Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 53 (2004) 381–407
Science of Computer Programming
www.elsevier.com/locate/scico

XVCL: a mechanism for handling variants in software product lines

Hongyu Zhang[a,*], Stan Jarzabek[b]

[a] *School of Computer Science and Information Technology, RMIT University, Melbourne 3001, Victoria, Australia*
[b] *School of Computing, National University of Singapore, Lower Kent Ridge Road, Singapore 117543, Singapore*

Received 12 January 2003; received in revised form 10 April 2003; accepted 24 April 2003
Available online 24 July 2004

Abstract

Software reuse focused on product lines has emerged as one of the promising ways to increase software productivity and quality. XVCL (XML-based Variant Configuration Language) is a variability mechanism that we developed for handling variants in software product lines. We apply XVCL to develop product line assets (including the domain model, product line architecture and generic components) as a set of x-frames that are capable of accommodating both commonality and variability in a domain. Specific systems, members of a product line, can be constructed by adapting and composing x-frames. In this paper, we illustrate our approach using examples from our product line project on the Computer Aided Dispatch (CAD) domain.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Variability mechanism; XVCL; Frame technology; Software product lines

1. Introduction

In recent years, the software product line approach [3,4], initiated by Parnas back in the 1970s [15], has emerged as a promising way to improving software productivity and quality. A product line (also called a product family or system family) arises from situations when we need to develop multiple similar products for different clients, or from a single system over years of evolution.

* Corresponding author.
E-mail addresses: hongyu@cs.rmit.edu.au (H. Zhang), stan@comp.nus.edu.sg (S. Jarzabek).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2003.04.007

Members of a product line share many common requirements and characteristics. They may perform similar tasks, exhibit similar behavior or use similar technologies. While having much in common, members of a product line also differ in certain requirements, design decisions and implementation details. The variability stems from many sources such as customer's specific needs, mutability of the environment, system maintenance and system evolution. In the product line approach, we identify both commonality and variability in a domain, and build generic and adaptable assets such as the domain model, product line architecture and generic components. In the development of each specific product, we reuse the product line assets instead of working from scratch.

Variants (including functional variants, variant design decisions and implementation-level variants) result from the variability in a domain. Product line assets should be generic and flexible enough to accommodate the variants, and to be reusable across members of the product line. However, there could be a large number of variants in a product line. The explosion of possible variant combinations and complicated variant relationships make the manual, ad hoc accommodation and configuration of variants difficult. How to effectively handle variants in a product line is a major challenge faced by both product line researchers and practitioners.

An effective way to deal with the problem of handling variants is to design a variability mechanism [4,7] that supports automated customization and assembly of product line assets. We developed the XML-based Variant Configuration Language (XVCL) [9,18], a variability mechanism based on frame technology [1]. Using XVCL, we develop product line assets as a set of x-frames that are capable of accommodating both commonality and variability in a domain. X-frames represent domain knowledge in the form of product line assets. Specific systems, members of a product line, can be constructed by composing and adapting x-frames.

In this paper, we describe our XVCL-based approach to product line development. In our approach, we perform domain analysis and capture common and variant requirements for a product line in a feature diagram. We then build reusable product line assets (including the domain model, product line architecture and generic components). We apply XVCL to help us accommodate variants into product line assets and customize them to construct custom systems. This paper summarizes our practices in using XVCL as a variability mechanism.

We have applied our approach to many product line projects, including projects on Facility Reservation Systems (FRS), Key-Word-In-Context (KWIC) and Computer Aided Dispatch (CAD) product lines. The CAD project is a Singapore–Ontario joint project,[1] involving both academic and industrial partners. The project started in 2000. One of our industrial partners, SES Systems Pte Ltd, provided CAD application domain expertise. Other partners, including the National University of Singapore and Netron Inc., Toronto, provided expertise in software reuse. In this paper, we illustrate our approach using examples from the CAD project.

[1] This project was supported by research grant NSTB/172/4/5-9V1 funded by Singapore National Science and Technology Board and Canadian Ministry of Energy, Science and Technology.

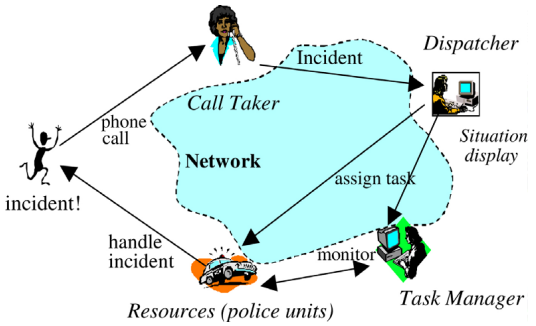

Fig. 1. The basic operational scenario in a CAD system for the police.

2. CAD domain analysis

Computer Aided Dispatch (CAD for short) systems are mission-critical systems that are used by the police, fire & rescue, health services and others. Fig. 1 depicts a basic operational scenario and roles of a CAD system for the police.

Once a Caller reports an incident, a Call Taker in command and a control center capture the details about the incident and the Caller, and create a task for the incident. The system shows the Dispatcher a list of undispatched tasks. The Dispatcher examines the situation, selects suitable Resources (e.g. police units) and dispatches them to execute the task. The Resources carry out the task instructions and report to the Task Manager. The Task Manager actively monitors the situation and at the end, closes the task.

2.1. Variants in the CAD domain

At the basic operational level, all CAD systems are similar—basically, they support the dispatch of units to handle incidents. However, there are differences across CAD systems relating to functional requirements, design decisions and implementation details. We classify them into three categories, such as functional variants, variant design decisions and implementation-level variants. Some of the variants that we have identified are as follows:

Functional variants

1. *Call Taker and Dispatcher roles* (CT-DISP for short). In some CAD systems, Call Taker and Dispatcher roles are separated (taken by two different people), while in other CAD systems the Call Taker and Dispatcher roles are taken by the same person. The CT-DISP variant has an impact on system functionalities. For example, in the former case, the Call Taker needs to inform the Dispatcher of the newly created task, but in the latter case, once the Call Taker creates a task, she/he can straightaway dispatch Resources (e.g., police units) for this new task.
2. *Validation* of the caller and task information differs across CAD systems. In some CAD systems, a basic validation check (i.e., checking the completeness of the Caller and Task information) is sufficient; in other CAD systems, validation includes duplicate task checking, VIP place checking, etc.; in yet other CAD systems, no validation is required at all.

