UNIVERSITI TEKNOLOGI MALAYSIA

BORANG PENGESAHAN
LAPORAN AKHIR PENYELIDIKAN

TAJUK PROJEK : GENERATIVE REUSE APPROACHES FOR COMPONENT-BASED SOFTWARE ENGINEERING

Saya DAYANG NORHAYATI BINTI ABANG JAWAWI
(HURUF BESAR)

Mengaku membenarkan **Laporan Akhir Penyelidikan** ini disimpan di Perpustakaan Universiti Teknologi Malaysia dengan syarat-syarat kegunaan seperti berikut :

1. Laporan Akhir Penyelidikan ini adalah hakmilik Universiti Teknologi Malaysia.
2. Perpustakaan Universiti Teknologi Malaysia dibenarkan membuat salinan untuk tujuan rujukan sahaja.
3. Perpustakaan dibenarkan membuat penjualan salinan Laporan Akhir Penyelidikan ini bagi kategori TIDAK TERHAD.
4. * Sila tandakan (/)

[ ]	SULIT	(Mengandungi maklumat yang berdarjah keselamatan atau Kepentingan Malaysia seperti yang termaktub di dalam AKTA RAHSIA RASMI 1972).
[ ]	TERHAD	(Mengandungi maklumat TERHAD yang telah ditentukan oleh Organisasi/badan di mana penyelidikan dijalankan).
[X]	TIDAK TERHAD	

TANDATANGAN KETUA PENYELIDIK
DAYANG NORHAYATI ABANG JAWAWI
Project Leader VOT 78094
Department of Software Engineering
Faculty of Computer Science and Information Systems
Universiti Teknologi Malaysia
81310 UTM Skudai, Johor Darul Takzim
email: dayang@utm.my

Nama & Cop Ketua Penyelidik

Tarikh : 27 November 2007

CATATAN : **Jika Laporan Akhir Penyelidikan ini SULIT atau TERHAD, sila lampirkan surat daripada pihak berkuasa/organisasi berkenaan dengan menyatakan sekali sebab dan tempoh laporan ini perlu dikelaskan sebagai SULIT dan TERHAD.*

VOT 78094

GENERATIVE REUSE APPROACHES FOR COMPONENT-BASED SOFTWARE ENGINEERING

(GUNA SEMULA GENERATIF UNTUK KEJURUTERAAN PERISIAN BERASASKAN KOMPONEN)

DAYANG NORHAYATI ABANG JAWAWI
RADZIAH MOHAMED
SAFAAI BIN DERIS
SHAHLIZA ABD. HALIM
ROSBI BIN MAMAT

RESEARCH VOTE NO:
78094

Department of Software Engineering
Faculty of Computer Science and Information System
Universiti Teknologi Malaysia

2007

ACKNOWLEDGEMENTS

We would like to extend our appreciation to the Ministry of Higher Education for funding this project and our acknowledgement to the Research Management Center, Universiti Teknologi Malaysia for their support in managing this research.

GENERATIVE REUSE APPROACHES FOR COMPONENT-BASED SOFTWARE ENGINEERING

(*Keywords: Software reuse, generator and software product line*)

Generative reuse is an approach in software reuse where it combines reusable part that 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 (SPL) 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. SPL is a type of reuse where common artifacts can be shared by similar software or members in the product line. Besides sharing common features, each member in the product line has significant variations referred as variability. Variability implementation requires focus on two important issues: delaying design decision and also ease of changes in software. This study is based on the initial proposal of two methods: stepwise refinements and separation of concerns, for the use in generator implementation. Generators with the implementation of these methods have been reportedly used in various SPL implementations. Based on this motivation we study the underlying concepts of these methods and the origin of its idea. We also study the issues and its current implementation in generator. The results of this research can help designer and researcher who are interested in the development of application generator in SPL to comprehend the underlying methods and also its usage in generator.

Key researchers:
Dayang Norhayati Abang Jawawi (Head)
Safaai bin Deris
Rosbi bin Mamat
Radziah Mohamed
Shahliza Abd. Halim

E-mail: dayang@utm.my
Tel. No.: 07-5532354
Vote No.: 78094

GUNA SEMULA GENERATIF UNTUK KEJURUTERAAN PERISIAN BERASASKAN KOMPONEN

(*Kata kunci: Guna semula perisian, penjana and barisan keluaran perisian*)

Guna semula generatif adalah satu pendekatan guna semula perisian di mana ia menggabungkan bahagian-bahagian boleh guna semula yang bukan sahaja kod perisian tetapi termasuk juga seni bina generik dan variasi komponen untuk perubahan perisian akan datang. Guna semula generatif dengan penjana aplikasi boleh menjimatkan kos bila sistem perisian yang hampir sama perlu ditulis berkali-kali sepanjang proses pembanguan perisian. Barisan keluaran perisian (SPL) adalah bidang yang sesuai untuk melaksanakan penjana aplikasi yang mana ia boleh membantu penjanaan sistem yang serupa dan juga perubahan variasi keperluan untuk berbagai keperluan sistem. SPL adalah teknik guna semula di mana artifak yang sama boleh dikongsi oleh perisian yang serupa atau ahli dalam satu barisan keluaran. Di samping berkongsi ciri yang sama, setiap ahli dalam barisan keluaran mempunyai variasi bererti yang dipanggil kebolehubahan. Perlaksanaan kebolehubahan memerlukan fokus kepada dua isu: iaitu menangguh keputusan reka bentuk dan kemudahan perubahan pada perisian. Kajian ini berdasarkan dua kaedah pelaksanaan penjana yang sediada iaitu: penghalusan berperingkat dan pemisahan usaha. Dua kaedah ini telah digunakan dalam pelbagai perlaksanaan SPL, oleh itu ia memberi motivasi kepada kami untuk mengkaji konsep dasar kepada kaedah-kaedah ini dan asal usul idea ini. Kami juga mengkaji isu dan perlaksanaannya dalam penjana. Hasil kajian ini boleh membantu pereka bentuk dan penyelidik yang berminat dalam pembangunan penjana aplikasi SPL untuk memahami kaedah dasar dan juga penggunaannya dalam penjana.

Key researchers:
Dayang Norhayati Abang Jawawi (Ketua)
Safaai bin Deris
Rosbi bin Mamat
Radziah Mohamed
Shahliza Abd. Halim

E-mail: dayang@utm.my
Tel. No.: 07-5532354
Vote No.: 78094

CONTENTS

CHAPTER I

INTRODUCTION

1.1 Overview

Software reuse is the process of creating software systems from existing software artifacts rather than building software systems from scratch. This vision was introduced by Mc Illroy in 1968. Software reuse is the use of existing software or software knowledge to construct new software. Reusable assets can be either reusable software or software knowledge (Frakes and Kang, 2005).

Biggerstaf and Perlis have divided technologies in software reuse into two groups, the composition based and generation based (Biggerstaff and Perlis, 1989). Composition based reuse in concrete library has its own drawback in terms of scaling problem when the software evolve over time. Vertical scaling resulted from the developer's tendency to add new requirement to already existing component because of there is simply lesser code to add in an already existing component and also the components already exist are free of bugs. This can save a lot of developer's time but it will affect the size of the components where it will keep increasing. This scenario will make it difficult for the component to fit in other application. Another scenario is horizontal scaling where in order for the components to have high potential to fit in the target application the developer tends to create variations of the components. While this horizontal scaling gives the components wider variability for usage, it will be only marginally reusable and not really fit well in the target application because of its generic nature (Biggerstaff, 1998). Sources of change stem not only from new (or

variant) functional and non functional requirements but also from new version of computing environment such as tools, operating systems and networks (Jarzabek and Knauber, 1999). The changes of the component version for each combination of these variants will make the components grow in size and number. The cumulative effect of this uncontrolled growth may likely become prohibitive to reuse.

Generative reuse avoids the scaling problem by customizing the components based on the variant requirements of the product or software system. Application generators generalize and embody the commonalties and the software systems are implemented once and then reused each time a software system is built using the generator (Krueger, 1992). In another field of research initiated by Parnas in 1976, 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 (Zhang and Yang, 2003). With the use of application generator in the software product family system, it can maximize the automation of application development. Given a system specification, generators use a reusable components derived from Component Based Software Engineering (CBSE) to generate the concrete system (Czarnecki, 2000).

1.2 Background of the Problem

“Feature combinatorics problem” have the effect of scalability in component library thus affecting programmers productivity. Batory *et al.* in 1993 has studied C++ data structure libraries where features relate to data structure, memory allocation, scheme, access mode, concurrency etc. Based on the study, features may appear in classes in many different combinations. As there is need for a unique class for each legal combination of features, there is also need to develop and maintain a large number of similar classes. Form the study, (Batory *et al.*, 1993) 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 (Jarzabek, 2003). The similar problem also being researched by (Biggerstaff, 1998) where he refers to this problem as the scaling dilemma and he

has further categorized the scaling dilemma into two types of scaling, the vertical and the horizontal scaling.

