SBSelector: Search Based Component Selection for Budget Hardware

Lingbo Li(✉), Mark Harman, Fan Wu, and Yuanyuan Zhang

CREST, Department of Computer Science, University College London, Malet Place, London WC1E 6BT, UK
lingbo.li.13@ucl.ac.uk

Abstract. Determining which functional components should be integrated to a large system is a challenging task, when hardware constraints, such as available memory, are taken into account. We formulate such problem as a multi-objective component selection problem, which searches for feature subsets that balance the provision of maximal functionality at minimal memory resource cost. We developed a search-based component selection tool, and applied it to the KDE-based application, *Kate*, to find a set of *Kate* instantiations that balance functionalities and memory consumption. Our results report that, compared to the best attainment of random search, our approach can reduce at most 23.70 % memory consumption with respect to the same number components. While comparing to greedy search, the memory reduction can be up to 19.04 %. *SBSelector* finds a instantiation of *Kate* that provides 16 more components, while only increasing memory by 1.7 %.

1 Introduction

Using Component Based Software Engineering (CBSE) [6], a new software edition (or instance) can be developed by composing pre-existing components, each of which contributes new functionalities to the system. In an ideal world, we would simply include all components, thereby yielding maximal functionality. However, in practice, resource constraints need to be taken into account. In this paper, we focus on the resource constraint of memory consumption.

There are many component selection methods that use an iterative selection/rejection model to filter components based on pre-defined rules/criteria (i.e., stakeholders' requirements) or expert experience [4]. From Requirements Engineering perspective of view, the component selection problem is also known as the Next Release Problem (NRP) [1,10], which can be addressed using search-based techniques. Previous work on SBSE formulations of component selection [2] proposed a single-objective NRP model to select components, later Zhang et al. [12] introduced Multi-Objective NRP (MONRP), and Li et al. [9] extended MONRP with a simulation-based approach to address uncertainty. Kwong et al. [8] also demonstrated how NRP can be re-deployed for multi objective component selection. In their work, selecting highly rated components and the coupling and cohesion relationships among components were considered as improving optimisation objectives.

© Springer International Publishing Switzerland 2015
M. Barros and Y. Labiche (Eds.): SSBSE 2015, LNCS 9275, pp. 289–294, 2015.
DOI: 10.1007/978-3-319-22183-0_25

In this paper, we develop and implement a tool *SBSelector*, which uses a multi-objective SBSE approach to component selection, and apply it to the large, real world system *Kate*, a text editor written in C++. *Kate* is a configurable multi-platform text editor [7]. It can be extended by 'plug-in' type components to enrich its functionality. Some plug-ins are written in native C++, while others are written in Python. There is a special C++ plug-in called Pâté that switches on/off functionality to support Python-based plug-ins. There are currently 37 plugins available, yielding a component selection search space of 2^{37} candidate instances; already too many to support exhaustive exploration, thereby motivating search-based approach.

2 Component Selection as an Instance of MONRP

This section briefly outlines the problem formulation and implementation.

Objectives: There are two objectives which are taken into account. Both of them aim to maximise the users' satisfaction. The first objective is to maximise the number of enabled supplement components of *Kate*: $Maximize\ Component(\vec{x}) = \sum_{i=1}^{n}(x_i)$. In general, the more components are integrated, the more features are available for users to increase their satisfaction. The second objective is defined as minimising the worst-case memory consumption of *Kate*: $Minimize\ Memory(\vec{x}) = \max_{1 \leq j \leq m} memory(\vec{x})$. Where n is the number of components, and m is the number of simulations. The decision vector $\vec{x} = \{x_1, \cdots, x_n\} \in \{0,1\}^n$ determines the inclusion of components in the system: x_i is 1 if component i is selected and 0 otherwise. The function $memory(\vec{x})$ denotes one evaluation of memory consumption for decision vector $\vec{x}$.

Algorithmic and Implementation Details: The optimisation process is implemented in a Java-based Linux toolkit, *SBSelector*, which directly modifies the configuration file to select components. Since *SBSelector* is a component selection tool, there are no changes made to the source code of the software, making *SBSelector* easily applicable to other Linux software.