3. *Dispatch algorithm*. There are different ways to dispatch Resources, using a shortest-distance search algorithm or a location code search algorithm.

Variant design decisions

4. *Database*. The database used in CAD systems could be either a centralized database or a distributed database.
5. *Encryption*. A decision on whether or not to encrypt the data sent between clients and servers.

Implementation-level variants

6. *Package*. CAD components for different systems may be arranged into different Java packages (directory structures).
7. *New attributes/methods*. We include anticipated attributes/methods into the classes. However, there may be additional class attributes/methods needed by a specific system, due to the new, unexpected changes in the requirements. For example, additional information about the Task and Caller required by certain systems can be added as new class attributes.

Besides the above variants, we have also identified other variants. In this paper, we will not describe all the variants in detail.

2.2. Capturing variants in a feature diagram

Feature diagrams [11] are often used to model commonality and variability in a domain. A feature diagram provides 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 (Czarnecki and Eisenecker also proposed the or-features [5]). Commonality can be modeled as mandatory features whose ancestors are also mandatory. Variability can be modeled as optional, alternative and or-features. For example, the "Call Taker and Dispatcher roles" requirement has two alternative variants: "Separated" and "Merged". The optional "Validation" requirement has two or-variants: "Basic Validation" and "Advanced Validation", which means that the "Validation" requirement can be "Basic Validation", "Advanced Validation" or both. Fig. 2 shows a fragment of the feature diagram for the CAD product line.[2]

3. Problems with handling variants during CAD product line development

We will accommodate the variants depicted at Fig. 2 in the development of a CAD product line. Variants may have impact on many product line assets, such as domain models, software architecture and components. In this section, we will describe the problems that

[2] A complete feature model consists of a feature diagram and other information about the features, including a semantic description, rationale, binding time, constraint and dependency. Fig. 2 does not show all the information associated with a feature model.

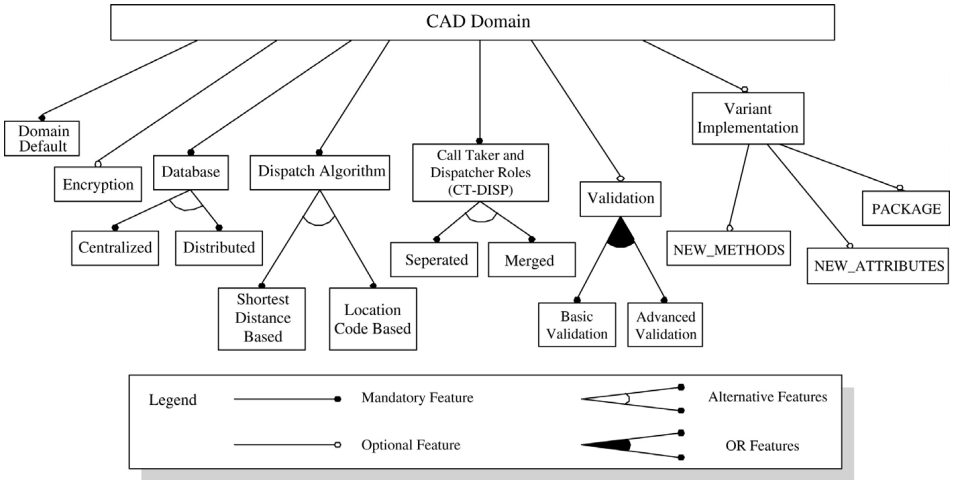

Fig. 2. The CAD feature diagram (partial).

we encountered in handling variants in product line development, and describe criteria for a good variability mechanism.

3.1. The impact of variants on the domain model

A domain model [20] describes common and variant requirements for a product line. It is like a system requirement specification except instead of modeling just one system, the domain model focuses on a family of systems. In our approach, it plays at least two important roles: first, during early stages of domain engineering, a domain model captures information that helps us design a product line architecture. Later, during reuse-based system engineering, a domain model describes the scope of functionality (both common and variant requirements) that has been implemented into a product line for reuse.

UML notations used for modeling a single system can be extended with "variation points" to cater for variant requirements [7]. We use UML and its extension mechanisms [16] to model the CAD domain [8]. Fig. 3 gives an example of a *Create Task* use case in the CAD domain model. Variant behaviors can be inserted into the "extension points" {Validation} and {CT-DISP}.

The extension mechanisms provided by UML are at the descriptive level (for modeling purpose only). The UML model for a single system could be complicated, especially for a large and complex system. Modeling variant requirements adds an extra level of complexity, making the UML model even more complicated. As the number of variants grows, domain models could become difficult to understand and customize, undermining the very purpose of domain modeling.

Let us examine the *Create Task* use case described above. The *Create Task* use case is rather small—it only includes two variation points. If two more variation points within the *Create Task* use case are identified, assuming each variation point has two possible

1. Introduction
Create Task allows a Call Taker to create a task for an incident reported by an emergency caller.
2. Flow of events
An emergency call is received
Include *Get Task Info* use case
Include *Get Caller Info* use case
(extension point **{VALIDATION}** here)
If validation failed then **Include** *Report Error* use case and abort the session.
Create a new task
(extension point **{CT-DISP}** here)
3. Special Requirement
Call Taker should respond to the emergency call within "**RESPONSETIME**".

Fig. 3. The *Create Task* use case in the CAD domain model.

values, the total number of variant combinations will be 2^4. The exponential explosion of possible variant combinations makes the manual customization (specialization) of the use case model difficult. In addition, the impact of variants is not limited to use cases but spreads over other domain model views. This example is based on a textual use case. In [8], we gave examples to illustrate that the same problem applies to graphical UML models too.

We believe that the above-mentioned problems cannot be solved at the description level alone. A variability mechanism is needed to help us alleviate the above-mentioned problems. Using the variability mechanism, we will be able to easily produce a customized domain model for a specific system. In application engineering, system analysts need only to understand the customized models they are interested in at a given moment without having to examine the entire domain model.

3.2. The impact of variants on software architecture and components

Variants also affect software architecture and components. At the architecture level, to accommodate a given variant, we may need to change the allocation of system functions to components, include new components into the system and/or modify components' interfaces.

