as a basis for representation, analysis and change. The problem of matching one graph structure against another thus has many applications. In its most general formulation, we seek an Error-Correcting Subgraph Isomorphism (ECSI) [6], which can be represented as a sequence of edit operations that transform an attributed digraph \( G_1 \) into \( G_2 \). Possible edit operations include the deletion, insertion, or substitution (i.e. attribute change) of nodes or edges, the combination of which is sufficient to transform one arbitrary graph into another. For software engineering (as with other applications), edit operations are unequal; some have greater significance, depending on the context and the problem in hand. This can be catered for by a cost function which assigns different costs to each edit operation. The optimisation problem is thus formulated as a search for the minimum cost sequence of edit operations that transforms \( G_1 \) into \( G_2 \).

In this paper we propose to adopt this formulation of subgraph isomorphism for software engineering graph isomorphism problems using tabu search. We argue that our approach can incorporate domain knowledge, making it generically applicable to many different software engineering application areas and present initial results that suggest that our approach is practical. We also hope that our research agenda will lead to approximate and partial solutions, trading precision, completeness and performance, thereby opening up new possibilities for scalable approximate solutions for software subgraph isomorphism previously considered intractable.

Space does not permit a full description of the different applications of our approach, so in the remainder of this introduction we outline some of the many such problems, thereby motivating the need for a more generic approach to software subgraph isomorphism.
Graph Based Querying: Software graphs are ubiquitous, being used to represent, for example, abstract syntax trees, call graphs, control flow graphs and dependence graphs [2] and their analysis [5, 15]. Having defined a graphical representation of the system, it is naturally to seek to query it in terms of sub-graph isomorphism. For example, Consens et al. [8] search for structural patterns in software graphs using a query editor, while Mendelzon et al. [21] describe a system that searches software graphs for design-patterns [11]. Paul and Prakash [23] argue that some form of graphical representation is essential for such querying. All such software graph query problems also require a generic facility, able to accommodate arbitrary match criteria, such as domain-specific wildcards. Graph based models are also widely used in Model Driven Engineering [1, 7]. Sadly, as is common in other growing disciplines, standards and representation formats proliferate, motivating the unification of different software engineering representations and approaches. This unification will widen the generality, abstraction level and applicability of techniques such as SBSE [16]. Common representations and problem formulations, such as subgraph isomorphism, will further help to support such a ‘round-trip’ management of the entire software lifecycle.

Graph Based Clone Detection: There has also been much recent interest and development of clone detection techniques [26]. Many clone detection techniques rely upon software graphs as intermediate structures. When such software graphs are used, as is common practice, for example, in the work of Raghavan et al. [25], Liu et al. [20]) and Krinke [19], clone detection is naturally formulated as a subgraph isomorphism problem, either explicitly or implicitly. Furthermore, subgraph isomorphism is a natural way to identify the (more challenging) so-called Type III and IV clones[26], which inherently involve concepts of edit distance. A scalable approach to subgraph isomorphism may also overcome some of the previously reported issues [17] with graph-based clone detection techniques.

Model Checking: Model checking is an umbrella term for the verification of properties of systems, for which subgraph isomorphism is important. In almost all model checking formulations, some form of graphical representation of the system’s state expansion is adopted. SBSE has been used in previous work on model checking and graph isomorphism has formed a part of this previous work [14]. For example, Alba and Chicano [3], use an Ant-Colony algorithm to find refutations of safety properties for concurrent systems benchmarks, while Flum and Grohe [10] show that model checking with first-order formulae is equivalent to subgraph isomorphism. Ghamarian et al. [12] also present case studies using the GROOVE¹ model checker for model-to-model transformation, verification of communication protocols and simulation prototyping (of an ant-colony model), while Kant [18] shows that working modulo equivalence of canonical forms can speed up isomorphism detection in larger graphs.