Software Product Line (SPL) requires generic assets to cover all elements the product family is built from, and their corresponding composition rules. Clarifying how the various parts may be combined is a very challenging task. Generic assets in SPL in SPL differ than the asset of one system in the fact that they embrace common and variable product aspects. Generic assets can be instantiated, that is, product specific asset can be derived from them. In order for SPL to handle the variability in its product the implementation approach in SPL tends to have the same feature combinatorics and also horizontal and vertical scaling problem (Anastasopoules, 2001).

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 in order to map abstract user requirements onto appropriate configurations of components. Figure 1.1 shows three components which are essential to implement generative reuse based on (Czarnecki, 1999). They are problem space, configuration knowledge and solution space.

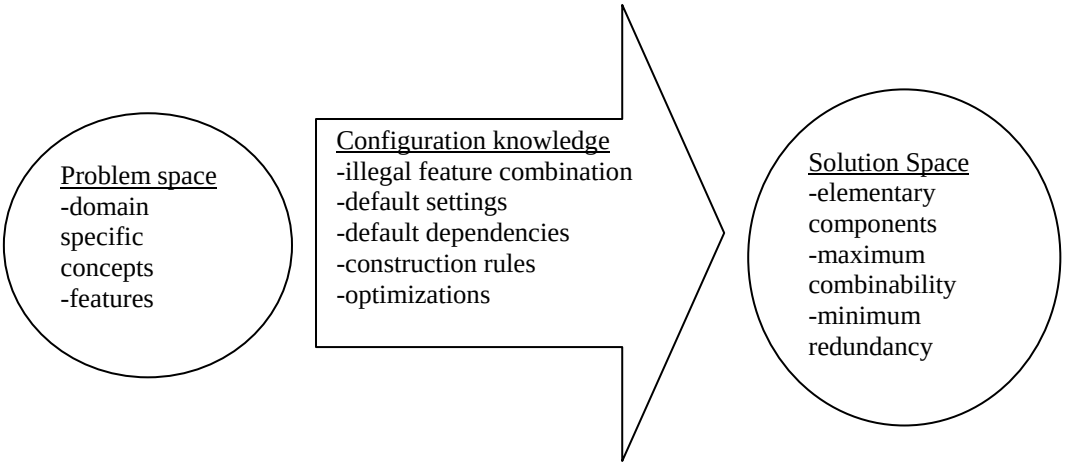

Figure 1.1: Components essential for generative reuse

The three essential components lead to three main challenges generative reuse as shown in Figure 1.2. The challenges mainly in specification and generator structure that cover all the three components in Figure 1.1. In order to propose solution for the challenges two basic questions need to be answered: the architecture of generator and the methods and mechanism of generator implementation.

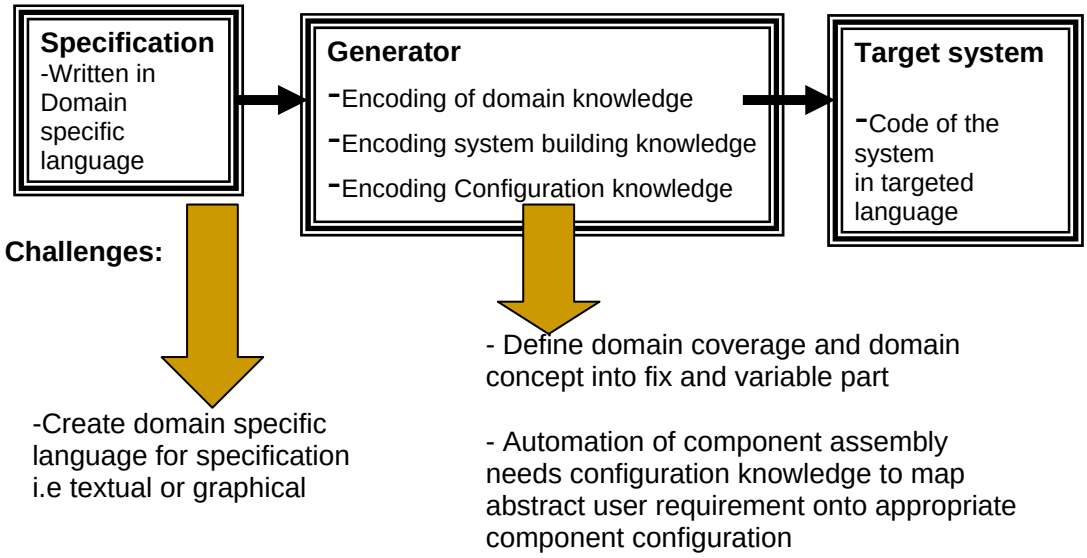

Figure 1.2: Challenges in generative reuse

1.3 Objectives

1. To study on the basic architecture of a generator.
2. To study the mechanism of generators implementation.

1.4 Scope of the Study

The scope of this research was limited to the following;

1. This study concentrates on code generation only and human intervention is still needed in order to execute the code because there will be no compiler or parser built for the purpose of code execution.
2. This study only considers the use of application generators in Software Product Line only.

1.5 Report Outline

Chapter II discusses the literature review related to this study. In this chapter, the core elements of this research: generative reuse, CBSE, application generator, domain engineering and SPL will be reviewed in detail.

Chapter III describes the application generator and the basic architecture of a generator. The chapter also discusses state of the art for SPL variability implementation methods.

In Chapter IV, the review results on generator technologies using two fundamental methods for handling variability in SPL: stepwise refinements and separation of concerns will be concluded.

CHAPTER II

LITERATURE REVIEW

2.1 Software Reuse

Traditionally, software development addressed challenges of increasing complexity and dependence on external software by focusing on one system at a time and on delivery deadlines and budgets, while ignoring the evolutionary needs of the system. This has led to a number of problems such as failure of the majority of projects to meet their deadline, budget, and quality requirements and the continued increase in the costs associated with software maintenance. To meet these challenges, software development must be able to cope with complexity and to adapt quickly to changes (Crnkovic, 2003). Software reuse is one field in software engineering focusing on the researches solely for the intention to avoid developing software from scratch thus helping developers to face with the inherent difficulties in software development. From this prespective Component Based Development (CBD) appears to be the right approach. In CBD, software systems are built from assembling components already developed and prepared for integration (Crnkovic, 2003).

The basic concepts of systematic reuse are simply by developing systems of components of a reasonable size and reuse them (Jacobson *et al.,* 1997). This idea has been extended beyond the reuse of codes to the reuse of requirements, analysis models, design and test. Ted in (Biggerstaff and Perlis, 1989), has given the framework of theory and application in software reuse. 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 as composition based technologies. Instead of in composition reuse, the building block of reusable assets can be pointed before and after its use, in generative reuse the components being reused are often pattern woven into the fabric of generator program (Biggerstaff and Perlis, 1989). 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 (Jarzabek, 1995). Application generators the second group of generation technologies, translate specifications into application programs. Lastly the transformation systems 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 (Baxter, 2002). The framework for reusability technology is as shown in Table 2.1.

Table 2.1: Framework for reusability technology

Features	**Approaches to Reusability**				
Components Reused	Building Blocks		Patterns		
Nature of Component	Atomic and Immutable Passive		Diffuse and Malleable Active		
Principle of Reuse	Composition		Generation		
Emphasis	Application Component Library	Organization and Composition Principle	Language Based Generators	Application Generators	Transformation Systems
Typical System	*Libraries of Subroutines	*Object Oriented *Pipe Architecture	*VHLLs * POLs	*File Mngmt	*Language Transformation

2.2 Component-Based Software Engineering

CBSE shift the emphasis from programming software to composing software systems. Foundation of CBSE is the foundation that there is sufficient commonalty in

many large software systems to justify developing reusable components to exploit and satisfy that commonality.

An approach to CBSE is component composition. Under component composition, there are two important issues:

a. given a set of components, and a schema for composing them, check that the proposed composition is feasible ***(verification)*** and satisfies a given set of requirements ***(validation);*** this is referred **as** the composition verification and validation problem and
b. given a set of requirements, find a set of components within a component library whose combined behavior satisfies the requirements; we refer to this **as** the bottom up design problem.

Our previous works on software reuse at analysis and design level using software analysis pattern and component-based software engineering (CBSE) (Jawawi et. al, 2005) have found that in order for the end-users or software developers to fully benefited the reuse from the CBSE, an environment which is tightly coupled to the domain engineering process must be provided. The generative reuse approach (Frakes and Kang, 2005), has been identified as an approach to provide this environment since, it can give higher system reliability by replacing error prone human processes in software development by automation that can produce a more reliable system.

2.3 Generative Reuse

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 (Frakes and Kang, 2005).