Fig. 4 shows the component diagram of the CAD software architecture. Many variants have an impact on the CAD architecture. For example, for the CT-DISP variant, if the Dispatcher and Call Taker roles are played by one person, then the CallTaker UI and Dispatcher UI components in Fig. 4 can be merged into a single user interface component.

In many cases, addressing a variant also requires us to make certain changes to components' internal implementation. Components in the CAD product line must be generic and adaptable, capable of incorporating both commonality and variability in a domain. Table 1 shows the impact of variants on CAD components. From Table 1, we can see that it is difficult to localize the impact of variants—one component is often affected by many variants, and one variant affects many components.

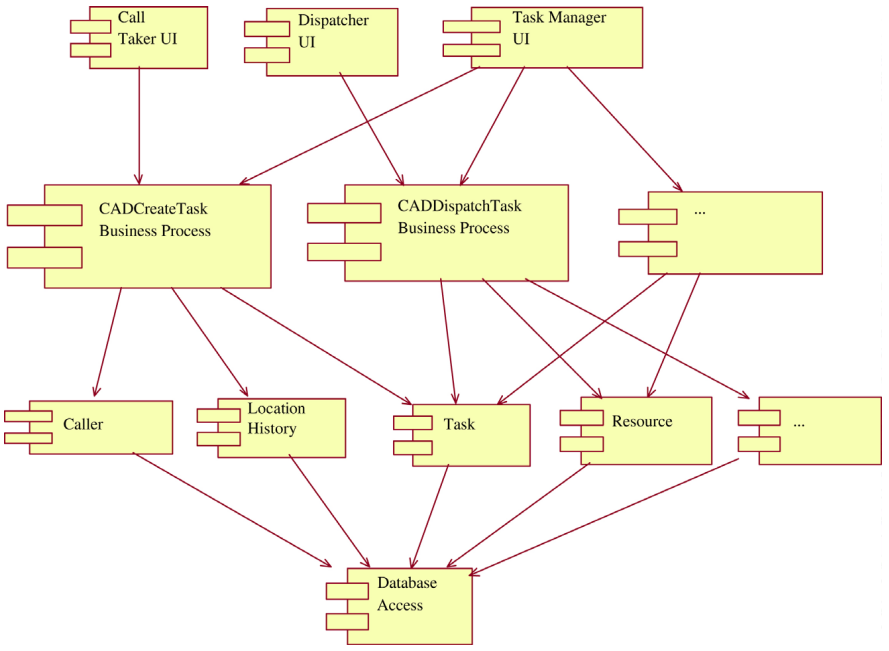

Fig. 4. The CAD software architecture.

Table 1
The impact of variants on CAD components (partial)

Variant Component	Validation	Dispatch Algorithm	CT-DISP	Encryption	NEW_METHODS/ NEW_ATTRIBUTES	...
Call Taker UI			X	X	X	
Dispatcher UI		X	X	X	X	
CADCreateTask	X		X	X	X	
CADDispatchTask		X		X	X	
Database Access					X	
...						

3.3. The criteria for a good variability mechanism

From the above analysis, we believe that a good mechanism for handling variants will have the following important properties:

- **Ability to automate the customization of product line assets**

As the number of variants grows, it is difficult to customize product line assets by hand. Thus, a variability mechanism should be able to automate the customization process, and rapidly produce customized product line assets on demand.

- **Ability to handle variants that have different levels of impact**

Variants may have different levels of impact. At architecture level, to address certain variants, we may need to include new components into the system, remove existing components from the system, modify components' interfaces or change the allocations of the functions to components. Thus for some product line members, their runtime software architecture could be different from the others. At component level, addressing a variant requires us to make certain changes to components' internal implementation. We need a mechanism that can effectively handle variants at both architecture level and code level.

- **Ability to handle variants that have different degrees of impact**

Variants may have different degrees of impact on product line assets. Some variants may have limited impact on one or a few components. To accommodate these variants, only one or a few components need to be changed. Other variants may have a wide impact on many components. To accommodate these variants, we should be able to trace the impact of variants across the components and make pervasive changes. For example, from Table 1, we can see that one component could be affected by many variants, and one variant may also affect many components. Very often, it is difficult to localize the impact of one variant into one component. A variability mechanism should be able to handle variants that have different degrees of impact. It should also enable us to separate changes required for accommodating different variants, therefore facilitating maintenance and evolution.

- **Ability to handle both anticipated variants and unexpected variants**

Variants can be generally classified into two categories based on their predictability. For some variants, we can anticipate the actual changes required for incorporating them. To satisfy a specific requirement, we can select one of the anticipated implementations of these variants. For other variants, we understand that they have impact on domain model or software architecture. However, we do not know the exact changes required to accommodate these variants as different systems may demand different implementations. We called these two categories of variants *anticipated* variants and *unexpected* variants. Addressing unexpected variants is essential in poorly understood and evolving domains where requirements are always changing.

- **Ability to handle variants in multiple product line assets**

The variability mechanism should have the ability to handle variants across multiple product line assets such as models, architecture, components and documentation. Many variants have impact on multiple product line assets (e.g., the CT-DISP variant affects the *Create Task* use case, software architecture and *CADCreateTask* component). These variants are traceable from feature diagram, to domain model, product line architecture and components. It is essential to ensure the traceability of variants so that the integrity of a product line can be maintained.

Traceability is easier to achieve if we apply a uniform variability mechanism across all the product line assets. Having a uniform mechanism also enables us to customize multiple product line assets for a specific system based on a single specification file. In this way,

we can reduce the error of having inconsistent customizations of product line assets for the same system.

- **Ability to work with contemporary software development paradigms**

The variability mechanism should be able to work with other contemporary software design methods and programming languages, such as object-oriented approaches, architecture-centric component-based development, UML modeling and the Java programming language.

4. XVCL: a variability mechanism

XVCL (XML-based Variant Configuration Language) is a general-purpose mark-up language that we developed for configuring variants in programs and other kinds of documents. We can apply XVCL to configure variants in a variety of software assets such as the domain model, program code and test cases. In fact, XVCL can be used for managing variants in any artifacts that can be represented as a collection of textual documents.

4.1. A brief history of XVCL

XVCL is based on the same concepts as the frame technology [1]. The concept of "frame" was first introduced by Marvin Minsky in 1975. In his paper entitled "A Framework for Knowledge Representation", Minsky described a frame as follows:

Here is the essence of the frame theory: When one encounters a new situation (or makes a substantial change in one's view of a problem) one selects from memory a structure called a "frame". This is a remembered framework to be adapted to fit reality by changing details as necessary [12].

