Context-Aware Recommender Systems for Non-functional Requirements

Antonina Danylenko and Welf Löwe
School of Computer Science, Physics and Mathematics
Linnaeus University,
Växjö, Sweden 35155
{antonina.danylenko, welf.loewe}@lnu.se

Abstract—For large software projects, system designers have to adhere to a significant number of functional and non-functional requirements, which makes software development a complex engineering task. If these requirements change during the development process, complexity even increases. In this paper, we suggest recommendation systems based on context-aware composition to enable a system designer to postpone and automate decisions regarding efficiency non-functional requirements, such as performance, and focus on the design of the core functionality of the system instead.

Context-aware composition suggests the optimal component variants of a system for different static contexts (e.g., software and hardware environment) or even different dynamic contexts (e.g., actual parameters and resource utilization). Thus, an efficiency non-functional requirement can be automatically optimized statically or dynamically by providing possible component variants. Such a recommender system reduces time and effort spent on manually developing optimal applications that adapts to different (static or dynamic) contexts and even changes thereof.

Keywords—context-aware recommender systems; non-functional requirements; context-aware composition.

I. INTRODUCTION

To develop a fully-featured system, a system designer has to make sure that the system meets both functional and non-functional requirements. The functional requirements define *what* a system is supposed to do, while the non-functional requirements specify *how* a system is supposed to behave [1].

Non-functional requirements, such as performance, dynamic memory consumption, or memory footprint of the code, are usually considered in all phases of the project development and highly dependent on architecture and design decisions. For large systems, it becomes relatively complicated to deal with all non-functional requirements at the same time the system designer focuses on the functionality of the core of the system. Especially, when the requirements quickly emerge during the development, as normally the case in agile development processes, handling of all requirements at once can get easily out of control. For example, in order to write an efficient software that involves solving a computational problem the software

designer must carefully choose and tune the algorithms to keep the number of operations as low as possible that eventually will lead to a high application efficiency. Therefore, it is beneficial to postpone architectural decisions a system designer makes for meeting the efficiency non-functional requirements and save the effort and time spent on these decisions.

On the other side, an attempt to work out non-functional requirements at the end of the development phase can lead to a significant system rework or even to a failure of time and effort estimations [2]. At each step of the development process there is a danger that a design decision has an undesirable side-effect that eventually leads to either inaccurate or inefficient application performance. It is therefore important to make sure that architectural decisions regarding functionality do not later inhibit meeting some non-functional requirements.

The main goal of this paper is to address these problems by suggesting a *context-aware recommender system* that helps system designers and developers to create fully-featured software with less effort and within shorter amount of time. Special focus is put on the computational software with efficiency non-functional requirements, such as performance, and their portability, such as performance-portability of the software between different multi-core systems.¹

Context-aware recommender systems for software development, in general, recommend software artifacts depending on the development context, such as the task to perform [3], [4] or the code to be extended [5], [6], [7]. Our recommender system uses context-aware composition as the basic technique. This approach intends to postpone the architectural decisions to be made in order to meet non-functional requirements and allows a system designer to focus on the developing of a functionality of a software first. Context-aware composition can be used for improving an appli-

¹While this paper exemplifies the suggested context-aware recommender systems using performance requirements and performance-portability, the results generalize to other efficiency non-functional requirements and their portability.

cation's runtime performance and other efficiency non-functional optimization goals like decreasing energy or memory consumption [8], [9], [10]. Therefore, it separates the concerns of defining component variants and the decision in favor of any of these variants in a (static or dynamic) composition context. The former is done by system or component designers who develop variants that might be advantageous in certain composition contexts possibly supported by variant generators, generating, e.g., variants implementing different schedules. The latter, a recommendation system automates: a composition context is evaluated, e.g., the static system environment or the runtime resource utilization. Depending on that, the recommender system suggests the respective best-fit component.

The recommendations can be used by a software developer who writes the composition code, which statically binds to the best-fit variant in a static context, or dynamically dispatches to the best-fit variant in a dynamic context. Alternatively, this composition code can be generated automatically using, e.g., using Aspect-Oriented Programming.

The remainder of this paper is structured as follows: Section II gives an overview of our suggested context-aware recommender system and sketches the main ideas of how this system can be applied in the developing process using context-aware composition techniques. Section III gives examples and experimental results that show the improvement gained in terms of application performance. Section IV discusses related work, and Section V concludes the paper and points out directions for future work.