As mentioned in the previous section, there are three generation technologies but in this study focuses only on the application generator concept. Constructing an application generator is appropriate (Krueger, 1992):

1. when many similar software systems are written,
2. when one software system is modified or rewritten many times during its lifetime, or
3. when many prototypes of a system are necessary to converge on a usable product

The first point is the applicable reason why application generator is chosen and the reusable assets are extracted from domain analysis of software product line.

2.3.1 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 (Thibault and Charles, 1997).

In (Goebl, 2000), application generators is viewed as soft automatic programming systems where all soft automatic programming approaches can be seen as software reuse approaches. All of these soft automatic programming systems 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 how to generate a compiler from the input grammar and generative CASE tools build class-templates from Business Object model to relational database structures.

Cleaveland (1998) has suggested a few steps in building an application generator which are described in the following sections:

2.3.1.1 Recognizing domains

Domain that is amenable to generator technology will have implementations with recognizable patterns at the source code level, or at higher levels in the form of similar programs, designs and architectures. The pattern recognition can be done by identifying similarity patterns or cloning in application systems (Jun and Stan, 2005), (Basit *et al.*, 2005), (Rajapakse and Jazarbek, 2005).

2.3.1.2 Defining Domain Boundaries

Domain boundaries determine the range of the generator in terms of what features should be included or excluded (i.e., setting the domain coverage). Increasing the domain coverage allows the generator to handle more problems but typically makes the generator less efficient and harder to use. Narrowing the domain width increases the domain leverage, so that the generator can do more work but for a limited range of problems.

2.3.1.3 Defining an Underlying Model

Identifying the abstraction presented to the user of the application generator. Common abstractions include sets, directed graphs, trees, formal logic systems, and computational models such as finite-state machines and spreadsheets.

2.3.1.4 Defining Variant and Invariant Parts

A generators variant part usually corresponds to system specification. Invariant parts (the how) are usually fixed assumptions about the domain or implementation. They are design details that the user prefers not to worry about. In order to build application generator, understanding of the domain common and variant structures must be done earlier. This process is basically done in domain analysis. The modeling of variations and also the commonalties in the application domain is crucial because the variations contribute to the independent changes to the components and the commonalties of the components contribute to the generic architecture of the existing components.

2.3.1.5 Defining the Specification Input

Defining the way in which the user specifies each instance of a generated program. As mentioned earlier, options include textual specification languages (application-oriented languages, fourth generation languages, etc.), templates, graphical diagrams, interactive menu-driven dialog, and structure oriented editing.

2.3.1.6 Defining and Implementing Products

Defining product of application generator is to determine what output the generator will produce. Typically a generic software design must be developed that will meet the needs of all applications in the domain. The generator must tailor the generic design for each application.

2.4 Domain Engineering

To enable product-line engineering, a well-accepted convention is to divide the engineering process into two different processes: domain engineering and application engineering as stated in (Macala *et al.*, 1996), (Harsu, 2002). Domain engineering and application engineering can be called *engineering-for reuse* and *engineering-with-reuse*, respectively. The purpose of domain engineering is to provide the reusable core assets that are exploited during application engineering when assembling or customizing individual applications.

Domain engineering is most often divided into three phases: domain analysis, domain design, and domain implementation. In the following sections, the phases in domain engineering will be elaborated further while application engineering due to its varieties of implementation in application generator will be presented in subsection on related work.

2.4.1 Domain Analysis

Domain analysis is first introduced by Neighbors to denote studying the problem domain of a family of applications (Neighbors, 1980). The output of domain analysis

is domain model. (Harsu, 2002) has generalized the common artifacts or processes belonging to domain model as follows:

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

Generative reuse is closely related with the domain analysis where the knowledge of the domain will be kept in the application generator in order for it to customize affected components based in the configuration knowledge. Domain is defined as a family or set of systems including common functionality in a specified area (Hwang, 2006). Domain analysis for reusability is the process of analyzing an application domain in order to built reusable design. Domain analysis for reusability is 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 (Lubars, 1991). However software has many variables, which differ from the reusability of hardware, and software variations are much more difficult to standardized, identify and control (Hwang 2006). A suitable domain analysis method is crucial in order for a systematic analysis to capture the domain applications commonalties and also variants can be achieved.

2.4.2 Domain Design

Domain design means designing the core architecture for a family of applications.
It comprises the selection of the architectural style (Czarnecki, 1999), (Harsu, 2002). In addition, the common architecture under design should be represented using different views. The core architecture should also provide variability between applications. In this phase, it is decided 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 (Harsu, 2002).

2.4.3 Domain Implementation

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 (Harsu, 2002).

2.5 Software Product Family (SPL)

The goal of the software product family 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 (Jaring, 2004). In software product line, among the main issues that have to be catered are how to capture the features of SPL and how to map the features to develop suitable architectures for SPL. The subsequent sections describe these two main issues.

2.5.1 SPL Models

The terms ”domain” and ”product line” are very close to each other. However, the difference is that a domain consists of conceptual items, while a product line

comprises concrete products or applications to be developed (Harsu, 2002). SPL engineering involves the analysis, design and implementation of a product line that satisfies the requirements of all target applications (Aquil Saleh, 2005). The main focus of SPL is to model the commonalties and also the variabilities of SPL. Backer defines variability as 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.

Feature diagrams are often used to model common and variant product line requirements. Feature diagrams provide a graphical tree-like notation that shows the hierarchical organization of the features. By traversing the feature trees, we can find out which variants have been anticipated during domain analysis.

Features are classified as mandatory, optional and alternative and or-features (Zhang and Yang, 2003), (Czarnecki, 2000). Common requirements can be modeled as mandatory features whose ancestors are also mandatory. Variant requirements can be modeled as optional, alternative, or or-features (Zhang and Yang, 2003). An example of feature diagram for a car is as shown in Figure 2.1 taken from (Czarnecki, 1999).

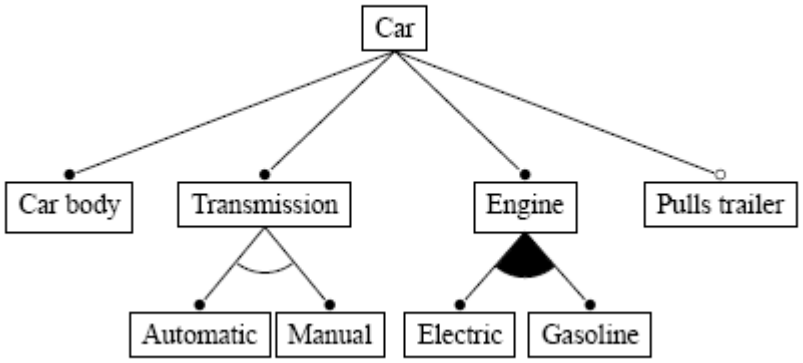

Figure 2.1: Feature diagram of a simple car

Based on the diagram mandatory features are marked with filled circle in the head of the line. Optional features have an empty circle in the head of the line. For example, a car can either pull a trailer or cannot. Alternative features are connected with an empty arc and filled arc connecting features denotes or-features. From the diagram, it

shows twelve different car variants where there are two different transmissions, three kinds of engine and an optional trailer coupling. Constraints that cannot be expressed in a feature diagram have to be recorded separately (Czarnecki, 1999).

A multiple-view model for a software product line defines the different characteristics of a software family, including the commonality and variability among the members of the family. A multiple-view model is represented using the UML notation and considers the product line from different perspective. The functional requirements of a system are defined in terms of use cases and actors. When modeling a SPL, kernel use cases are those use case required by all the members of the family. Optional use cases are those use cases required by some but not all the members of the family. Some use case may be alternative, that is different versions of the use case are required by different members of the family (Aquil Saleh, 2005).

2.5.2 SPL Architecture

Software architecture is defined as a set of components that interact with each other through well defined interfaces (connector) to deliver the required system behavior. A Product-Line Architecture is a design for families of related applications; application construction also called product instantiation is accomplished by composing reusable components. SPL architecture or generic software architecture is different from software architecture for a specific system that it forms a blueprint for a family of systems. A generic architecture is customized during program construction time to accommodate variant requirements into a target system. An instance of a generic architecture that results from customization forms a runtime architecture for a specific system (Cheong, 1999).

2.6 Related Works

Application generators are practical and attractive when high-level abstractions from an application domain can be automatically mapped into executable software system (Krueger, 1992). The main problem in product line development is the design of common set of assets (Nitto, 1997) and also handling the variabilities for the customization of the product line. The related work is focused on how other researcher models the commonalities and variabilities, for which common sets of assets (generic assets) do they focused on and what are the implementation approach to variability (Anastasopoules, 2001) they use to implement the configuration knowledge in the generator.

(Cheong, 1999) has used frames as an implementation approach for customizing generic architecture of Facility Resource System SPL. The variant requirement is mapped into sequence of activities. The customization method has reported as having the advantages of domain independence, have partially transformed the requirements to implementation and language independent. The implementation has a few weaknesses where there is lack of support in requirement dependency and also if there are changes to the SPL, an in-depth knowledge of the frame assembly is needed in order to implement new requirement.