According to Minsky, a frame may be viewed as a static data structure used to represent well-understood stereotyped situations. We adjust to new situations by calling up past experiences captured as frames. We then specially revise the details of these past experiences to represent the individual differences for the new situation.

Frame has been widely used as a structured knowledge representation scheme. In 1979, Paul Bassett applied the frame concept in the context of software engineering. Bassett's frames are analogous to the Minsky frames. Bassett's frames are reusable pieces of program code that can be adapted to meet specific needs. Bassett's use of frame hierarchies and default BREAKs was inspired by the "frames" and "slots" concepts proposed by Minsky [1].

Bassett's frame technology has been extensively applied by Netron Inc. to manage variants and evolve multi-million-line, COBOL-based information systems. While designing a frame architecture is not trivial, subsequent complexity reductions and productivity gains are substantial. An independent analysis showed that frame technology has reduced large software project costs by over 84% and their times-to-market by 70%, when compared to industry norms [1]. These gains are due to the flexibility of the resulting architectures and their evolvability over time. The excellent record of frame technology in large-scale software applications was the main reason that led us to implement XVCL. Designed in 1970s and 1980s, frame technology is very much influenced by the COBOL

language and does not address many contemporary design methods and language features. We believed that frame technology could be enhanced to blend into contemporary software development practices (such as architecture-centric, component-based product line development).

In 2000, together with Netron Inc., our research team at the Software Engineering Laboratory of the National University of Singapore developed XVCL (XML-based Variant Configuration Language), a new form of frame language that is free of COBOL heritage. In XVCL, Bassett's frame commands are designed as XML tags. Frames in XVCL are called x-frames, which are used to represent domain knowledge in the forms of product line assets (such as domain model or generic components). In September 2002, we made XVCL available at an Open Source forum [21], from which the latest XVCL language specification, processor and source code can be downloaded.

4.2. How XVCL works

XVCL works on the principle of constructing custom systems by composing generic, reusable meta-components (or asset fragments), after possible adaptations. Adaptations of a meta-component take place at designated variation points marked by XVCL commands. This "composition with adaptation" process turns meta-components into concrete components of the custom system that we wish to build, at construction time.

In XVCL, an x-frame is an XML file with program code (or other software artifacts) instrumented with XVCL commands (designed as XML tags) for ease of customization. X-frames represent domain knowledge in the forms of product line assets. XVCL commands allow the composition of the x-frames (via <adapt> command), and also make x-frames customizable by allowing one to select pre-defined options based on certain conditions (via <select> command), by marking breakpoints (slots) where additional changes can be inserted (via <break> and <insert> commands), and by providing variables and expressions as a parametrization mechanism (via <set> and <value-of> commands).

X-frames are organized into a layered hierarchy called an x-framework. X-frames at lower levels are building blocks of higher-level x-frames. At construction time, lower-level x-frames are composed into higher-layer x-frames after possible adaptation. The hierarchical x-framework enables us to handle variants at all granularity levels.

XVCL supports automated configuration of variants in product line assets. A configuration of required variants are recorded in a special x-frame (specification x-frame or SPC for short). Given the SPC, the *XVCL processor* traverses an x-framework, performs composition and adaptation by executing XVCL commands embedded in x-frames and constructs components of a specific system, a member of a product line.

Some major XVCL commands are given in the Appendix. For more details about XVCL, we refer the readers to our Website [21].

4.3. X-frame design principles

Separation of concerns

We design x-frames according to the principle of separation of concerns. For example, we design the Business Logic x-frames to encapsulate the business logic concerns, the

UI x-frames to encapsulate the user interface concerns, the error handling x-frames to encapsulate the error handling concerns. In addition to the separation of concerns that we can achieve using conventional design and programming techniques, using XVCL we can achieve more "advanced" separation of concerns.

Conventional design and programming techniques force developers to decompose a solution space into modules along a single, "dominant" dimension of concern [19]. For example, in object-oriented methods the dominant dimension is class (object) dimension, the solution space is decomposed into a set of classes. With procedural programming languages, the solution space is decomposed into a set of functions. Using XVCL, we can have arbitrary decomposition of solution space, without necessarily following the class or function dimension. An x-frame could encapsulate a class, a function or a code fragment at any granularity level that separates certain concerns.

XVCL provides composition mechanisms (via <adapt>s and <insert>s) that compose separate x-frames together at program construction time. Given SPC, the XVCL processor traverses the x-framework, performs composition and adaptation, and constructs a custom system meeting required variants.

Capturing common abstractions

During x-frame development, we also identify common abstractions and design x-frames to represent them. If we find that some components (or other asset fragments) have much in common, we design x-frames to capture the commonalities. We also instrument the x-frames with XVCL commands to accommodate the variabilities among the components. From the x-frames, different but similar components can be constructed. By capturing commonalities and variabilities among components, we reduce the redundant code within an x-framework, making the x-framework more compact and easy to maintain.

5. Applying XVCL to handle variants in a CAD product line

We apply XVCL as a mechanism for handling variants in a product line. X-frames are developed to represent generic, adaptable product line assets. In this section, we describe how we apply XVCL at domain model, component and architecture levels, using examples from the CAD project. We also describe how we customize the x-frames to construct specific systems.

5.1. Applying XVCL to handle variants in the CAD domain model

We instrument (mark-up) domain models with XVCL commands, transforming the domain model into a set of x-frames [8]. Fig. 5 shows an x-frame for the *Create Task* use case (Fig. 3). XVCL commands are shown in bold. The <x-frame> tag denotes the x-frame for *Create Task* use case. The <set> command defines an XVCL variable RESPONSETIME with default value of "30 secs". The <adapt> command indicates the UML ≪include≫ relationship. When the XVCL processor encounters the <adapt> command, it will customize and include a copy of the specified x-frame (e.g., *Get_Task_Info.uc*) into this x-frame. The <break> command indicates the variation

```
<x-frame name="x_CreateTask.uc">
<set var="RESPONSETIME" value="30 secs"/>
1. Introduction
Create Task allows a Call Taker to create a task for an incident reported by an
emergency caller.
2. Flow of events
  An emergency call is received
  <adapt x-frame="Get_Task_Info.uc"/>
  <adapt x-frame="Get_Caller_Info.uc"/>
  <break name="Validation"/>
  If validation failed then <adapt x-frame= "Report_Error.uc"/>
                                          and abort the session.
  Create a new task
  <select option="CT-DISP">
    <option value="SEPARATED">
      <adapt x-frame="InformDispatcher.uc"/>
    </option>
    <option value="MERGED">
      <adapt x-frame="DispatchTask.uc"/>
    </option>
  </select>
3. Special Requirement
Call Taker should respond to the emergency call within
  <value-of expr ="?@RESPONSETIME? "/>.
</x-frame>
```

