Synergy between Generative Reuse and Software Product Line

Shahliza Abd Halim, Dayang Norhayati Abg Jawawi and Safaai Deris
Faculty of Computer Science and Information Systems, Universiti Teknologi Malaysia, 81310 Skudai, Johor, Malaysia
shahliza@fsksm.utm.my, dayang@fsksm.utm.my, safaai@fsksm.utm.my

Abstract

Software reuse through concrete component library has a strong support for component composition, refinement and specialization. Despite of the library's support, scalability problem occur in its implementation. Amongst the problem being reported were feature combinatorics problem, vertical and also horizontal scaling of the components. Generative reuse is an approach in software reuse where it combines reusable part that are not only code but also generic architectures and variations of components for future customization. Generative reuse via application generator is cost effective to build when many similar software systems are written or when evolution of software requires the software to be written and rewritten many times during its lifetime. Software Product Line is a suitable field to implement application generator where it can help to generate similar systems and also customize variations needed to the systems functionalities. This paper briefly discusses the issues in generative reuse via application generator and software product line. The main contribution of this paper is in the explicit mapping between generative reuse specifically the development process of application generator with the Software Product Line systematic reuse process. This explicit mapping shows several points of synergistic process between both fields. This mapping can also help designer and organization who are interested in the development of application generator in software product line to know the underlying process and artifacts for both fields.

1. Introduction

Software systems if built from scratch can result to significant duplication in processes and causing overlap result. Software component reuse is the key to significant gains in productivity where already compiled components can be plug into the software for functionality. However, with the changes and also evolution in software system, the components created for reuse increased combinatorily and it raises the scalability problem in component library.

"Feature combinatorics problem" have the effect of scalability in component library thus affecting programmers productivity. Batory [1] has studied features related to data structure, memory allocation, scheme, access mode and concurrency in C++ data structure libraries. Based on the study, features may appear in many different combinations of classes. As there is a need for a unique class for each legal combination of features, there is also a need to develop and maintain a large number of similar classes. From the study, [1] concluded that in order to have scalable library, the library must offer much more primitive building blocks and be accompanied by generators that compose blocks to yield data structures needed by application programmer [2]. The similar problem also being researched by [3] where he refers to this problem as the scaling dilemma categorized into the vertical and horizontal scaling. Vertical scaling based on the specificity of the components and horizontal scaling due to the generic variations of component has thus compromise the performance of the components and also resulting on components that are only marginally reusable in other target applications

Generative based reusability technology was the second technology discussed by [4] where they compared it to composition based reusability technology. In composition technologies, the components to be reused are atomic and unchanged during its reuse but it can be modified or changed to fit the computational requirement. Generation technologies on the other hand reused components that are not concrete where the components being reused are often pattern woven into the fabric of generator program thus the components are not concrete and self contained entity. Generation technologies are categorized into three groups, language based, application generators and transformation system. Language based generation technologies has been particularly successful in the

area of programming language systems, such as compilers, language based editing systems and static program analyzers [5]. Application generators the second group of generation technologies, translate specifications into application programs. Transformation systems, the last group of generation techniques are software tools that "rewrite" constellations of concepts (characters, strings, trees and graphs) into alternative constellations. Practical transformations systems are extremely generalized compilers where among the possible applications of transformation systems are translation of code from one language to another, refactoring and code generation [6].

In another field of research initiated by Parnas [7], software is viewed as product line where it arises situations when we need to develop multiple similar products for different clients, or from a single system over years of evolution. Members of a product line share many common requirements and characteristics [8]. With the use of application generator in the software product family system, it can maximize the automation of application development.

The remainder of this paper is organized as follows: Section 2 outlines the concept of generative reuse and application generator while section 3 highlights the concept of Software Product Line (SPL). Section 4 then describe the systematic reuse of SPL, processes involved in SPL, artifacts generated from the processes and also the tool or development method for the processes. The following section conceptualizes the mapping of application generator and SPL process. Lastly section 6 presents the conclusion of this paper.

2. Overview of Generative Reuse and Application Generator

Generative reuse is done by encoding domain knowledge and relevant system building knowledge into a domain specific application generator. New systems in the domain are created by writing specifications for them in a domain specific specification language. The generator then translates the specification into code for the new system in a target language. The generation process can be completely automated, or may require manual intervention [9].