As this brief overview reveals, there are many applications of subgraph isomorphism in software engineering, yet each of the applications described above

¹ groove.cs.utwente.nl
uses different representations, leading to a proliferation of subgraph isomorphism techniques and a lack of results that report on comparisons, both between and within subareas of application. In the remainder of this fast abstract we take some initial steps towards addressing this problem.

1.1 A Generic ECSI Framework using Reactive Tabu Search

Determining the existence of a (proper) subgraph isomorphism was proved to be NP-complete in [9]. Since ECSI is concerned with obtaining the least cost such isomorphism, it is clearly also NP-complete. The problem has been studied extensively [6] and approaches have been proposed using both local search and continuous optimisation methods, the latter including techniques such as eigen-decomposition [28]. To be of widespread use in software engineering, any SBSE approach to subgraph isomorphism must use a computational search that can simultaneously cater for large graphs, while also seamlessly incorporating those domain-specific aspects that are pertinent to the software engineering application area. It is frequently the case in optimisation that the ability to incorporate domain knowledge is the deciding factor in making a problem tractable. Therefore we seek a scalable SBSE approach that easily facilities domain specific software engineering problem knowledge.

Messmer suggests that his “decision tree of subgraphs” algorithm [22] is the local search method of choice for larger graphs. However, this algorithm requires exponential space [6], rendering it unsuitable for software graphs. We therefore elect to use a tabu-search approach, since this allows the ready injection of domain knowledge into any of several aspects of the solution process, while offering the potential for a fast, space-efficient (yet adaptive) local search. Our formulation in terms of costed (weighted) edit operations allows domain specific information to be captured using the selection of suitable edit weights, while the specific adoption of tabu search further facilities the incorporation of domain knowledge, in three primary ways: a) in the permissible attribute equivalences; b) in the mappings prohibited by the ‘permanent tabu’ list; c) in the choice of ‘tabu-state’ by, for example, prohibiting certain edit operations for a period of time. Our chosen solution methodology extends the approach adopted in [24] and employs the variant known as reactive tabu search [4].

The contemporary presentation of Tabu Search is due to Glover [13]. The basic mechanism (termed recency-based memory) maintains a restricted local neighbourhood by prohibiting the choice of a neighbour if (some attribute of) that neighbour has been encountered recently. The simplest (or fixed-tabu) implementation implements the recency structure as a sequence of the last $k$ states encountered, where $k$ is the tabu-tenure. In [24], Petrovic et al. employ a tabu-search based approach to graph matching (specifically in the context of retrieving cases from a large case database) with encouraging results. Their strategy has two parameters that determine short-term tabu-tenure. Although these parameters can vary adaptively, the range in which they may do so is nonetheless itself a fixed attribute of the system. A more flexible method is to use metrics obtained from a dynamical systems analysis of the search trajectory. The essential idea
of reactive tabu search is to use dynamical systems metrics to drive a pair of control mechanisms that dynamically alter the search parameters and recognize when the search lies in the attractor of an unpromising region. We employ the same representation of solution state as [24], i.e. the match between $I_1$ and $I_2$ is determined by a map $m$ from $V_1$ into $V_2 \cup \perp$, where $\perp$ represents an unmapped input vertex. Solution state may consequently be represented by a list of pairs of vertices of length $\max(|V_1|,|V_2|)$. We also share their neighbourhood structure, in which for each pair of distinct vertices $(v_1, v_2) \in V_1 \times V_1$ we create a new mapping $m'$ by swapping $m(v_1)$ and $m(v_2)$. For each element of $m$, there are one or more implied edit operations. For example, if $v$ has attribute $a$ and $m(v)$ has attribute $b$, then the associated edit operation is the replacement of $a$ with $b$. Other edit operations are analogous, and we adopt the convention that in the case of the map element $v \mapsto \perp$, the operations removing the edges incident to $v$ are also implied. In contrast to the implementation in [24], which is concerned with maximizing a similarity measure, the objective of our search is to find the mapping $m$ that minimizes the cost of the implied edit operations.