The core of *SBSelector* adopts Non-dominated Sorting Genetic Algorithm-II (NSGA-II) [3]. Initially, *SBSelector* generates and evaluates solution population P_0 with size N randomly. Each individual is a component configuration representing the selection of components. Tournament selection, single point crossover, and bit-flip mutation are then applied to reproduce a new population P_0'. Each generated offspring solution in P_0' is evaluated, and merged into P_0, which is then sorted by non-dominated sorting, thereby, producing a new population P_1 with size N for next generation. The population evolves until a termination condition is met. In our experiments, the evolution terminates when the pre-defined generation number is reached. The main evaluation process of *SBSelector* is presented in Algorithm 1.

In each simulation, *Kate* is executed for 1.5 s, and its memory consumption is measured every 100 ms through analysing the results of standard Unix utility, *ps*. *SBSelector* evaluates the dependence constraints, using a 'repair method' [11] to ensure that all dependence constraints are met.

Algorithm 1. SBSelector evaluation process

```
Require: the solution (configuration) $S$ for evaluation
  $com\_num = CountSelectedComponents(S)$
  $memories = \emptyset$
  for $i = 1, .., m$ do
      $memory = EvaluateMemory(S)$
      $memories = memories \cup \{memory\}$
  end for
  $max\_memory = GetMax(memories)$
  $SetFitnessOne(S, com\_num)$
  $SetFitnessTwo(S, max\_memory)$
  return $S$
```

3 Experiments and Results

In this paper, to evaluate the feasibility and effectiveness of *SBSelector*, we answer the following Research Questions:

RQ1 Does the extra memory consumed by enabling all plugins of *Kate* simply equal to the summation of the extra memory consumed by enabling each plugin one at a time?

We ask this question as a baseline for this work. If two plugins share some libraries, it is likely that the extra memory consumed by enabling both plugins will be less than the sum of the extra memories consumed by enabling each of them. Therefore, if we observe that the extra memory needed by *Kate* with all plugins enabled is much less than the summation of that with each plugin enabled, there might be hidden shared dependencies between these plugins. This motivates the use of search of optimisation to find the combination of enabled plugins.

RQ2 How effectively can *SBSelector* find optimised combination of enabled plugins compared to random search and a greedy strategy selection?

Without SBSE, human developers (or users) may include components randomly or greedily based on the memory consumed. We use random search as well as greedy search and compare the optimised combination of components given by NSGA-II against the result of random search and greedy search, to understand how much improvement we can achieve with search based techniques. The initial population size of NSGA-II is set to 50, and the total number of evaluations is 2500. The random search is performed as a sanity check [5], thus the total number of evaluations is the same with that of NSGA-II. Since the Greedy search is deterministic, it is executed once only.

RQ3 Given some mandatory plugins, can *SBSelector* still find combinations of optional plugins that only trade a little amount of memory consumption?

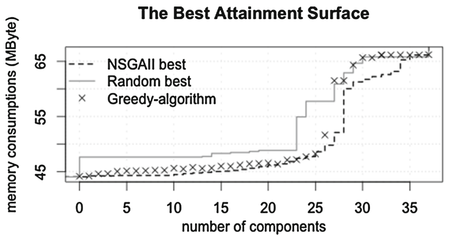
In reality, some of the components are mandatory to the software or to the user, thus can not be excluded. In this question, we want to know whether the fixed inclusion has any impact to the effectiveness of *SBSelector*. We evaluate *SBSelector* for one particular scenario ***S1***, where all Python plugins, 'Search and Replace', and 'SQL Plugin' are essential for Python developers. The result of ***S1*** is compared to scenario ***S2*** where all components are open to select.

In order to provide the experiment results in the form of statistical power, we execute our experiments for 10 times. In this case, 10 executions proved to provide a sufficient statistical power to avoid Type two errors, since the results were so strongly better than random search, the baseline against which we compared. All experiments were performed using a machine with dual Intel Core i5 3.20 GHz CPU and 4 GB RAM. The operating environment is Ubuntu 13.04 with Qt 4.8.4, KDE Development Platform 4.11.5, and KIO Client 2.0.