Constructing an application generator is appropriate when many similar software systems are written, when one software system is modified or rewritten many times during its lifetime, or when many prototypes of a system are necessary to converge on a usable product [10]. In [11], application generators is viewed as soft automatic programming systems, a form of reuse approaches where it reuse patterns inside the generators.

Soft automatic programming such as GUI builders reuse the knowledge on how to translate the high level graphical specification into executable code. Compiler and parser generators reuse the knowledge of how to generate a compiler from the input grammar and generative CASE tools build class-templates from Business Object model to relational database structures.

Application generators are practical and mattractive when high-level abstractions from an application domain can be automatically mapped into executable software system [10]. A difficult challenge for implementers of application generators is defining optimal domain coverage and optional distribution of domain concepts into the fix and variable parts of the abstraction [10]. Another challenge is to create a domain specific language for the application generator. Domain specific language and application generators represent a flexible form of reuse that not only allows the reuse of the implementations of abstract functional units as in component-based approaches, but also allows the reuse of how these functional units are combined to form a complete system. Furthermore, application generator allows this knowledge to be reused by non-programmers because the domain-specific language can provide an interface to the domain-user in familiar notations [12].

In generative reuse, candidates are identified and instantiated at the modeling level rather than at the coding level and all the work necessary to integrate the customized components code into the application is done automatically by generators. In order to automate the component assembly, the application generator needs configuration knowledge to map abstract user requirements onto appropriate configurations of components. Figure 1 shows three components which are essential to implement generative reuse based on [13].

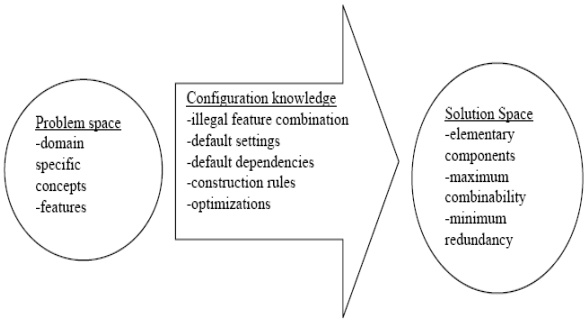

Figure 1 Components essential for generative reuse

3. Overview of Software Product Line

The goal of the software product line (SPL) approach is the systematic reuse of core artifacts for building related software products or product diversities. A software product family typically consists of a product family architecture, a set of components and a set of products. Each product derives its architecture from the product family architecture, instantiates and configures a subset of the product family components and usually contains some product specific code. Product diversification is based on the concept of variability and appears in all family artifacts where the behavior of the artifacts can be changed, adapted or extended [14]. The main focus of SPL is to model the commonalties and also the variabilities of SPL. In [15], variability represents a capability to change or adapt a system. Such a change or adaptation can affect the behavior of the system as well as it qualities. Viewed from technical perspective, variability is a means to delay a design decision to a later phase in the lifecycle of the software system.

The following section discusses the domain engineering and application engineering approach in SPL.

4. Systematic Approach to Reuse in SPL

Based on [16], the problem facing software engineering is not a lack of reuse, but a lack of widespread systematic reuse. In product-line engineering, systematic approach to reuse is to divide the engineering process into two different phases: domain engineering and application engineering as stated in [17], [18]. Domain engineering provides the reusable core assets that are exploited during application engineering when assembling or customizing individual applications. While on the other hand, application engineering configures target applications from domain engineering reusable core assets. In other word, application engineering is the instantiation process of target product in SPL.

This section will elaborate more on domain engineering because this process gives higher contributions towards systematic reuse in software product line compared to application engineering which only uses the reusable assets developed in domain engineering.

Domain engineering is most often divided into three main processes: domain analysis, domain design, and domain implementation. Domain analysis is first introduced by Neighbors to denote studying the problem domain of a family of applications [19]. It concerned with examining a variety of related applications to identify their common architectures, reusable components, design alternatives and domain oriented terminology. This information can then be expressed in terms of abstract classes and subclasses, protocols, framework constraints and inference rule [20].

The output of domain analysis is domain model. In [18] the generalization of the common artifacts or processes belonging to domain model are as follows:

- Domain scoping (domain definition, context analysis)
- Commonality analysis
- Domain dictionary (domain lexicon)
- Notations (concept modeling, concept representation)
- Requirements engineering (feature modeling)

There are researches which concentrate on developing CASE tools to produce domain models that affectively reflect the commonality and variability in SPL such as RequiLine [21] and DREAM [22].