Fig. 5. The x-frame for the *Create Task* use case.

point where additional customizations may occur. In this example, it indicates the variation point brought up by the variant requirement Validation. Use case segments that are related to Validation variant may be <insert>ed into/after/before this variation point during customization. The <select> command is used to indicate the variation point where anticipated customization will occur. In this example, the customization of CT-DISP variant is denoted by the <select> command. The reader may find the variants described in this section in the feature diagram of Fig. 2.

For graphical models that cannot be directly manipulated by the XVCL processor, we first convert models into the equivalent textual representation, and then instrument the text with XVCL commands. For example, we convert the UML diagram into the equivalent representation in XMI [14] before framing. XMI supports the round-trip transformation of UML models from diagrams to XML files without loss of information.

To customize the domain model represented as x-frames, we design a specification x-frame (SPC) that records a specific variant configuration required by a particular system. Given the SPC, the XVCL processor can promptly provide systems analysts with the

customized domain model (i.e., the model for a specific system). We will describe the customization of the x-frame in Section 5.4.

We believe that our approach provides an automated way of managing domain models. XVCL and its processor help us alleviate the problem of having overly complicated domain models. Rather than working with models by hand, a time-consuming and error-prone process, our approach automates the process of producing a customized domain model for selected variants (i.e., requirement specifications for a product line member that satisfies those variants).

5.2. Applying XVCL to handle variants in CAD components

Using XVCL, we design generic, adaptable components as x-frames that incorporate both domain defaults and variants. The resulting x-frames are meta-components, from which concrete components are constructed during application engineering (the process of producing a specific system using the reusable assets).

Fig. 6 shows the x-frame for constructing the CADCreateTask component in Fig. 4, which contains a control class for creating a task. To accommodate variants, we instrument the CADCreateTask component with XVCL commands (highlighted as bold lines in Fig. 6).

For the "Call Taker and Dispatcher roles" (CT-DISP) variant, we accommodate it by using a `<select>` command (`<select option="CT-DISP"/>`), which indicates the variation point where anticipated customization will occur. When the XVCL processor encounters a `<select>` command, it will select appropriate contents based on the value of "CT-DISP".

The `<break name="Validation">` command indicates the variation point brought up by the optional variant requirement Validation. Code that is related to the Validation variant may be `<insert>`ed into this variation point during x-frame adaptation.

XVCL variables, such as "PACKAGE", provide another means of handling variants. We may want to put components of different CAD systems into different Java packages. We define a meta-variable "`PACKAGE`" to represent the package name, with the default value of "BusinessLogic". In Fig. 6, we use an XVCL command `<value-of expr="?@PACKAGE?"/>` to indicate a place holder at which the actual value of `PACKAGE` is filled in during x-frame adaptation.

Unexpected changes in requirements in the future may require us to add new attributes/methods to classes in CAD components. To accommodate these unexpected changes into the CADCreateTask component, we introduce two breakpoints `CREATETASK_NEW_ATTRIBUTES` and `CREATETASK_NEW_METHODS`. As we may also need more Java packages, we introduce a breakpoint `CREATETASK_NEW_IMPORTS`. In addition, we introduce a breakpoint `CREATETASK_NEW_PARAMETERS` to indicate a slot at which more XVCL parameters can be inserted. Specific code can be inserted into breakpoints by `<insert>` commands defined in the higher-level x-frames (e.g., in SPC) during x-frame adaptation.

The development of x-frames is an incremental process. We start by examining domain defaults, and then inject variabilities into the defaults incrementally until all variants are addressed.

```
<x-frame name="x_CreateTask" language="java">

<set var="PACKAGE" value="BusinessLogic"/>
<break name="CREATETASK_NEW_PARAMETERS"/>

package <value-of expr="?@PACKAGE?"/>
      ...
import java.util.*;
<break name="CREATETASK_NEW_IMPORTS"/>

public class CADCreateTask {
    private Caller    aCaller;
    private Task      aTask;

<break name="CREATETASK_NEW_ATTRIBUTES"/>

    public Caller GetCallerInfo() {
        ...                  // code about capturing Caller's info
      return aCaller;
    }
       ...
    public int SaveTask() {
      ...                    // code about saving a task

<break name ="Validation"/>

      int nTaskID = aTask.Save();
      return nTaskID;
    }

<select option="CT-DISP">
  <option value="SEPARATED">
    <adapt x-frame = "InformDispatcher"/>
  </option>
</select>

<break name="CREATETASK_NEW_METHODS"/>
}
</x-frame>
```

Fig. 6. The *x_CreateTask* x-frame.

5.3. Applying XVCL to construct CAD product line architecture

Product line architectures [2,3] are designed for software product lines instead of one-of-a-kind products. They are more general than the software architecture for a single

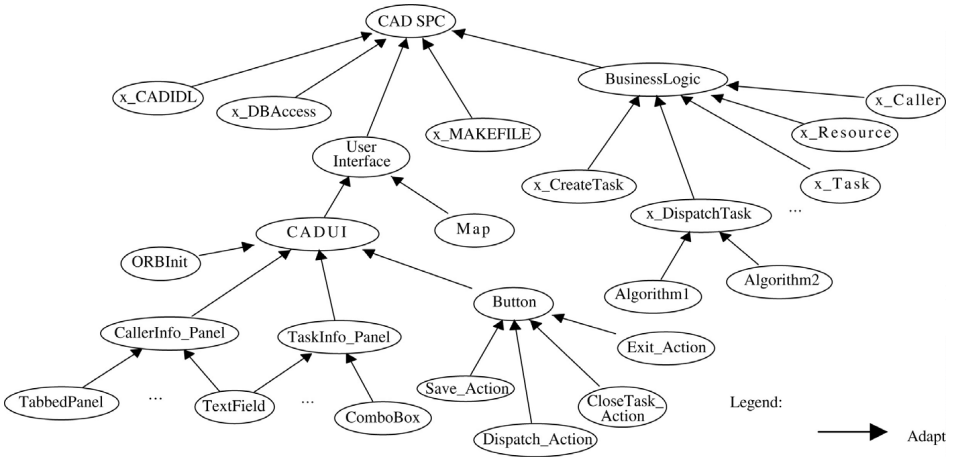

Fig. 7. The CAD x-framework.

system because they are engineered to be reusable, extendable and configurable. For software architecture of a single system, we focus on its runtime behavior (the interacting components that are working at system runtime, as described in [17]). For product line architecture, we are more concerned about the construction-time architecture—the generic architecture that can generate product line members having runtime software architectures.