Another implementation approach for research in application generator in SPL is by (Czarnecki, 1999). Gen Voca architecture is used for domain grammar and reflection approach in the form of template metaprogramming. Reflection approach relates strongly to metaprogramming where objects in higher levels of abstraction (metalevels) are established to represent entities like operating systems, programming languages, processors, object models, etc. Reflection enables access to such metaobjects and therefore allows architecting flexible systems. Reflection can be combined with dynamic class loading in order to load modules unknown until runtime, depending on the deployment context and invoke operations on these modules. Base functionality can be “reflected” and manipulated according to a configuration.

(Czarnecki, 1999) reported that the approach is hard to debug and maintain and suggest the use of active libraries instead. (Anastasopoules, 2001) also reflected the same problem with reflection and added the difficulty in understanding the nature of reflection and strongly recommended for its use in special systems (e.g. object inspectors) and its usage in other systems should be handled with care.

(Jarzabek, 2003) and (Jun, 2005) has also implemented reflection approach for unifying similarity pattern to open source JDK 1.1 Buffer library and also to a web portal SPL. The unifying pattern refers to the similar program structure of any kind and granularity repeated many times within a program or across programs. With the use of reflection, the XVCL metaprogramming, the size of the original web portals have been reduced by 61% (Jun 2005) and with the JDK library by 68% (Jarzabek, 2003).

The use of Aspect oriented programming (AOP) has been reported in (Saleh, 2005). Aspect Oriented Programming (AOP) is a technique developed at Xerox PARC which enables the modularization of crosscutting concerns, namely aspects, as well as the integration of join points. Join points are the locations in systems that are affected by one or more cross-cutting concerns. The process of integrating join points involves describing how a cross-cutting concern affects code at one or more join points. The integration process is referred to as composition or weaving (Anastasopoules, 2001).

While it is reported that with the use of AOP, a better understanding of the product line can be obtained by separating the common source code from the variable source code, the optimization of code when using AOP have not been reported.

The basic concepts underlying the generation based reuse and also SPL has been identified. From the literature, all the implementation approaches in SPL such as inheritance, parameterizations, dynamic class loading, frames, reflection, AOP and design pattern has the potential in handling the product line variability at code level. Further review is required to identify the methods and mechanisms used in each approach.

Software Product Line Variability Implementation Methods and its Application in Generator Implementation

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@utm.my, dayang@ utm.my, safaai@ utm.my

Abstract

Software Product Line (SPL) is a type of reuse where common artifacts can be shared by similar softwares or members in the product line. Besides sharing common features, each member in the product line has significant variations referred as variability. Variability implementation requires focus on two important issue, delaying design decision and also ease of changes in software. This study is based on the initial proposal of two methods, stepwise refinements and separation of concerns for the use in generator implementation. Generators with the implementation of these methods have been reportedly used in various SPL implementations. Based on this motivation we study the underlying concepts of these methods and the origin of its idea. We also study the issues and its current implementation in generator. The result of this paper can help designer and researcher who are interested in the development of application generator in SPL to comprehend the underlying methods in SPL implementation and also its usage in generator.

1. Introduction

Software Product Line (SPL) is software reuse which concentrates on large grain reuse of software asset. Granularity of reusable software asset as described by Reusable Asset Specification (RAS) by Object Management Group (OMG)[1] increases when it addresses multiple problems, and/or may offer alternative solutions to the problems. In SPL, common artifacts and software assets can be shared by multiple similar softwares or members in the product line. Besides sharing common features, each member in the product line has significant variations referred as variability. Variability has been seen as the core issue in SPL and the management of variability can be seen as the essence in SPL practice and various variability mechanisms have been published for the purpose of managing variability in different artifact level [2]. In this paper we concentrate on implementation-level and application-level artifact of variability mechanism.

In SPL, variability can be achieved by delaying design decision to a later phase in the lifecycle of the software system. It is important also for SPL implementation to anticipate changes and thus the programs should facilitate design which is easy to change. In order to achieve the delayed design decision and alleviate changes in software, two classic methods in software design i.e. stepwise refinement and module specification (later on known as separation of concern by [3]) has been proposed as suitable for SPL implementation more than three decades ago [4-7]. Variations of these methods were still implemented by various researchers since then in different kinds of SPL implementation [8-18]. Based on the importance of these two methods in SPL implementation, we will study the underlying concepts of these methods and the origin of its idea. Another focus of this paper is to study the generative reuse with the use of generator where it has also been reportedly used in SPL implementation [8-10, 15-19].

Generative reuse is reuse at the specification level with application generators or generators [20]. Generative reuse is done by encoding domain knowledge and relevant system building knowledge into a domain specific application generator [21].

The main contribution of this paper is the study of the current variability implementation method and its mechanism: separation of concern and stepwise refinement in generator implementation. The basic concepts of these two methods will also be discussed. In this paper we will discuss in detail the basic architecture of a generator.

The remainder of this paper is organized as follows: In section 2 we discuss the research framework of this paper. Section 3 discusses the application of generator and the basic architecture of a generator. In Section 4 these two methods will be discussed in detail. The following section, section 5 discusses the current implementation of these methods in generator implementation. Lastly in section 6, the conclusion and also the future work of this research will be discussed.

2. Research Framework

The framework of our study comprises of literature study, initial findings and state of art analysis of variability implementation method as shown in Figure 1. Firstly we reviewed literatures on generator architecture for identifying the basic structure of generators. Secondly we reviewed related literatures on basic methods proposed for SPL implementation in order to identify the most common method used in various SPL implementations.

Our initial results were the basic architecture of generator in general and also the study of the two basic methods for SPL implementation separation of concerns and stepwise refinements. Finally we did an analysis on the state of art in mechanism used in generator implementation which specifically uses the variability implementation methods and produce a table encompassing the analysis results. The results can be used as a reference for the various mechanisms for SPL implementation.

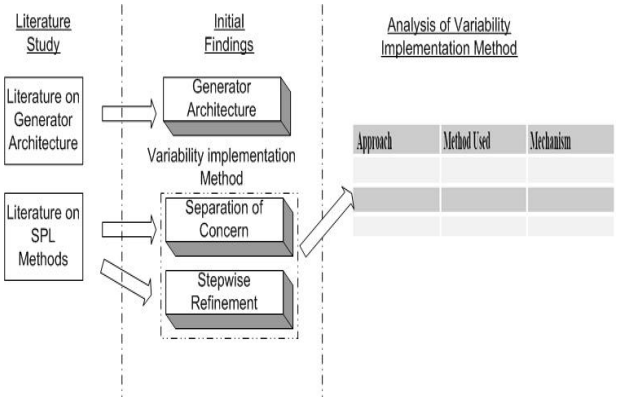

Figure 1 Research Framework

3. Generator Application and its Architecture

Application generator or simply generator has been applied successfully in the area of programming language systems such as compilers, language based editing systems and static program analyzers [9] and also database generators [22]. It's usage has been extended in [23] to handle scalability problem in traditional library of software components and its effectiveness has been reported in [23, 24] and experimented in [11]. Previously reported generator success focused on generating domain specific application. As stated in [25] application generator implementation is appropriate when many similar software systems are written, or 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. Generator has then been used for generating multiple similar application in SPL where it was used as a mechanism to handle variability in the SPL implementation [10, 11, 26-29] .

There are several papers [30-33] describing the basic architecture in a generator. Figure 2 show the basic architecture of a generator which consists of the front-end of generator and also its back-end adapted from [30, 33]. Basically generator accepts input of abstract specifications to describe the software either in a text or graphical form. Textual input specification requires the front-end of the generator to have a lexer in order to break the specifications into tokens. Parser then parse the token into parse tree and semantic analyzer checks for semantic error [31, 33].

The intermediate representation for the generator or also known as an underlying model based on [31] can be in the form of sets, directed graphs, formal logic systems and computational model like finite-state machine. Other specification formats (i.e., graphical representation) may map straightforwardly to the intermediate representation.

Generator also has been reported to encode domain knowledge and design knowledge and use repository of components for customization in order to produce code for new system in the domain [33]. Domain knowledge consist of domain specific scheme for encapsulation of system knowledge [32]. For an ideal design knowledge, [34] has referred it as design history where it records the original specification of the problem, series of transformation applied to the specification and the justification for these transformation. In the repository, component interfaces are specified formally and also with adapters for composition purposes [32]. This repository can be based on any component model available.

In [16, 30] the use of transformation engine is highlighted in order to implement transformation to the intermediate representation. With transformation, a concrete executable program which is still in intermediate representation form (represented as a flow graph or an abstract syntax tree) will be generated. The final process is where the program generator transforms the intermediate representation into textual representation which is usually in the form of high-level programming language.