II. RECOMMENDER SYSTEMS USING CONTEXT-AWARE COMPOSITION

The suggested context-aware recommender system operates on the following concepts:

A **formal context** is a defined program point corresponding to a call expression (method invocations or object creations) with formal context parameters. Formal context parameters can include (abstractions and selections of) the formal call parameters (e.g., the problem size), the locally assessable system state, the data type used, the current method or a class this call expression occurs in, or even the system environment (e.g., the number of available processors).

An **actual context** is a valuation of the formal context parameters at latest before a corresponding call expression is executed.

A **component interface** is an interface of a callee that can be bound to the call expression of a formal context. Formally, it is an abstract method with pre-conditions (post-conditions) implied by (implying) the call expression's pre-conditions (post-conditions).

Figure 1. The recommender system in the software development process.

A **component variant** is a component interface implementation, a co-variant subtype of the component interface. The implementation variants range from alternative algorithms and data structures to alternative schedules or exploitations of hardware accelerators.

A **utility** is an optimization goal function that maps an actual context and a component variant to a numerical value representing the variant's non-functional attribute (performance, footprint, memory consumption etc., or combinations thereof) in that context.

Context-aware composition [8], [9] recommends the presumably best-fit component variant for *any* given actual context. Therefore, it profiles the actual utility of the variants in a *sample* set of actual contexts. Then it abstracts from these sample points using machine learning; different machine learning technologies are pluggable here [10]. Note that the selected component variant itself may define formal contexts. Hence, context-aware composition is a recursive process and the order in which decisions are made is decisive.

Figure 1 depicts the conceptual building blocks of the recommender system and the whole development process. **Profiling** tests alternative component variants in different contexts, and captures the champion variant of each context. A **Learner** extrapolates and interpolates the information received from the profiler (using machine learning) from a total mapping of contexts to champion variants. A **Recommender** predicts the presumably optimal variant for each actual context.

For profiling, test data is generated for each actual (sample) context. Profiling accesses the actual context, invokes all implementation variants, measures utility

Table I
USAGE SCENARIOS.

	Scenario (1)	Scenario (2)	Scenario (3)
Offline	profiling learning recommendation composition	profiling learning	
Online		recommendation composition	profiling learning recommendation composition

of each variant, and captures the best variant together with the actual context. The learner takes the result of profiling and, using machine learning approaches, generates a recommender.

Figure 1 shows the whole process of applying the recommender in the software development process. A **system designer** is required to define formal contexts affecting the choice of the component variants that might be advantageous in certain actual contexts. Based on this information a **software developer** implements the functions assessing the actual contexts at each corresponding call expression and the alternative component variants. After profiling and learning, the recommender system outputs a recommendation plan that can be used either by a software developer (system designer) or by a composition **code generator**.

Based on the given recommendation, the software developer (code generator) writes (generates) a **connector** which binds to the best-fit component, statically or dynamically. More specifically, the connector can be a static monomorphic call or a complex dynamic dispatch pattern. The former is preferred when the actual context is static or the recommendation for different dynamic contexts is always the same. The latter is more expensive in terms of runtime overhead, but necessary when the actual context changes at runtime and the best-fit component variant also may change. Then the connector implements a dynamical adaptation of the application behavior in an actual context. The (static or dynamic) **composition** eventually puts together the alternative component variants using the respective connector, and binds a caller in one component to a matching callee in another component.

The profiling and learning phases can happen offline, when the application runs a test regime under the supervision of a system developer, or online, at system runtime (depending on whether the actual contexts and the variants' utility can realistically be assessed statically or not). The recommendation and composition phases can happen offline or online as well, depending on whether the formal context parameters allow for a static actual context evaluation

and composition or not. Possible usage scenarios are summarized in Table I and detailed below.

Scenario (1) If the formal context (e.g., target hardware) and the variant utility (e.g., memory footprint) can be evaluated statically or if in all sampled actual contexts the same variant is recommended, the composition can be done statically, i.e., offline. Composition is implemented with a static monomorphic call.

Scenario (2) If the formal context needs to be evaluated dynamically (e.g., the problem size) and different variants are recommended for different actual contexts, the recommendation and composition are performed online, and the connector is adaptive. Profiling and learning phases can still be done offline if the actual contexts occurring online can be simulated offline.