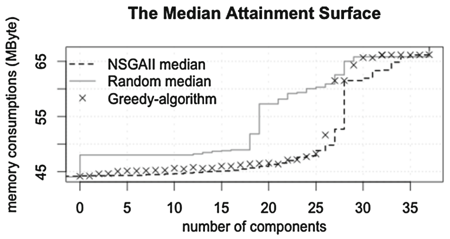
Answer to *RQ1:* The sum of all plugins' individual memory consumption is 45776.4 Kbytes, meanwhile, the extra memory consumed by *Kate* with all the plugins enabled is 22127.6 Kbytes. Specially, Pâté is the most expensive plugin (consumed 14255.6 Kbytes), while the Python program language based plugins are the cheapest. They use very little memory when they are enabled. The probable reason of this interesting finding is that, Pâté is the infrastructure of Python program language based plugins. It has to provide comprehensive invokable interface for those Python based plugins. Moreover, Python is a lightweight dynamic programming language, which means the loading memory consumed by these Python based plugins may be negligible. Consequently, enabling Pâté means nearly all required Python based underlying libraries are loaded. In summary, the result reveals that there are some plugins sharing the underlying libraries consuming less memory together. This promotes the applicability of our tool for the user without exact source code level knowledge of *Kate*.

Answer to *RQ2:* The results are plotted in two figures for two types of attainment: the best attainment (Fig. 1a) and the median attainment (Fig. 1b) surfaces for three approaches: NSGA-II, random search, and greedy search. It can be observed that, in both best and median attainments, there is an obvious gap between the attainment surfaces generated by NSGA-II and random search. The gap is considerably larger when the number of enabled plugins is between 24 and 34. Up to 23.79 % memory can be saved by proper component selection. To perform a statistics comparison between random search and NSGA-II, we use hypervolume indicator to represent the quality of the results. Wilcoxon signed-rank test is performed and its outcome denotes that there is a significant difference between the Pareto-front generated by NSGA-II and random search (p-value = 0.004, Vargha-Delaney effect size = 1). This indicates that our tool *SBSelector* outperforms the simulated human behaviour in terms of finding the solutions with more components while consume less amount of memory.

When human developers or users select components using greedy strategy, assuming they have the knowledge of the memory usages of all individual plugins, the outcome is better than random selection and close to the outcome of NSGA-II. Figure 1 exhibits that, with some basic information, greedy strategy can find, though not optimal, considerably better solutions than random search. Despite good solutions found by greedy search, NSGA-II constantly outperforms greedy search. Specially, with respect to including exact 27 components, the memory reduction from the greedy solution to the best solution found by NSGA-II is 19.04 %. In other words, without knowledge of the exact dependencies among underlying libraries, greedy strategy may mislead the software developer to sub-optimal solutions. Such loss will be amplified with the growth of the scale of the



(a) Best attainment surface



(b) Median attainment surface

Fig. 1. Answer to ***RQ2***. Comparison of 10-run attainment surfaces for NSGA-II, random search, greedy search.

problem. The Wilcoxon signed-rank test result indicates that the Pareto-front generated by NSGA-II and greedy search are moderately significantly different (p-value = 0.064, Vargha-Delaney effect size = 0.3). In Summary, comparing to simulated human behaviour, search based techniques can effectively find more memory-efficient component combinations. The amount of economized memory can be up to 23.79 %.

Answer to *RQ3:* In order to evaluate the effectiveness and applicability of our tool, we apply our tool in the scenario where *Kate* is used by a Python programmer. The result of ***S1*** and the comparison with the best attainment surface of ***S2*** is presented in Fig. 2.

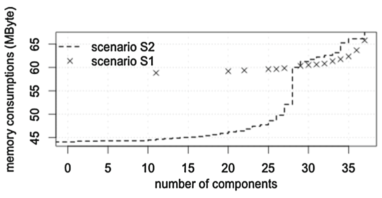


Fig. 2. Anwser to ***RQ3*** The comparison between the Pareto front of ***S1*** and the best attainment surface of ***S2***.