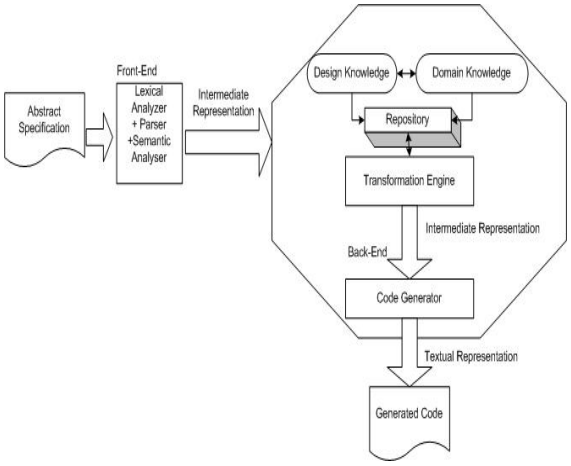

Figure 2 Basic Architecture of a Generator

4. SPL Implementation Method to Handle Variability

Variability is the main issue in SPL implementation. In delaying design decisions, a family member will share the same software assets until it come to the point where it differs from other member of the family and this point is referred as variation point. Variability also requires the programs to anticipate the changes that might occur hence accommodate the ease of change facility.

We concentrate on two methods for development of program families or SPL i.e. programming by stepwise refinement and module specification based on the first proposal by Parnas in [6]. These methods are introduced as it is suitable for postponing and therefore giving the programmer the ease to change decisions about program implementation.

4.1 Separation of Concern

Most software engineering activities involve certain type of concerns. For example, among the concerns in software design are on the features, non functional requirements and many other concerns. In [18], concerns is identified as any area of interest in program solution either in functional features, quality requirements, software architecture, detail design or implementation. Dijkstra was the first person to introduced the concept of separation of concern [18]. He refers to this concept as an effective ordering of ones thought where one particular aspect is considered in isolation from other aspects. Other aspects however are not being ignored but merely irrelevant at that time [3].

With concerns basically separated among each other, the complexity of dealing with all concerns at the same time can be lowered and developers can deal with the concern individually. Module specification introduced by Parnas in [35] can be associated with separation of concern where the former method is the separation of concern implementation as it separates the concern of what services a module offers and how the services are implemented within the module with the help of API [18].

There are few stated problems when concerns are implemented conceptually in conventional programming language. Among the problems are inadequate abstraction of concern at the implementation level results in intertwined code [36] making the code hard to understand and modify. Concerns which are scattered in different levels of software development known as crosscutting concerns are hard to be localized in single modular decomposition and requires unconventional solutions such as generative techniques to separate concerns at a meta-level extra plane [18].

4.2 Stepwise Refinement

Stepwise refinement on the other hand concentrates on the refinements of an abstract concept to a more concrete and concise implementations. This method starts with describing functionality at a very high level specification, then successively decomposing design decisions into more detailed levels one level at a time until the detail forms into a target code [37, 38].

The notion of stepwise refinement was first introduced by Dijkstra in [7]. His idea revolves around a concept which views the top most part of a program as an abstract form, the lower part as the refinement of the abstraction and the bottom part as standard interface. He also envisioned an incomplete program where the top half of it can be regarded as a complete program to be executed and the bottom half gives a feasible implementation.

In [4] the detail implementation of Dijkstra's idea was shown by refining specification until the specification is precise and its near to programming implementation. The incomplete programs envisioned by Dijsktra is viewed by Parnas as an intermediate stages represented by programs which are complete except for the implementation of certain operators and operands type [6]. This ability can be used for delaying design decision.

The initial intention of stepwise refinement was as a method for program correctness [4, 6, 7]. Since refinement has its intermediate stages, it allows programmers to codify design decision without committing to an implementation technology up front where refinements can be encode as COM objects, Java objects or as meta-programs [11].

With this ability stepwise refinement not also can be used in delaying design decision but it can also support separation of concern where concern in implementation technology is separated from application design [11]. In [11] also, the author has acknowledge the problem of the sizing the scale in refinement.

5. State of Art for the SPL Variability Implementation Method

Related work focuses on the researchers that have been done which reportedly incorporate either one or both of these two methods in the implementation of their generator for generating SPL application.

Czarnecki separates problem domain concerns which become intimately and unavoidably interwoven with problem solution concerns in program components [16, 18]. With the use of configuration knowledge, abstract requirements in the problem domain is mapped onto appropriate configurations of components in the problem solution [16].

Separation of concern implementation also has been reportedly used for the generic architecture of SPL [9, 10, 14, 39]. Domain models and generic software architectures facilitate the reuse of code[9]. In [10], separation of concern revolves around separating specification of how variant affect a generic component separately from the component itself. The variant specification and also generic architecture guides generator in customizing components.

The implementation of separation of concern also governs the aspect oriented based generator as reported in [17]. In [17] concerns are separated by the separation of common source code from the variable source code. Code weaving is used to customized target applications hence an executable program will be generated.

Generators concentrates fundamentally on refinements of specification where it takes the abstract specifications and progressively making the specifications into a concrete program [30]. GenVoca use stepwise refinement to refine layers of components

Based on the problem of scale in stepwise refinement implementation, Batory in [15] has implement algebraic specification using AHEAD as composer to transform equation in order to refine code and noncode artifacts to form a synthesized system. Algebraic specification enables a consistent refinement of artifacts hence lowering the scalability problem. Another work in [19] has reportedly combined the use of aspect oriented programming in the refinement of mixin layers in AHEAD.

Table 1 Approach for SPL Methods in Generator Implementation

Approach	Method Used	Mechanism
Configuration knowledge	Separation of concern	Template metaprogramming
XVCL	Separation of concern	Tree Structure of XVCL command
Aspect Oriented	Separation of concern	Modularized unit of concern
GenVoca	Stepwise refinement	Refinements of Mixin Layers
AHEAD	stepwise refinement	Scaling refinements of Mixin Layers
AHEAD	stepwise refinement and separation of concern	Aspectual refinements of Mixin Layers

The state of art for generator implementations are not an exhaustive review as there are numerous other implementations of these methods implicitly reported in the literature. We merely chose literatures either by prominent researchers in this field or literature which explicitly proclaim the use of this technique in their research. Table 1 shows the summary of these works where it shows majority of the work done are based on either stepwise refinement or separation of concern but it can be seen that there is one research which combine both methods in order to separate concerns and refine the program in generator.

6. Conclusion and Future Work

It can be said generally that with the implementation of stepwise refinement concept it helps in postponing the design decision and with the use of separation of concern ease of changing design decisions can be achieved. In stepwise refinement, this method has more formal approach with the use of algebraic specification compared to

separation of concern which is more abstract in its approach. Although majority of the work focuses in using either methods, but it can be seen that the most current work has focused on using both methods in the generator.

For future work, we will do a comparison of generator technology. In order to achieve this we have to find features that suitably reflect the generator implementation. Two of the features have already been identified in this paper, refinement and also separation of concern.

6. References

[1] *OMG Object Management Group. Reusable Asset Specification: Version 2.2. Needham: Object Management Group*. 2003.

[2] Clements, P.C. *Managing Variability for Software Product Lines: Working with Variability Mechanisms*. in *10th International Software Product Line Conference (SPLC '06)*. 2006. Baltimore, Maryland , USA: IEEE.

[3] Dijkstra, E.W., *On the Role of Scientific Thought.* Springer-Verlag, New York, 1982(Selected Writings on Computing: A Personal Perspective): p. 165-182.

[4] Wirth, N., *Program Development by Stepwise Refinement.* Communications of the ACM, 1971. **14**(4): p. 221-227.

[5] Parnas, D.L., *On the Criteria to be Used in Decomposing Systems into Modules.* Communications of the ACM, 1972. **15**(12): p. 1053-1058.

[6] Parnas, D.L., *On the Design and Development of Program Families.* IEEE Transactions on Software Engineering, 1976. **SE-2**(1).

[7] Dijkstra, E.W., *Structured Programming*, in *Software Engineering Techniques*, B.J. N and R. B, Editors. 1970. p. 84-87.

[8] Don Batory, V.S., Jeff Thomas, Sankar Dasari, Bart Geraci, Martin Sirkin, *The Gen Voca Model of Software-System Generators.* IEEE Software, 1994.

[9] Jarzabek, S., *From reuse library experiences to application generation architectures.* ACM, 1995 p. 114-122.

[10] Jarzabek, S., Knauber, P., *Synergy between Component-based and Generative Approaches.* Lecture Notes in Computer Science, Springer Verlag, 1999: p. 429-445.

[11] Smaragdakis, Y. and D. Batory, *Application Generators*. Software Engineering volume of the Encyclopedia of Electrical and Electronics Engineering. 2000: J. Webster, John Wiley and Sons.