We develop a product line architecture based on the runtime software architecture, and realize the product line architecture as an x-framework—a hierarchy of x-frames. Fig. 7 shows the CAD x-framework. Being built of x-frames, the CAD product architecture has the ability to accommodate both commonality and variability in the CAD domain.

The x-frames in Fig. 7 are designed according to the principle of separation of concerns. For example, the *x_CreateTask* x-frame separates the concern of creating a task (containing a control class), the *Button* x-frame separates the concern of creating Java buttons (containing code fragments for button definition, initialization and action). The root of the CAD x-framework is the SPC, which separates configuration knowledge for building a specific CAD system.

We also identify common abstractions and design x-frames to represent them. For example, we find that CAD UI components, such as Call Taker UI and Dispatcher UI, have much in common: basically, they contain groups of elementary UI components and perform actions when these elementary components are activated. The differences are in the specific properties (such as number, name, layout) of the elementary components and in the ways they respond to the activations (i.e., specific implementations of the actions). The commonalities among UI components inspire us to design a generic *CADUI* x-frame for constructing these components.

An x-framework incorporates variants that have architectural level impact. An example of these variants in a CAD product line is the CT-DISP variant. If the Dispatcher and Call Taker roles are played by one person, then the CallTaker UI and Dispatcher UI components can be merged into a single user interface component. As illustrated by Fig. 8,

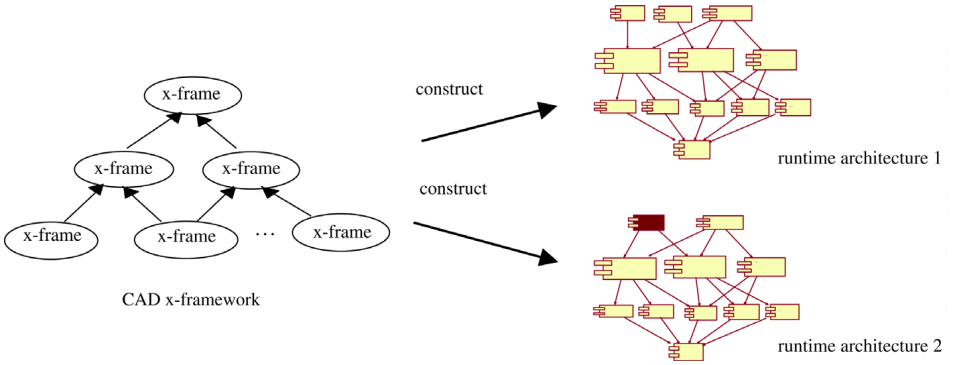

Fig. 8. Constructing different runtime architectures from an x-framework.

an x-framework should be able to construct systems having different runtime software architectures. In Fig. 8, two CAD runtime architectures are constructed from the CAD x-framework. Runtime architecture 2 includes a merged component while architecture 1 does not.

Fig. 9 shows how we incorporate the CT-DISP variant at architectural level by using XVCL. The inclusion/exclusion of the components is achieved by a <select> command.

The *UserInterface* x-frame constructs CAD user interface components based on the value of the variable CT-DISP. If the value of CT-DISP is SEPARATED, two separated UI components are constructed by adapting a generic x-frame *CADUI*. Otherwise, one merged UI component is constructed.

5.4. Customizing CAD x-frames

The x-frames that we developed are assets of an organization. In reuse-based application engineering, we reuse these x-frames instead of developing each individual system from scratch.

To develop a specific system, we first examine the feature diagram (such as Fig. 2) to understand how many variants are identified in the product line, and to select required variants. This is done by first analyzing customer's requirements for a specific system, and then finding a matching set of features (variants) from the feature diagram. It is important that application engineers work with customers to select the variants that the customers really want. Some variants may be mutually dependent—the selection of one variant may be dependent on the selections of other variants. Also, different variant configurations may lead to systems that satisfy the same set of functional requirements, but differ in certain quality attributes. At this stage, we should be careful about the decisions made, as not all configurations of variants are valid or "good" with respect to the functional and quality requirements.

Having selected variants for a specific system, we record the variant configuration in an SPC, as well as possible additional implementations needed by certain variants. Given the SPC, the XVCL processor adapts and composes necessary x-frames, and constructs systems satisfying the specific requirements.

```
<x-frame name="UserInterface">

<select option="CT-DISP">
  <option value="SEPARATED">
    <set var="UI_Name" value="CallTakerUI"/>
    <adapt x-frame="CADUI.xvcl" outfile="FINAL/CallTakerUI.java">
        ...                          // adapting CADUI x-frame for constructing
                                     // CallTaker UI component
    </adapt>

    <set var="UI_Name" value="DispatcherUI"/>
    <adapt x-frame="CADUI.xvcl" outfile="FINAL/DispatcherUI.java"/>
        ...                          // adapting CADUI x-frame for constructing
                                     // Dispatcher UI component
    </adapt>
  </option>

  <option value="MERGED">
    <set var="UI_Name" value="MergedUI"/>
    <adapt x-frame="CADUI.xvcl" outfile="FINAL/MergedUI.java">
        ...                          // adapting CADUI x-frame for constructing
                                     // Merged UI component
    </adapt>
  </option>
</select>

</x-frame>
```

Fig. 9. The UserInterface x-frame.

We can design a single specification x-frame (SPC) for customizing multiple product line assets. Fig. 10 shows a partial SPC for customizing the generic CAD use case (*x_CreateTask.uc*) and meta-component (*x_CreateTask*). The value of the CT-DISP variable is set to "SEPARATED", which means that the Call Taker and Dispatcher roles are separated. The <insert> command inserts specific code or a use case fragment into the breakpoint Validation, to satisfy the requirement of "Basic Validation".