Scenario (3) If, however, there is uncertainty about the actual contexts occurring online (e.g., the target system or the actual parameter profile are unknown or changing), it might be beneficial to postpone even profiling and learning to runtime. A special adaptive connector composes according to the (initial random) recommendation of the recommender, but it deliberately deviates from time to time from the presumably best-fit component variant. The profiler monitors the selected component variant in the deployed and running application. A learner runs in parallel to the application, aggregates monitoring results, and constantly updates the recommender. The recommender provides (better and better) recommendations to the special adaptive connector.

Notice that in all three cases, the connector can be written by a software developer or generated by a code generator.

III. EXAMPLES AND EXPERIMENTS

We present experimental results that show how performance, a non-functional requirement, in a Sorting application was guaranteed by our approach presented in the previous section.

A software designer has to define alternative sorting components (algorithms) and the formal context that influences the Sorting performance. We implemented the well-known variants of SelectionSort, QuickSort, MergeSort, and two naive parallel versions of QuickSort and MergeSort (spawning a new thread with each recursive call). Note that all recursive variants contained sort call sites themselves. The selection of the actual algorithm variant was left open for all sort call suites, even the recursive ones.

The formal contexts are a problem size N and availability of unused cores P , where N is a continuous integer sampled at powers of two ($2^0 \dots 2^{16}$), being a P boolean with 0 (no unused cores available) and 1 as

Figure 2. Sorting performance for Scenarios (1) and (2). The x-axis displays the problem size, the y-axis the time in msec.

possible values. To meet the performance requirement, we aimed at selecting the best-fit algorithm variant for each actual context. Note that in all three scenarios the connector was implemented manually by a software developer. All experiments are executed on a 2 core MacBook Pro running Mac OS X(10.6.5) on an Intel Core i5 at 2.4 Ghz and 8GB RAM.

Figure 2 shows the performance of the sorting application for a range of problem sizes using the scenarios (1) and (2). In Scenario (1) a static composition for all actual contexts always composes with the (on average) fastest single variant - sequential QuickSort. Scenario (2) performs an online composition and, therefore, selects for each actual context the best-fit variant. It gives a speed-up of about 1.46 relative to the first scenario because it starts with the parallel MergeSort when cores are available, recursively calls sequential QuickSort, and eventually, for small problems, prefers SelectionSort. For details about the experiments using the first two scenarios including other machines, we refer to [10]. For other examples, we refer to [9].

Figure 3 shows the results for Scenario (3), which performs self-adaptation at runtime. During the first iterations (calls to sort) no meaningful variants are recommended, and all variants are selected and monitored by-and-large randomly. However, while the application runs, more knowledge is gained, and the recommendations get better until they eventually approach and even outperform the performance of Scenario (2). This happens due to the sub-optimality of the offline profiling decision while in online scenario the recommender can choose another optimal algorithm variant that outperforms the offline case.

Altogether, the experiments show that our approach can efficiently speed-up the application performance by providing the recommendations on the best-fit com-

Figure 3. Sorting performance for Scenario (3). The problem size is constant $N = 100,000$, the x-axis is the number of sorting invocations, the y-axis the corresponding time in msec.

ponent variants for certain program contexts. Applying this approach in the development process can support the software developer in making sub-optimal decisions of the best component configuration.

IV. RELATED WORK

Recent approaches to recommender systems in software development have been focusing on the information overload problem that occurs due to the growing number of software artifacts a programmer should consider when working on a software project [11], [12]. Several approaches improve programmer productivity by mechanisms that filter and recommend relevant software artifacts to a particular development task. For instance, Kesten and Murphy [3] provide a task context model that shows the artifacts by ranking and filtering the information presented in the development environment. Čubanić *et. al* [4] developed a tool for recommending artifacts, including electronic media, bug report and test plans, created as part of the development process.

Another possible recommendation context can be a particular source code the developer is working in. For instance the works in [5], [6] and [13] present approaches and tools for providing relevant source code examples using the structure of the source code and for reusing the software components by actively recommending methods that are suitable in a context, respectively. Fluri *et. al* [7] suggest a recommender system for software developers that provides context change recommendations by applying automated code adaptation support leveraging changes applied in the past. Felfernig *et al.* [14], in turn, relate to a requirement engineering that can improve the overall quality of decision processes and requirement development of individual stakeholders that deal with continuously

changing and quickly ageing requirements. In contrast to these works, our approach concentrates only on efficiency non-functional requirements. However, it also intends to improve the quality of software processes and to reduce the overall costs of software projects by improving developer's productivity.