[12] Hong Yu Zhang, S.J., Soe Myat, Swe, *XVCL Approaches to Separating Concerns in Product Family Assets.* Lecture Notes in Computer Science. Proceedings of the Third International Conference on Generative and Component-Based Software Engineering, 2001. **2186**: p. 36-47.

[13] Krueger, C.W. *Using Separation of Concerns to Simplify Software Product Family Engineering*. in *In Dagstuht Seminar No. 01161*. 2001. Dagstuhl Castle, Wadern, Germany, April, 2001.

[14] Zyl, J.v. *Product Line Architecture and the Separation of Concern*. in *Software Product Line Conference*. 2002: Springer-Verlag Berlin Heidelberg.

[15] Batory, D., *Scaling Step-Wise Refinement.* IEEE Transactions on Software Engineering, 2004. **30**(6): p. 1-17.

[16] Czarnecki, K. *Overview of Generative Software Development*. in *In J.-P. Banâtre et al. (Eds.): Unconventional Programming Paradigms (UPP) LNCS 3566*, . 2005. Mont Saint-Michel, France.

[17] Saleh, M. and Gomaa, H. *Separation of Concerns in Software Product Line Engineering*. in *International Conference on Software Engineering*. 2005. St. Louis, Missouri.

[18] Jarzabek, S., *Chapter from book "Software Maintenance and Evolution: Reused based Approach"*. 2007: AUERBACH Publications.

[19] Don Batory, V.S., Jeff Thomas, Sankar Dasari, Bart Geraci, Martin Sirkin, *The Gen Voca Model of Software-System Generators.* IEEE Software, 1994.

[20] Apel, S., T. Leich, and G. Saake. *Aspect Refinement in Software Product Lines*. in *In Aspects and Software Product Lines (ASPL'05): An Early Aspects Workshop at SPLC-Europe'05*. 2005. Rennes, France.

[21] Prieto-Diaz, R., *Status report: software reusability.* IEEE Software, 1993. **Volume 10**(Issue 3): p. 61 - 66.

[22] Frakes, W.B., Kang, K., *Software Reuse Status and Future.* IEEE Trans. On Software Engineering, 2005. **31. No 7**: p. 529-536.

[23] Horrowitz, E., A. Kemper, and B. Narasimhan, *A Survey of Application Generators.* IEEE Software 1995. **12**: p. 40-54.

[24] Biggerstaff, T.J., *A Perspective of Generative Reuse.* . Annals of Software Engineering, 1998. **5**: p. 169-226.

[25] Biggerstaff, T.J. *The Library Scaling Problem and the Limits of Concrete Component Reuse*. in *3rd International Conference on Software Reusability*. 1994. Rio de Jeneiro, Brazil: IEEE Press.

[26] Batory, D. *Refinements and Separation of Concerns*. in *Second Workshop on Multi-Dimensional Separation of Concerns, International Conference on Software Engineering*. 2000. Limerick, Ireland.

[27] Krueger, C., W., *Software Reuse.* ACM Computing Surveys, 1992. **24**(2): p. 132-183.

[28] Hongyu Zhang, Jarzabek, S., *XVCL: A Mechanism for Handling Variants in Software Product Lines.* Science of Computer Programming Elsevier Science, 2004. **53**.

[29] Batory D., C.J., Bob MacDonald and Dale von Heeder. *Achieving Extensibility through Product Lines and Domain-Specific Languages: A Case Study*. in *International Conference on Software Reuse*. 2000. Vienna Austria.

[30] Cheong, Y.C.a.J., S., *Modeling Variant User Requirements in Domain Engineering for Reuse.* Information Modeling and Knowledge Bases: p. 220-234.

[31] Stan Jarzabek, W.C.O.a.H.Z., *Handling Variant Requirements in Domain Modeling.* The Journal of Systems and Software, 2003. **68**: p. 171-182.

[32] Batory, Y.S.a.D., *Application Generators*. Software Engineering volume of the Encyclopedia of Electrical and Electronics Engineering. 2000: J. Webster, John Wiley and Sons.

[33] Cleaveland, C., J., *Building Aplication Generators.* IEEE Software, 1988: p. 25–33.

[34] Marcelo Sant'Anna, J.C.S.a.A.F. *A Generative Approach to Componentware*. in *Workshop on Component-Based Software Engineering, International Conference on Software Engineering (ICSE'98)*. 1998. Kyoto, Japan.

[35] Frakes, B., et al., *Panel: Linking Domain Analysis and Domain Implementation.* IEEE Software, 1998.

[36] Batory, D. *Program Comprehension in Generative Programming: A History of Grand Challenges*. in *12th International Workshop on Program Comprehension (IWCP '04)*. 2004. Bari, Italy.

[37] Parnas, D.L., *A Technique for Software Module Specification with Examples.* Communications of the ACM, 1972. **15**(5).

[38] Hursch, W.L. and C.V. Lopes. *Separation of Concerns*. in *Technical Report NUCCS-95-03*. 1995. College of Computer Science, Northeastern University.

[39] Robert G. Reynolds, J.I.M., Stephen E. Porvin, *Stepwise Refinement and Problem Solving.* IEEE Software, 1992. **9**(5): p. 79-88.

[40] Howe, D. (2007) *FOLDOC - The Free Online Dictionary of Computing*.

[41] Jaring, G. and Bosch, J., *On the Notion of Variability in Software Product Lines.* IEEE, 2001.

Generator Technologies Using Fundamental Methods in 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@utm.my, dayang@ utm.my, safaai@ utm.my

Abstract

Software generator when applied to Software Product Line (SPL) can reap the benefit of automation in software development and also systematic reuse. Software Product Line (SPL) is a type of reuse where common artifacts can be shared by similar members in the product line. In addition each member in the product line also has significant variations referred as variability. There are two fundamental methods for handling variability in SPL: stepwise refinements and separation of concerns. Generators with the implementation of these methods have been reportedly used in various SPL implementations. Our review approach is to study the mechanism of generators implementation based on the review framework that we have presented in the paper. Although there are reviews being done on generator technologies, but to our best knowledge there are no explicit review based on these two methods.

Keywords: Software Product Line, Generator, variability, scalability, stepwise refinement, separation of concern.

1. Introduction

Generator is a program that takes a higher-level specification of a piece of software and produces its implementation. The piece of software could be a large software system, a component, a class, a procedure and so on [1]. Using generator as a means for reuse is referred by reuse community as generative reuse. In [2], generative reuse is defined as reuse at the specification level with application generators or generators. Generative reuse is done by encoding domain knowledge and relevant system building knowledge into a domain specific application generator [3].

Generative reuse via 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[4, 5]. We are interested in the first application of generator where we analyze generator implementation in Software Product Line (SPL) domain. Software Product Line (SPL) is a type of reuse where common artifacts can be shared by similar members in the product line. In addition each member in the product line also has significant variations referred as variability.

In SPL, variability can be achieved by delaying design decisions to a later phase in the lifecycle of the software system. It is important also for SPL implementation to anticipate changes and thus the programs should facilitate design which is easy to change. In order to achieve the delayed design decision and alleviate changes in software, two classic or fundamental methods in software design i.e. stepwise refinement and module specification (also known as separation of concern by [6]) has been proposed in [7]. The motivation in writing this paper is based on the different approaches of these methods were still implemented by various researchers since then in different kinds of SPL implementation [8-13]. Although there are reviews being done on generator technologies, but to our best knowledge there are no explicit review based on these two fundamental methods. i.e. separation of concern and also stepwise refinements.

The remainder of this paper is organized as follows: Section 2 outlines the fundamental methods in SPL. In Section 3 our review framework together with the details of the review are highlighted. Section 4 discusses the findings of the review. The last section presents the conclusion of this paper.

2. Fundamental Methods in SPL

We concentrate on two methods for development of program families or SPL i.e. programming by stepwise refinement and separation of concern based on the first proposal by Parnas in [7] for variability. implementation method.

2.1 Separation of Concern

In [14], concerns are identified as any area of interest in program solution either in functional features, quality requirements, software architecture, detail design or implementation. With concerns basically separated among each other, the complexity of dealing with all concerns at the same time can be lowered and developers can deal with the concern individually.

There are few stated problems when concerns are implemented conceptually in conventional programming language. Among the problems are inadequate abstraction of concern at the implementation level resulting in intertwined code [15] making the code hard to understand and modify. Another problem is when the concerns are scattered in different module known as cross cutting concern where the concern is hard to be localized in single modular decomposition. Separation of concerns contribute to variability implementation in SPL where this method can help in accommodating difference in design decision when instantiating members of the product line

2.2 Stepwise Refinement

Stepwise refinement on the other hand concentrates on the refinements of an abstract concept to a more concrete and concise implementations. Traditional work on step-wise refinement focused on microscopic program refinements where numerous refinements have to be applied to yield admittedly small programs. Notion of stepwise refinement was first seen as an incomplete programs envisioned an then viewed by Parnas as an intermediate stages represented by programs which are complete except for the implementation of certain operators and operands type [7].