Given the SPC in Fig. 10, the XVCL processor adapts the x-frames for the generic *Create Task* use case and component, customizes them according to the instructions given as XVCL commands, and constructs a specific use case and component that meet the specific requirements ("Basic Validation" and "Separated Call Taker & Dispatcher Roles") of a CAD system. Fig. 11 shows the customized *Create Task* use case description. Fig. 12 shows the generated code of the *CADCreateTask* component. Although the variants CT-DISP and Validation have cross-cutting impact within and across product line assets, with the aid of the XVCL processor we avoid manual and inconsistent customization of product line assets.

```
<x-frame name="CAD_SPC">
 <set var="Database" value="Centralized"/>
 <set var="Encryption" value="NO"/>
        ...
 <set var= "CT-DISP" value = "SEPARATED"/>
                          // adapting a generic use case
 <adapt x-frame="x_CreateTask.uc" outfile="CreateTask.uc">
   <insert break="Validation">
      Perform basic validation checking
   </insert>
 </adapt>
                        // adapting a generic component
 <adapt x-frame="x_CreateTask"  outfile="CADCreateTask.java">
   <insert break="Validation">
     if (!aTask.BasicValidation())      // code about Basic Validation
        return -1;
     if (!aCaller.BasicValidation())
        return -1;
   </insert>
 </adapt>
    ...
</x-frame>
```

Fig. 10. A partial SPC for a CAD system.

SPCs such as the one given in Fig. 10 only contain configuration knowledge for one specific system. To develop other CAD systems that meet different requirements, we can design different SPCs that specify values for anticipated variants and include the implementation for additional changes. In this way, we develop specific CAD systems, members of the CAD product line.

Currently, we write SPCs manually. However, it is possible to develop a GUI-based "wizard" tool to help us generate SPCs from the feature diagram.

6. Related work

The simplest way to handle variants in a product line could be "copy-and-modify". When programmers are called upon to write a new program, naturally they look for a similar one and copy it. They then modify the code to implement variants. Although a step in the right direction, the "copy-and-modify" method is rather ad hoc and difficult to scale up.

It is also possible to implement all anticipated variants into a program directly and to select required variants by using runtime variables or configuration files. This method incurs redundant code and unnecessary performance penalties.

1. Introduction
Create Task allows a Call Taker to create a task for an incident reported by an emergency caller.
2. Flow of events
An emergency call is received
(content of *Get Task Info* use case)
(content of *Get Caller Info* use case)
Perform basic validation checking
If validation failed then (content of *Report Error* use case) and abort the session.
Create a new task
(content of *Inform Dispatcher* use case)
3. Special Requirement
Call Taker should respond to the emergency call within "30 secs".

Fig. 11. The customized *Create Task* use case description.

There are many variability mechanisms available, such as conditional compilation, template meta-programming, object-oriented inheritance, delegation and aspect-oriented programming. Most of them can be used to accommodate variants into code, but cannot address other product line assets such as the domain model, documentation and test cases. In contrast, XVCL can be applied to configure variants in a variety of product line assets. In this section, we briefly compare XVCL to other variability mechanisms.

6.1. Conditional compilation

Conditional compilation has been widely used in C/C++ programs. It is also included into the newly developed C# language. Traditionally, conditional compilation is applied to handle machine or programming environment related variants.

Conditional compilation enables desired code segments to be included in or excluded from a program. Pre-processing directives (such as #ifdef, #else and #endif) mark the variation points in the program. Similar to the XVCL <select> command, the #ifdef directive supports selection of implementations based on the configuration of variants. Similar to the XVCL <adapt> command, the #include directive supports separation and composition of variant implementations.

Conditional compilation is an old and unsophisticated variability mechanism. Unlike XVCL, conditional compilation does not support variable references, expressions and scoping rules; thus it is less flexible. More importantly, conditional compilation does not support breakpoints that indicate slots where additional code can be inserted. This makes conditional compilation incapable of handling unexpected variants (the variants whose implementations are uncertain at the time at which the breakpoints are defined).

6.2. Template meta-programming

Template meta-programming [5] is a generative programming technique. It is largely based on the C++ template mechanism that is a part of the ISO/ANSI C++ standard.

```
package BusinessLogic;
      ...
import java.util.*;

public class CADCreateTask {
    private Caller    aCaller;
    private Task      aTask;

    public Caller GetCallerInfo() {
          ...              // code about capturing Caller's info
       return aCaller;
    }
          ...
    public int SaveTask() {
          ...
       if (!aTask.BasicValidation())    // code about Basic Validation
          return -1;
       if (!aCaller.BasicValidation())
          return -1;

       int nTaskID = aTask.Save();
       return nTaskID;
    }
    public int InformDispatcher (Task aTask) {
          ...              // code about informing dispatcher of a newly created task
       return 0;
    }
}
```

Fig. 12. The generated *CADCreateTask* component.

The template mechanism is C++'s implementation of the concept of genericity (parametrized types or parametric polymorphism). Templates are much the same as parametrized classes or functions. They are heavily exploited in the C++ Standard Template Library (STL), in which a class is constructed from a composition of an Algorithm, a Container, an Element and several Adapter elements [13]. The template and other C++ features constitute a compiler-time sublanguage of C++, enabling meta-program development.

XVCL supports template-like parametrization (via <set> and <value-of>), and template-like meta-programming (via <select> and <while>). In addition, XVCL can also accommodate unexpected changes (via <break> and <insert>), which is not easy for template meta-programming. In XVCL, additional code that caters for unexpected changes can be inserted into the breakpoints (slots) defined in an x-frame during customization. X-frames may be viewed as highly configurable templates with breakpoints.

Another limitation of the template meta-programming is that it is tightly coupled with the programming language such as C++, while XVCL is programming language independent. In fact, XVCL can be applied to any software artifacts that have a textual representation.

6.3. Inheritance

Inheritance is object orientation's primary variability mechanism. By inheritance, we mean the ability of a subclass to inherit from a more generalized superclass. It is well known that inheritance has limitations as well [1,7]. Sometimes, even when we only want to inherit a few properties from a superclass, inheritance will only permit a subclass to inherit all the properties of its superclass, thus causing much redundant code. In addition, many OO programming languages, such as Java, do not support generics directly. If the number of variants increases, the class hierarchy will expand quickly, causing the maintenance and scaling problems.