V. CONCLUSIONS AND FUTURE WORK

In this paper we addressed the problem of handling efficiency non-functional requirements during the software development process. Our work mainly focused on the design and development of a context-aware recommender system that intends to postpone the architectural decisions and allows to focus on the design of the core system functionality. The recommender system suggests the best-fit component variants for certain actual contexts which are later on used by a composition technique to improve application runtime performance. The experiments showed how the performance of the sorting application for a range of problem sizes could be improved using this recommender system. The suggested approach can help to improve the software development process (especially developing computational software) by decreasing the burden on a system designer or a software developer in terms of time and effort spent on the project.

As a matter of future work, we will extend our approach to handle memory and energy consumption requirements. The former may need trimming of memory footprint so that smaller, less-expensive memories can be used. This may be achieved by efficient data management that recommends the use of different data structures or different streaming approaches (these have to be defined by the designer). The latter may effect the power consumption of the resulting product that uses developed software. Moreover, for hand-held or wireless devices, the software's power consumption has to be optimized in order to extend the life of small, cheap power sources. The problem can be handled by recommending different component variants that proved to use less power consumption (e.g., by monitoring this consumption using different hardware utilities). Finally, we have to empirically evaluate our approach in a real industrial project environment and measure the productivity of software developer and designer in terms of programming efficiency when the decisions about non-functional requirements are handled by the recommender system.

REFERENCES

- [1] G. Kotonya and I. Sommerville, *Requirements Engineering - Processes and Techniques*. John Wiley & Sons, 1998.
- [2] R. Malan and D. Bredemeyer, "Defining non-functional requirements," Tech. Rep., 2001.
- [3] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 1–11.
- [4] D. Čubanić, G. C. Murphy, I. C. Society, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *IEEE Transactions on Software Engineering*, vol. 31, pp. 446–465, 2005.
- [5] R. Holmes, R. J. Walker, and G. C. Murphy, "Approximate structural context matching: An approach to recommend relevant examples," *IEEE Trans. Softw. Eng.*, vol. 32, pp. 952–970, December 2006.
- [6] Y. Ye and G. Fischer, "Reuse-conducive development environments," *Automated Software Engineering.*, vol. 12, pp. 199–235, April 2005.
- [7] B. Fluri, J. Zuberbühler, and H. C. Gall, "Recommending method invocation context changes," in *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, ser. RSSE '08. New York, NY, USA: ACM, 2008, pp. 1–5.
- [8] C. Kessler and W. Löwe, "A framework for performance-aware composition of explicitly parallel components," in *Parallel Computing: Architectures, Algorithms and Applications*, ser. Advances in Parallel Computing, vol. 15. IOS Press, 2008, pp. 227–234.
- [9] J. Andersson, M. Ericsson, C. Kessler, and W. Löwe, "Profile-guided composition," in *Proc. 7th Int. Symp. on Software Composition, March 2008.*, 2008, pp. 157–164.
- [10] A. Danylenko, C. Kessler, and W. Löwe, "Comparing machine learning approaches for context-aware composition," in *Proc. of the 10th Int. Conf. on Software Composition*, 2011, pp. 18–33.
- [11] J. S. Breese, D. Heckerman, and C. Kadie, "Empirical analysis of predictive algorithm for collaborative filtering," in *Proceedings of the 14 th Conference on Uncertainty in Artificial Intelligence*, 1998, pp. 43–52.
- [12] H.-J. Happel and W. Maalej, "Potentials and challenges of recommendation systems for software development," in *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, ser. RSSE '08. New York, NY, USA: ACM, 2008, pp. 11–15.
- [13] F. Mccarey, M. O. Cinnéide, and N. Kushmerick, "Rascal: A recommender agent for agile reuse," *Artif. Intell. Rev.*, vol. 24, pp. 253–276, November 2005.
- [14] A. Felfernig, M. Schubert, M. Mandl, F. Ricci, and W. Maalej, "Recommendation and decision technologies for requirements engineering," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE '10. New York, NY, USA: ACM, 2010, pp. 11–15.