In [16], the problem of the size of the scale in refinement has been acknowledged. Stepwise refinement facilitate in variability implementation where design decisions can be delayed until the implementation comes to the point of variation where the decision differs from other members of the product line.

3. Review Framework

This section contributes to the review of generator technology based on 3 elements its classification, the fundamental methods in SPL and also scalability of its implementation. The following is the discussion of the three elements:

i) The classification of generator technologies

Classification of generators has been done based on various focuses. In this section we try to classify the generator based on the classes or group mostly reported in the literature [1, 17-19]. We apply the classification based on technical distinction in [17] where distinct technologies for implementing generators and its components have been highlighted and among the technologies are compositional, metaprogramming and transformational. Alongside the classification we also include sample approaches for each classification as technical distinctions are not enough in discussing generators [16, 20]. However in [17] there is no mentioning of sample approach in its classification and we further refer in [18, 19] for sample approach on each classification of the generator technology.

ii) Fundamental methods in SPL

There is no explicit discussion on what fundamental methods that generators use except in [14] where a few generator technologies have been associated with implementing separation of concern.

iii) Reuse Scalability Dimension

Scalability in reusing components has two dimensions i.e. vertical and horizontal scaling as in [21]. Vertical scaling refers to how well the generator scale in terms of raw size and programming leverage. Horizontal scaling on the other hand looks at how the generator scales up in terms of feature variation (also being associated with technological aspect) [19, 21].
In the following subsections we discuss the classification of generator together with its sample applications..

3.1 Transformation

Transformation systems basically applies transformational programming where program is constructed by using successive application of transformation rules starting from specification and ends with executable program [22].

Draco, Anticipatory Optimization Generator (AOG) and Aspect Oriented Programming (AOP) are examples of transformation generator technology. Stepwise refinement methods can be observe in Draco mechanism where phased refinements were implemented to map an abstract domain language into one or more mini domain languages until the whole program has been translated into targeted conventional programming language [18]. Draco has been reportedly experienced search space problem in its transition and AOG [23], has been designed to overcome the problem. In AOG a tag-driven transformation control which allows cross-component and cross domain optimizations in the programming language domain thus it has lower search space compared Draco. Both generators implemented stepwise refinement concept in their generators implementation.

OAG and AOP were reportedly to have separation of concern in its implementation [19]. In terms of scaling Draco implementation scaled horizontally where it support scaling in feature variation whereas AOG and AOP having both horizontal and vertical scaling in their implementation as reported in [18].

3.2 Composition

Composition based generators are called forward-refinements transformation in [1]. Forward refinements are transformation of higher level representation into a lower level without redefining the modular structure of the higher level representation. Generators with composition technology are Gen Voca, and Algebraic Hierarchical Equations for Application Design (AHEAD).

GenVoca extends stepwise refinement by scaling refinements to a component or layer (i.e., multi-class-modularization), so that each refinement adds a feature to a program, and composing a few refinements yields an entire application [24] there is no separation of concern reportedly implemented in Gen Voca.

AHEAD elevate separation of concerns to user level requirements [14]. For stepwise refinement implementation, instead of using conventional programming language specification as in Gen Voca, algebraic specification is used in AHEAD and composer is used to transform equation in order to refine code and noncode artifacts to form a synthesized system [11].

In terms of scaling dimension, Gen Voca has both scaling dimensions [19]. As for AHEAD, its mechanism has also being reported in [11] to have a higher scalability as algebraic specification enables a consistent refinement of artifacts.

3.3 Metaprogramming

The importance of metaprogramming in generator technology has been reported in [1, 25]. XVCL and templat metaprogramming are examples of SPL generators which are based on metaprogramming. With XVCL, separation of concern is implemented by separating the specification variability from the components. In XVCL, generic architecture implements commonality in a software product line while metalanguage is used to specify variations to be implemented in the custom system [10] .On the other hand in [12], concerns on problem domain are separated from the concern in problem solution using configuration knowledge in the form of template metaprogramming in C++. Though there were no explicit statements to highlight how well the scalability in XVCL implementation, experiments have been done to show it can scale component vertically by removing redundant code in J2EE libraries [26] and also horizontally in SPL implementation [10, 27]. However, there is no reported experiments being done t
o show the vertical and the horizontal scaling of template metaprogramming.

4. Discussion

This discussion is based on Table 1 which summarizes the review framework discussed in previous section. Generators in transformation class have the advantage of optimizing each transformation to reorganize program for performance. With optimization, it promises a higher custom component fit to the target application hence horizontal scaling is achieved. Though gaining optimization in performance, building transformational generator is inherently more complicated where each generation of implementation have to be coordinated for different higher level construct [1].

Table 1. Generator Classification with Sample Approaches Corresponding Fundamental Methods and Scaling Dimension

Generator Class	Sample Approach	Methods		Scale	
		SC	SR	Vertical	Horizontal
Transformation	Draco	-	√	√	√
	AOG	-	√	√	√
	Aspect Oriented	√	-	√	√
Composition	GenVoca	-	√	√	√
	Ahead	√	√	√	√
Metaprogramming	XVCL	√	-	√	√
	Templat Metaprogramming C++	√	-	NA	NA

Note: SC – Separation of Concern SR – Stepwise Refinement NA– Not Available

For application which does not rely on performance as one of its qualities, using compositional generator is more cost effective [19]. Metaprogramming using XVCL also reportedly to have more lightweight approach where report from industrial experience shows that it can shortens time and have small effort in development [27].

Table 1 shows that for all generator classes, each sample approach has implemented either one or both fundamental methods of SPL. Only one sample approach AHEAD has use both of these methods in its implementation. This shows the possibility of hybridizing both methods in generator implementation where current research in AHEAD [28] has reportedly combined the use of aspect oriented programming in the stepwise refinement of mixin layers in AHEAD.

Based on Table 1 also, almost all of the sample approaches in generator have both scaling dimensions. However the intensity of how well it scale is different as shown in [18]. The reason is due to the fact that all the generators were domain specific thus they must have programming leverage in order to scale vertically. At the same time in order to accommodate the variability implementation in SPL these generators must exhibit feature variations criteria and as a result their implementations have to be scaled horizontally. This shows the challenge in SPL generator where it must satisfies both scaling dimensions.

5. Conclusion and Future Work

From our study of the fundamental methods implementation in generator technology, it shows these methods have undergone various improvements and also the implementations of these fundamental methods have been refined thus achieving the scalability dimensions. There are also possibilities in hybridizing both methods to reap even more benefit in generator. Our future work is to study the lightweight approach in XVCL generator technology, in handling variability in SPL implementation. This is due to the fact that XVCL is cost effective generator in terms of reducing development time and also development effort.

6. References

[1] Czarnecki, K. and Eisenecker, U., *Generative Programming- Methods , Tools and Applications*. 2000, Boston MA: Addison-Wesley.

[2] Prieto-Diaz, R., *Status report: software reusability.* IEEE Software, 1993. Volume 10 (Issue 3): p. 61 - 66.

[3] Frakes, W.B., Kang, K., *Software Reuse Status and Future.* IEEE Trans. On Software Engineering, 2005. 31. No 7: p. 529-536.

[4] Cleaveland, C., J., *Building Aplication Generators.* IEEE Software, 1988: p. 25–33.

[5] Mili, H., Mili, F & Mili, A, *Reusing Software: Issues and Research Directions.* IEEE Transactions on Software Engineering, 1995. Volume 21.

[6] Dijkstra, E.W., *On the Role of Scientific Thought.* Springer-Verlag, New York, 1982(Selected Writings on Computing: A Personal Perspective): p. 165-182.

[7] Parnas, D.L., *On the Design and Development of Program Families.* IEEE Transactions on Software Engineering, 1976. SE-2(1).

[8] Batory, D., et al., The Gen Voca Model of Software-System Generators. IEEE Software, 1994.

[9] Jarzabek, S., Knauber, P., *Synergy between Component-based and Generative Approaches.* Lecture Notes in Computer Science, Springer Verlag, 1999: p. 429-445.

[10] Zhang, H. Y., Jazarbek, S., Swe S. M., . *XVCL Approaches to Separating Concerns in Product Family Assets*. in Proceedings of the Third International Conference on Generative and Component-Based Software Engineering. 2001.

[11] Batory, D., *Scaling Step-Wise Refinement.* IEEE Transactions on Software Engineering, 2004. p. 1-17.

[12] Czarnecki, K. *Overview of Generative Software Development.* in In J.-P. Banâtre et al. (Eds.): Unconventional Programming Paradigms (UPP) LNCS 3566, 2005. Mont Saint-Michel, France.

[13] Saleh, M.and Gomaa, H., *Separation of Concerns in Software Product Line Engineering*. in International Conference on Software Engineering. 2005. St. Louis, Missouri.

[14] Jarzabek, S., Chapter from book "Software Maintenance and Evolution: Reused based Approach". 2007: AUERBACH Publications.