2 Experimental Results

We have implemented a version of the framework in C++ and tested it on a family of automata representing the behaviour of part of a computer algebra system [27]. Fig. 1a) shows a factorization of substructure common to two automata (commonality is represented by the large circles, which denote the nesting of several thousand states), as obtained by ECSI. Execution timings as determined on a Pentium® IV PC (3.4 GHz, 2MB, Windows XP) are given in Fig. 1b). Edit costs are chosen to support incorporation of domain-specific knowledge: here, the cost of mapping an accepting to a non-accepting state was set to a prohibitively high value. We believe that the observed linearity is due in part to this incorporation of domain knowledge but in greater part to an initial construction phase that greedily matches vertices having an equivalent set of out-edge attributes.

3 Conclusion

We have outlined some of the possible applications within SBSE for subgraph isomorphism and described a generic framework for finding ECSIs that is flexible in its ability to incorporate domain knowledge for reasons of both efficiency and domain-specificity. Initial investigations reported here indicate that the approach is scalable when there is common substructure to be exploited. Future work will involve the application of the framework across other software engineering domains, in which less structural commonality may be present.

References

Fig. 1. a) Factorisation of automata obtained via ECSI b) Execution times by states


20. Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *In the Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD’06)*, pages 872–881. ACM Press, 2006.


Toward a Tool for Search-Based Testing of Simulink/TargetLink Models

Benjamin Wilmes
Berlin Institute of Technology
Daimler Center for Automotive IT Innovations (DCAITI)
Ernst-Reuter-Platz 7, 10587 Berlin, Germany
benjamin.wilmes@dcaiti.com

Abstract. An increasing demand for innovation, combined with the pressure of tight development cycles, makes the automation of testing activities ever more desirable for the automotive industry. Within model-based software development for embedded systems, initial case studies have shown that both structural and functional testing of real Matlab/Simulink models can be successfully automated via search-based testing. In order to integrate search-based testing into industrial development projects, we are working toward a tool called TASMO for both structural and functional automated testing of TargetLink-compliant Simulink models. For TASMO to be widely accepted in practice, improving efficiency and tackling shortcomings of the search-based approach is a must. Our tool concept, as well as first techniques aimed at raising efficiency, are introduced in this paper.

Keywords: Search-Based Testing, Simulink, Automotive, Tool

1 Introduction

Model-based development of embedded system software is well established in automotive industry. Such software is typically designed with Mathwork’s Simulink (SL) [1] while software code is generated from a SL model. For this purpose, the code generator TargetLink (TL) [2] is the prevalent choice in industry. While the constructive part of the model-based approach has been extensively tailored to fit the needs of embedded software development, we still see an unmet need for adaption of analytical approaches to model-based paradigms, especially in terms of testing.

In order to detect faults in embedded systems as early as possible, models would ideally be tested. Since testing models often has a lower priority than higher-level testing and time is limited in most industrial projects, the automation of this procedure would be of great benefit. In the industrial context, search-based testing has been applied in many cases to the automated generation of test data for SL models. It transforms the test data finding problem into an optimization problem and utilizes meta-heuristic search techniques like evolutionary
algorithms to solve it [3]. The approach has been used for both structural (white-box) and functional (black-box) testing of industrial-sized SL models [4].

Structure-oriented search-based test data generation for SL, as explored by Windisch [5] as well as Zhan and Clark [6], has delivered promising results in industrial case studies [7]. Nonetheless, the standard approach lacks efficiency when applied to larger SL models and shows difficulties targeting Boolean states and tackling complex dependencies within models. Likewise, function-oriented search-based testing of SL models has also demonstrated that it can be a useful complement to conventional testing [8], although the approach lacks acceptance in practice due to the necessity of manually defining a fitness function.

Working toward tool support for both structural and functional testing automation for TL-compliant SL models, we set our sights on overcoming, or at least mitigating, these shortcomings. The structure-oriented application is to be supported by integrating static techniques in the first instance. Subsequently, we intend to use the resulting structural test data generation procedure for functional model testing by treating the violation of a model property (which is also defined using the SL language) as a structural coverage goal.