Domain design means designing the core architecture for a family of applications. It comprises the selection of the architectural style[13], [18]. In addition, the common architecture under design should be represented using different views. The core architecture should also provide variabilities between applications. This process is to decide on how to enable this variability or configurability. According to feature models and commonality documents, it should also be selected which components or items (such as requirements) are provided in the core architecture and which items are implemented as variations in individual applications [18]. There are several methods for product line architectural design

such as COPA, FAST, FORM, KobrA and QADA. These design methods have been compared in [23].

Domain implementation covers the implementation of the architecture, components, and tools designed in the previous phase. This comprises, for example, writing documentation and implementing domain-specific languages and generators. The purpose of domain engineering is to produce reusable assets that are implemented in this phase. Thus, the result of whole domain engineering phase comprises components, feature models, analysis and design models, architectures, patterns, frameworks, domain-specific languages, production plans, and generators [18].

5. Mapping of Application Generator and SPL Process

As identified by Levy as cited in [24], there are three-step methodology for developing with application generators or referred by him as metaprogramming.

i) identifying the requirements of the generator
ii) building the generator
iii) using the generator.

Cleaveland has refined the requirements phase into six sub phase stated in detail in [10, 24, 25]. Table 1 shows the mapping of application generator development phases with SPL development process discussed in section 4. Based on Table 1, we have classified certain activity in application generator as a suitable process in domain analysis, domain design or domain implementation. The table also shows the artifacts and briefly the development tools or methods available based on undergoing research from both fields.

SPL through systematic reuse carried in domain engineering, assist application generator in encoding domain knowledge essential for the requirement process of application generator. Case tools designed in domain analysis can be an aid in defining domain coverage and optional distribution of domain concepts for application generator development. On the other hand, SPL variability handling mechanism experienced feature combinatorics problem as reported in [26]. Application generators can overcome this problem in domain implementation where generative approach basically generative programming in the form of metaprogramming such as using template metaprogramming in C++ [13, 27] and XVCL[2, 28, 29] can be used to overcome scalability problem in SPL. Based on these examples, application generator and SPL when paired together are capable of achieving more success in reusability than they would separately.

6. Conclusion

The aim of SPL is to develop not one product but several products which share commonalties and differ through certain key variabilites. SPL also faces the scalability problem where many of the implementation approaches used in the industry for handling variability has created growth in the components created. Generative reuse is useful to overcome the scaling dilemma where the implementation approach used in application generator automates the customization of components.

Application generator and SPL has synergistic relationship when both of this field are used together it will reap the benefit in more systematic approach towards reuse and also code optimization through scalable library and configurable target application. From this mapping, it will also enable developer who is new to the concept of application generator and also software product line field to anticipate the process and product of each process in both fields. Development tools and development methods for implementation purposes have also been described briefly in the mapping. The mapping can also be used by organization interested in software product line engineering processes and want to know the tools to optimize code generation for their product using application generator.

7. References

1. Batory, D., Singhal, V.,Sirkin, M. and Thomas, J. *Scalable Software Libraries*. in *ACM SIGSOFT'93: Symposium on the Foundations of Software Engineering*. 1993. Los Angeles, California: ACM.
2. Jarzabek, S.a.S., L. *Adapting Redundancies with a "Composition with Adaptation" Meta-programming Technique*. in *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundation of Software Engineering*. 2003. Helsinki: ACM Press.
3. Biggerstaff, T.J., *A Perspective of Generative Reuse*. . Annals of Software Engineering, 1998. **5**: p. 169-226.