When enabling 11 mandatory plugins in ***S1***, *Kate* consumes 58, 840 KBytes, which is the minimum memory consumption for ***S1*** as shown in Fig. 2. As the number of included plugins gradually increases, the memory consumption of the best solutions found by NSGA-II grows insignificantly. For instance, after including exact 16 optional plugins, the memory consumption of *Kate* only increases 1.7 %. Surprisingly, when including more than 19 optional plugins, NSGA-II found better solutions in ***S1*** than that found in ***S2***. This is because making 11 plugins mandatory significantly reduces the search space, thus NSGA-II can focus more on the rest solutions and performs better at certain areas. In summary, the answer to ***RQ3*** is, *SBSelector* is applicable and effective in practise when some plugins are mandatory.

4 Conclusions

In this paper, we demonstrated that component selection problem can be treated as an instance of MONRP, and addressed it using search based techniques.

The results presented illustrate the trade-off between two types of user experiences. Moreover, our results can be used to support to further investigate the hidden implicit relationships among *Kate*'s plugins. The results also highlight some solutions that, when embedding the same number of components, our approach can reduce the memory consumption by up to 23.79 %. In one specific use case, *SBSelector* can find a solution that provides 16 more components while only increase 1.7 % memory consumption. Future work will investigate applying our tool to large scale software systems (i.e., Chrome and Firefox), and consider the topic as well as the popularity of components as an added constraint & objectives.

References

1. Bagnall, A.J., Rayward-Smith, V.J., Whittley, I.M.: The next release problem. Inf. Softw. Technol. **43**(14), 883–890 (2001)
2. Baker, P., Harman, M. Steinhofel, K., Skaliotis, A.: Search based approaches to component selection and prioritization for the next release problem. In: Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM 2006), pp. 176–185. IEEE Computer Society, Washington, DC (2006)
3. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. Trans. Evol. Comput. **6**(2), 182–197 (2002)
4. Fahmi, S.A., Choi, H.-J.: A study on software component selection methods. In: Proceedings of the 11th International Conference on Advanced Communication Technology, ICACT 2009, vol. 1, pp. 288–292. IEEE Press, Piscataway (2009)
5. Harman, M., McMinn, P., de Souza, J.T., Yoo, S.: Search based software engineering: techniques, taxonomy, tutorial. In: Meyer, B., Nordio, M. (eds.) Empirical Software Engineering and Verification. LNCS, vol. 7007, pp. 1–59. Springer, Heidelberg (2012)
6. Heineman, G.T., Councill, W.T. (eds.): Component-based Software Engineering: Putting the Pieces Together. Addison-Wesley Longman Publishing Co. Inc., Boston (2001)
7. Kate. http://kate-editor.org/. Accessed in April 2015
8. Kwong, C.K., Mu, L.F., Tang, J.F., Luo, X.G.: Optimization of software components selection for component-based software system development. Comput. Ind. Eng. **58**(4), 618–624 (2010)
9. Li, L., Harman, M., Letier, E., Zhang, Y.: Robust next release problem: handling uncertainty during optimization. In: Proceedings of the 2014 Conference on Genetic and Evolutionary Computation, GECCO 2014, pp. 1247–1254. ACM, New York (2014)
10. Zhang, Y., Finkelstein, A., Harman, M.: Search based requirements optimisation: existing work and challenges. In: Rolland, C. (ed.) REFSQ 2008. LNCS, vol. 5025, pp. 88–94. Springer, Heidelberg (2008)
11. Zhang, Y., Harman, M., Lim, S.L.: Empirical evaluation of search based requirements interaction management. Inf. Softw. Technol. **55**(1), 126–152 (2013). Special section: Best papers from the 2nd International Symposium on Search Based Software Engineering 2010
12. Zhang, Y., Harman, M., Afshin Mansouri, S.: The multi-objective next release problem. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO 2007), pp. 1129–1137. ACM, New York (2007)