## 2 Background: Search-Based Testing of Simulink Models

SL is a graphical data-flow language for specifying the behavior of dynamic systems. Syntactically, a SL model is composed of functional blocks and lines connecting them, while most of the blocks are equipped with one or more input ports as well as output ports. The semantics of such a model results from the composed functionalities of the involved block types, e.g., sum blocks, relational blocks or delay function blocks. In addition, event-driven or state-based functionalities can be realized within SL models using Stateflow (SF) blocks. A SF block contains an editable Statechart-like automaton.

In the course of implementing software graphically using SL, researchers and practitioners have noticed opportunities to test earlier, concomitant with the implementation activity on the model level. In fact, industrial standards like ISO 26262 recommend testing the models, e.g., for conformance of model and resulting code. Search-based testing [3] can be used to automate various model testing activities. In short, it employs meta-heuristic search algorithms in order to find test data satisfying a search goal. The search is led by a fitness function, whose task is to rate the generated test data according to the goal.

In functional (requirement-oriented) testing, which is the primary quality assurance activity in industrial development processes, test cases are derived systematically, though usually manually. While most such tests are used to check if the model or system under test exhibits the expected behavior, search-based functional testing can be used specifically for finding test cases that provoke unwanted behavior. Additionally, the search-based approach is able to automate structural testing, i.e., finding test cases that execute and thereby cover internal structural elements of the test object, with respect to selected coverage criteria. Each coverage goal (CG) is targeted with a separate search and accompanied by
a specific fitness function. In contrast to the function-oriented approach, which requires a sophisticated fitness function to be manually designed, such a function can be derived automatically from the test object’s internal structure.

Windisch [5], and in parallel Zhan and Clark [6], adapted the search-based approach for performing structural testing of SL models. The work of Windisch also supports coverage criteria for SF diagrams [9]. For particularly covering SF transitions, Oh et al. investigated the use of messy genetic algorithms [10]. While applying search-based structural testing to SL, Windisch identified two fundamental differences compared to the application on code level. First, as SL models describe time-dependent behavior, the test data to be generated should be regarded as data sequences (signals) in order to reach every possible model state. When generating such complex test data with common search algorithms, Windisch recommends compressing the data structure [11]. His segment-based signal generation approach takes into account the need for being able to specify what test data signals are generated, e.g., amplitude bounds and signal nature, like wave or impulse form. Second, Wegener’s fitness function approach for code [12], consisting of approach level and branch distance, cannot be fully adopted. SL models have no execution paths because their execution involves the execution of every included block, generally at each time step. A CG-related fitness function therefore addresses only distances to the desired value of one or more model internal signals. For CGs in SF diagrams, however, a bipartite fitness approach is possible. In both cases distance values are time-related, i.e. a fitness function has to calculate its result based on distance signals.

### 3 Tool for Automated Structural and Functional Testing

Continuing the work of Windisch [7], we are working toward a tool for search-based testing of SL models, which performs acceptably and reliably in industrial development environments. This tool is called TASMO (Testing via Automated Search for Models). Real TL-compliant SL models from current development projects at Daimler, e.g., drivetrain related models of electric vehicles, will serve as case examples to evaluate further technical improvements and for practical tests of the tool. Our work concentrates on TL-compliant SL models as the code generator TL is widely utilized in automotive industry. TL extends SL by offering additional block types, while also excluding certain SL constructs or block types.