4. Biggerstaff, T.J., Perlis, Alan, J, *Software Reusability Concepts and Models*, ed. T.J. Biggerstaff, Perlis, Alan, J. Vol. 1. 1989: ACM Press and Addision Wesley.
5. Jarzabek, S., *From reuse library experiences to application generation architectures.* ACM, 1995 p. 114-122.
6. Baxter, I.D. *Transformation Systems: Generative Reuse for Software Generation, Maintenance and Reengineering*. 2002: Springer-Verlag Berlin Heidelberg.
7. Parnas, D., *On the Design and Development of Program Families.* IEEE Transactions on Software Engineering, 1976. **SE-2**(1): p. 1-9.
8. Zhang, H.J., S. and Yang, B., *Quality Perdiction and Assesment for Product Line.* Springer-Verlag Berlin Heidelberg, 2003: p. 681-695.
9. Frakes, W.B., Kang, K., *Software Reuse Status and Future.* IEEE Trans. On Software Engineering, 2005. **31. No 7**: p. 529-536.
10. Krueger, C., W., *Software Reuse.* ACM Computing Surveys, 1992. **24**(2): p. 132-183.
11. Goebl, W. *A Survey and a Categorization Scheme of Automatic Programming Systems*. in *GCSE'99*. 2000: Springer-Verlag Berlin Heidelberg.
12. Thibault, S.a.C., Charles. *A framework for application generator design*. in *Symposium on Software Reusability, Proceedings of the 1997 symposium on Software reusability*. 1997. Boston, United States: ACM Press New York, NY, USA
13. Czarnecki, K.a.E., U. *Components and Generative Programming*. 1999: Springer-Verlag/ACm Press.
14. Jaring, M.a.B., J. . *Variability Dependencies in Product Family Engineering*. in *PFE2003*. 2004: Springer-Verlag Berlin Heidelberg.
15. Becker, M. *Mapping Variabilities onto Product Family Assets*.
16. Prieto-Diaz, R., *Status report: software reusability.* IEEE Software, 1993. **Volume 10**(Issue 3): p. 61 - 66.
17. Macala, R.M., Stuckey, L and Gross, D., *Managing Domain-Specific, Product-Line Development.* IEEE Software, 1996.
18. Harsu, M., *A Survey on Domain Engineering*. 2002, Institute of Software Systems, Tampere University of Technology. p. 48.
19. Neighbors, J.M., *Software Constructions Using Components.*, in *Department of Information and Computer Science*. 1980, University of California, Irvine.
20. Lubars, M., D. . *Reusing Designs for Rapid Application Development.* in *ICC'91*. 1991.
21. Maßen, T.a.L., H., *RequiLine: A Requirements Engineering Tool for Software Product Lines.* Lecture Notes in Computer Science, 2004. **3014/2004**(Software Product-Family Engineering): p. 168-180.
22. Moon, M., Yeom, K and Chae, HS, *An Approach to Developing Domain Requirements as a Core Asset Based on Commonality and Variability Analysis in Product Line.* IEEE Transactions on Software Engineering, 2005. **31**(7): p. 551-569.
23. Matinlassi, M. *Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA*. in *Proceedings of the International Conference on Software Engineering (ICSE'04)*. 2004: IEEE.
24. Mili, H., Mili, F & Mili, A, *Reusing Software: Issues and Research Directions.* IEEE Transactions on Software Engineering, 1995. **Volume 21**.
25. Cleaveland, C., J., *Building Aplication Generators.* IEEE Software, 1988: p. 25–33.
26. Anastasopoules, M.a.G., C. . *Implementing Product Line Variabilities*. in *Symposium on Software Reusability. Proceedings of the 2001 symposium on Software reusability: putting software reuse in context* . 2001. Ontario, Canada.
27. Czarnecki, K.a.E., U., *Generative Programming- Methods , Tools and Applications*. 2000, Boston MA: Addison-Wesley.
28. Jun, Y.a.J., Stan, *Applying a Generative Technique for Enhanced Genericity and Maintainability on the J2EE Platform.* 2005.
29. Cheong, Y.C.a.J., S., *Modeling Variant User Requirements in Domain Engineering for Reuse.* Information Modeling and Knowledge Bases: p. 220-234.

SPL Phases	SPL Processes	Application Generator Phases	Artefacts	Development Tools/Methods available
Domain Engineering *(Aims at the development of reusable software)*	Domain Analysis	Identify requirement of the generator. Refined to steps: -Reconizing domains -Defining domain boundaries -Defining variant and invariant part -Defining the - Specification input method -Defining the product	Domain model which contains -domain scoping -domain dictionary -concept models -commonality analysis - feature models.	RequiLine DREAM
	Domain Design	Design an underlying model	-Architectural style -Core architecture with common feature and variability features	Methods for product line architectural design: COPA, FAST, FORM, KobrA and QADA.
	Domain implementation	Building the generator	Implementation of architecture, components, domain specific languages or generation tools.	-Template metaprogramming in C++ - XVCL
Application Engineering *(Aims at reusing the reusable assets developed in Domain Engineering*)		Using the generator	- Composition of components for the target application	

Table 1. Mapping of Application Generator and SPL Process