[15] Hursch, W.L. and C.V. Lopes. *Separation of Concerns*. in *Technical Report NUCCS-95-03*. 1995. College of Computer Science, Northeastern University.

[16] Smaragdakis, Y.and Batory, D., *Application Generators*. Software Engineering volume of the Encyclopedia of Electrical and Electronics Engineering. 2000: J. Webster, John Wiley and Sons.

[17] Batory, D. Domain Analysis for GenVoca Generators. in Proceedings. Fifth International Conference on Software Reuse. 1998. Los Alamitos, California.

[18] Biggerstaff, T.J. *Reuse Technologies and Their Niches*. in International Conference on Software Engineering. 1999. Los Angeles CA: ACM.

[19] Biggerstaff, T.J., A Characterization of Generator and Component Reuse Technologies. IEEE Software, 2001.

[20] Smaragdakis, Y., Huang, S. S., Zook, D., *Program Generators and the Tools to Make Them.* in ACM/SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation. Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation 2004. Verona, Italy.

[21] Bill Frakes, D.B., Ted Biggerstaff, Kyo Kang, *Panel: Linking Domain Analysis and Domain Implementation.* IEEE Software, 1998.

[22] Partsch H, Steinbrueggen R.., *Program Transformation Systems.* ACM Computing Surveys, 1983.

[23] Biggerstaff, T.J., *A New Architecture for Transformation-Based Generators.* IEEE Transactions on Software Engineering, 2004. **30**(12): p. 1036-1054.

[24] Batory, D., Johnson, C., Mc Donald, B. and Heeder, D. *Achieving Extensibility through Product-lines and Domain -Specific Languages: A Case Study*. ACM Transactions on Software Engineering and Methodology (TOSEM), 2002. **11**(2): p. 191-214.

[25] Jarzabek, S, Zhang, H.Y., Ru, S., Lam, V. T. and Zhenxin, S. *Analysis of Meta-programs: an Example.* Journal of Software Engineering and Knowledge Engineering, 2006. **16**(1): p. 77-101.

[26] Jarzabek, S., Shubiao, L. *Eliminating Redundancies with a "Composition with Adaptation" Metaprogramming Technique*. in Proc. European Software Egineering Conference and ACM SIGSOFT Symposium on the Foundations of Software EngineeringESEC-FSE'03. 2003. Helsinki.

[27] Pettersson, U. and Jarzabek, S. *Industrial Experience with Building a Web Portal Product Line Using Lightweight Reactive Approach*. in Proc. European Software Egineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering. 2005. Lisbon.

[28] Apel, S., T. Leich, and G. Saake. *Aspect Refinement in Software Product Lines. in In Aspects and Software Product Lines* (ASPL'05): An Early Aspects Workshop at SPLC-Europe'05. 2005. Rennes, France.

REFERENCES

Anastasopoules, M. a. G., C. (2001). Implementing Product Line Variabilities. Symposium on Software Reusability. *Proceedings of the 2001 symposium on Software reusability: putting software reuse in context* . Ontario, Canada.

Aquil Saleh, M. (2005). *Software Product Line Engineering Based on Web Services*. Information and Software System Engineering. Fairfax, Virginia, George Mason University. Doctoral of Philosophy: 277.

Basit, H., A., Rajapakse, Damith C. and Jazarbek, S. (2005). Beyond Templates: a Study of Clones in the STL and Some General Implications. *Conference on Software Engineering, ICSE'05*, Missouri, USA, ACM.

Batory, D., Singhal, V.,Sirkin, M. and Thomas, J. (1993). Scalable Software Libraries. *ACM SIGSOFT'93: Symposium on the Foundations of Software Engineering*, Los Angeles, California, ACM.

Baxter, I. D. (2002). *Transformation Systems: Generative Reuse for Software Generation, Maintenance and Reengineering*, Springer-Verlag Berlin Heidelberg.

Becker, M. *Mapping Variabilities onto Product Family Assets*.

Biggerstaff, T. J. (1998). "A Perspective of Generative Reuse. ." *Annals of Software Engineering 5*: 169-226.

Biggerstaff, T. J., Perlis, Alan, J (1989). *Software Reusability Concepts and Models*,

ACM Press and Addision Wesley.

Biggerstaff, T. J., Perlis, Alan, J (1989). *Software Reusability Concepts and Models*, ACM Press and Addision Wesley.

Cheong, Y. C. a. J., S. (1999). Frame-based Method for Customizing Generic Software Architecture. *Symposium on Software Reusability, SSR'99*, Los Angeles, USA.

Cleaveland, C., J. (1988). "Building Aplication Generators." *IEEE Software*: 25–33.

Crnkovic, I. (2003). Component-based Software Engineering-New Challenges in Software Development. 25th Int Conference Information Technology Interfaces IT 2003, Cavtat, Croatia.

Czarnecki, K. a. E., U. (1999). Components *and Generative Programming*, Springer-Verlag/ACm Press.

Czarnecki, K. a. E., U. (2000). *Generative Programming- Methods , Tools and Applications*. Boston MA, Addison-Wesley.

Frakes, W. B., Kang, K. (2005). "Software Reuse Status and Future." *IEEE Trans. On Software Engineering* 31. No 7: 529-536.

Goebl, W. (2000). *A Survey and a Categorization Scheme of Automatic Programming Systems. GCSE'99*, Springer-Verlag Berlin Heidelberg.

Harsu, M. (2002). *A Survey on Domain Engineering, Institute of Software Systems*, Tampere University of Technology: 48.

Hwang, H.-J. (2006). Domain Analysis for Components Based Developments International ConferenceComputational Science and Its Applications - ICCSA 2006 Glasgow, UK, *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg

Jacobson, I., Griss, M., Jonsson, P. (1997). *Software Reuse Architecture, Process and Organization for Business Success,* ACM Press.

Jaring, M. a. B., J. (2004). *Variability Dependencies in Product Family Engineering. PFE2003,* Springer-Verlag Berlin Heidelberg.

Jarzabek, S. (1995). ”From reuse library experiences to application generation architectures." *ACM*: 114-122.

Jarzabek, S. a. S., L. (2003). Adapting Redundancies with a "Composition with Adaptation" Meta-programming Technique. *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundation of Software Engineering,* Helsinki, ACM Press.

Jarzabek, S., Knauber, P. (1999). "Synergy between Component-based and Generative Approaches." *Lecture Notes in Computer Science*, Springer Verlag: 429-445.

Jawawi D. N. A., Rosbi Mamat and Safaai Deris, “Analysis Patterns for Component-based Development of Autonomous Mobile Robot Software.” *Proc. Of The 2nd International Conference on Mechatronics 2005,* Vol. 1., May 2005, Kuala Lumpur, pp. 185-192.

Jun, Y. a. J., Stan (2005). "Applying a Generative Technique for Enhanced Genericity and Maintainability on the J2EE Platform."

Krueger, C., W. (1992). "Software Reuse." *ACM Computing Surveys* 24(2): 132-183.

Lubars, M., D. (1991). Reusing Designs for Rapid Application Development. *ICC'91*.

Macala, R. M., Stuckey, L and Gross, D. (1996). "Managing Domain-Specific, Product-Line Development." *IEEE Software*.

Neighbors, J. M. (1980). Software Constructions Using Components. Department of Information and Computer Science, University of California, Irvine.

Nitto, E. a. F., A. (1997). *Product Lines: What Are the Issues*, IEEE Press.

Parnas, D. (1976). "On the Design and Development of Program Families." *IEEE Transactions on Software Engineering* SE-2(1): 1-9.

Rajapakse, D., C. and Jazarbek, S. (2005). An Investigation of Cloning in Web Application. International *Conference on Web Engineering (ICWE'05)*.

Saleh, M. a. G., H. (2005). Separation of Concerns in Software Product Line Engineering. *International Conference on Software Engineering*, St. Louis, Missouri.

Thibault, S. a. C., Charles (1997). A framework for application generator design. Symposium on Software Reusability, *Proceedings of the 1997 symposium on Software reusability*, Boston, United States, ACM Press New York, NY, USA

Zhang, H. J., S. and Yang, B. (2003). *Quality Perdiction and Assesment for Product Line*. Springer-Verlag Berlin Heidelberg: 681-695.

APPENDIX A

Papers Published from this Research Work

Shahliza Abd Halim, Dayang Norhayati Abg Jawawi and Safaai Deris, "Underlying Software Product Line Methods and Generator Implementation", *FSKSM Postgraduate Annual Research Seminar 2007 (PARS's 07)*, July 2007, Skudai.

Shahliza Abd Halim, Dayang Norhayati Abg Jawawi and Safaai Deris, "Generator Technologies Using Fundamental Methods in Software Product Line", *Malaysian Software Engineering Conference (MySEC2007)*, December 2007, Kuala Lumpur.