Apart from automated structural testing, TASMO will also support automated functional testing of such models. To this end, our approach is similar to that of Tracey at code level [13]. In his work, assertions that are inserted in the test object’s code are targeted with search-based techniques. Likewise, for testing SL models, the user may define a model property over selected (internal) signals and the search engine is then used to detect a violation of the property. Boström and Björkqvist, who investigated such an application of search-based techniques, suggest a first order predicate logic for expressing properties over SL signals [14]. In contrast, we intend to define properties using a subset of the SL language and perform the search for property violations structure-oriented.
Fig. 1. User interaction workflow of our tool for search-based testing of SL models

From a user’s viewpoint, TASMO is organized as shown in Figure 1. The functionalities of automated structural testing and searching for a property’s violation are directly available from within the SL editor. Depending on whether a subsystem of the SL model is currently selected/opened or not, the invoked TASMO is testing either the subsystem or the entire model. The model structure, as well as other model information, is gathered internally using the Matlab API.

Before performing a search, TASMO enables the user to specify the test data (input signals) to be generated, as proposed by Windisch [7]. As a result, the utilized search algorithm generates test data which solely complies with the signal specification. Additionally, further constraints on the signals to be generated can be defined and will be considered as a parallel search goal [15].

In the case of structural testing, the user has to select one or more coverage criteria, or certain CGs directly. TASMO will support decision coverage, condition coverage, modified condition/decision coverage (all of them for both SL and SF), lookup table coverage (for SL), state coverage and transition coverage (both for SF). An existing test suite can be imported in order to automatically deselect already-covered CGs. When functional testing is performed with TASMO, the model property must be defined in a model-based way, and its inputs must be referenced with the corresponding model signals. In both use-cases, the progress of the following search will then be visualized. Finally, TASMO delivers a report and provides generated test data in a reusable format.

4 Improving Practicability

Search-based testing has been successfully applied to structural test data generation for (proprietary) SL models originating from development projects at Daimler, e.g., a model of a windshield wiper controller [7]. In comparison to purely random test data generation of similar complexity, the search-based approach provides notably higher model coverage. Even when compared with a commercial tool, the search-based approach performs more effectively. Yet, despite these
promising results, it lacks efficiency, especially in cases of large, complex models. Since we want TASMO to perform solidly in industrial development environments, our work focuses on identifying and overcoming these shortcomings.

As a first step, we developed three static techniques that extend the standard search-based approach by analyzing the model under test before search processes for each CG are run. Unsatisfiable CGs are identified and excluded from the search by deploying an analysis of the model signal’s ranges. To reduce search space, a dependency analysis determines which model inputs a CG actually depends on. At code level, McMinn et al. [16] proposed a related approach. We also developed a technique that sequences CG-related search processes in order to maximize collateral coverage and reduce test suite size. Note that Fraser and Arcuri [17], as well as Harman et al. [18], recently addressed this problem too, but in a different way. For more details we refer to a recent paper of ours [19].

Further shortcomings hindering the search-based approach at model level were identified. Among others, the search might face a partial blindness in cases of complicated dependencies in the model under test (cf. [20]). Stimulating the model inputs with generated test data and measuring certain distances at some other point in the model leads to a disregard of any structural information between. Beyond that, the problem of targeting Boolean states in a model still remains. A fitness function capable of adequately leading the search is hard to find in such a case [7]. Even though Zhan and Clark proposed a technique [21] which mitigates this problem in certain situations, further work is necessary since industrial SL models often contain Boolean problems that cannot be traced back to non-Boolean ones without reasonable effort.

5 Conclusion and Future Work

While search-based techniques have been applied to testing SL models with promising results, as of yet, no suitable tool for industrial use has been developed. This paper presents our concept for such a tool and provides insight into its underlying implementation. Furthermore, deficiencies of search-based testing in application to structural testing of SL models are disclosed. First additional techniques for improving efficiency of the automated test data generation are proposed as well. Solutions addressing other shortcomings are part of future work. In particular, the hybridization of search-based testing with static techniques is in our interest. Extensive case studies with SL models from the automotive domain will follow in the course of our proceeding tool development.

References

19. Wilmes, B.: Automated structural testing of Simulink/TargetLink models via search-based testing assisted by prior-search static analysis. Submitted to: 4th Int. Conf. on Advances in System Testing and Validation Lifecycle (2012)