XVCL complements OO programming by providing generic solutions at program construction time. Concrete classes can be constructed from the composition and adaptation of x-frames. Using XVCL, we avoid uncontrolled growth of classes in size and number. An XVCL experiment on the Java Buffer library shows that our Buffer x-framework contains less than 40% of the code that we find in the original Java Buffer classes (details of this experiment can be found at [21]).

XVCL also supports arbitrary decomposition and composition of a program (such as having generic data structures defined independently of the generic methods that use them) without necessarily following the class boundary; thus XVCL can handle variants in any granularity levels.

6.4. Delegation

In object-oriented languages, delegation is a technique that enables objects to handle a request by delegating operations to its delegate. In most of OO programming languages, delegation is realized as object composition: keeping a reference to a delegation object in the delegating object. To accomplish a task, the delegating object invokes operations in the delegation object.

Delegation is an alternative to inheritance. The delegating object can contain common behaviors, while the delegation objects support variant behaviors. Once the delegating object receives a request that it cannot satisfy, it forwards the request to corresponding delegation objects. Delegation is widely used in many design patterns [6] to handle variability. Examples of these patterns include the State, Strategy and Visitor patterns. Wrapping [3] can be also considered as a special use of delegation.

Like x-frames, the delegating objects contain reusable knowledge and are extensible. The difference is that delegation composes behaviors at runtime, whereas XVCL supports adaptation and composition at program construction time. Being a runtime mechanism, delegation makes software more flexible. However, when the number of variants increases, the number of required delegation objects may grow quickly. There are also runtime inefficiencies with delegation.

6.5. Aspect-oriented programming

Recent work focuses on advanced separation of concerns [10,19]. Concerns in multiple dimensions may spread throughout the whole program and cannot be nicely confined to a small number of components. A number of approaches have been proposed to address cross-cutting concerns and concern compositions. In aspect-oriented programming [10], each computational aspect (and variant implementation) can be programmed separately and rules are defined for waving aspects with the base code (typically object-oriented classes).

Like AOP, XVCL allows the separation of concerns from the base code. They all provide mechanisms for composing concerns at program construction time. Unlike AOP, in XVCL we explicitly mark the points where specific code (or other reusable assets) can be inserted. Furthermore, AOP has a restricted set of jointpoints, whereas breakpoints in XVCL can be defined anywhere within the program.

An advantage of AOP is that it does not require the existing programs to be modified before weaving can begin, while XVCL requires additional effort on framing the existing programs.

7. Conclusion

In this paper, we described XVCL, a variability mechanism for handling variants in software product lines. XVCL is based on the concepts of frame technology [1]. Using XVCL, we develop generic, adaptable product line assets as x-frames. Specific systems, members of a product line, can be constructed by adapting and composing the x-frames. We illustrated our approach using examples from the CAD product line project.

We believe that XVCL meets the criteria for a good variability mechanism:

- Ability to automate the customization of product line assets. Being transferred into a set of machine-processable x-frames, product line assets can be easily customized with the aid of the XVCL processor. XVCL allows the separation of configuration knowledge (SPCs) and reusable assets (x-frames). Given different SPCs, similar but different products can be constructed from the same set of x-frames.
- Ability to handle variants at both architecture and component levels. During x-frame and x-framework design, we accommodate variants that have impact on components and software architecture. From x-frames, specific runtime software architecture and components that populate it can be constructed.
- Ability to handle variants that have global impact on a system. The impact of such a variant (such as the CT-DISP variant) cannot be nicely localized into a single component identified through conventional modularization approaches such as OO analysis and design. We accommodate these variants by instrumenting components with XVCL commands, and by designing lower-level x-frames that separate the impacts of variants. Although these variants have impact on many x-frames, the x-frame processor automates the composition and adaptation process so that the transition from the x-frames to concrete components is transparent to programmers, releasing programmers from handling variants manually.

- Ability to handle both anticipated and unexpected variants. The XVCL <select> command selects one option from a set of anticipated implementations, while the <break> command indicates a slot which new, unexpected changes can be inserted into. Its ability to address unexpected variants makes XVCL an ideal choice for handling variants in domains where requirements are always changing.
- Ability to handle variants in multiple product line assets. We use XVCL as a single, consistent variability mechanism for handling variants across many product line assets (including both models and code). Inconsistent customization of product line assets can be avoided by using a single SPC.
- Ability to work with a contemporary software development paradigm. XVCL can be blended into contemporary software development practices (such as architecture-centric, component-based product line development). Although x-frames in our examples contain UML models and Java code, XVCL is modeling and programming language independent.

The limitations of our approach are:

- Understandability of x-frames. Being generic, x-frames could be difficult to understand. The verbose XML syntax also has negative impact on understanding x-frames. These problems can be mitigated by documenting the design rationale, by carefully designing x-frameworks according to XVCL design rules and by re-factoring the design as an x-framework evolves. We are also developing an XVCL Workbench to facilitate x-framework development. The XVCL Workbench includes tools such as a smart x-frame editor that hides XML syntax and displays graphical views of x-frames, and a static analysis tool that helps us understand an x-framework.
- Additional effort is needed for transferring existing assets into x-frames. To apply an XVCL-based approach to product line development, we need to "frame" an organization's legacy assets and transfer them into x-frames. This requires additional efforts, whereas other methods such as aspect-oriented programming do not require direct modifications of existing programs.

Note that there are also economical, managerial and organizational issues in the development of a product line, such as market analysis, strategic planning and organization structure. These issues are all very important for the product line approach to succeed in industrial practices. However, in this paper, we have only discussed the technical aspect of the product line development, concentrating on a mechanism for handling variants in product lines.

8. Future work

In the future, we plan to:

Experiment on a larger scale

We will extend the scope of the experimentation, applying our approach to larger-scale industrial projects in a variety of application domains.

Incorporate constraint management

Variants may be mutually dependent or mutually exclusive. Customers may also specify constraints on variants. During reuse-based application engineering, it is important to address variants' dependencies and constraints to ensure the coherence of the final product. Currently our approach relies on human engineers (together with customers) to select a valid configuration of variants. We believe that a tool can be designed to automate the validation process.

Investigate methods for addressing quality attributes

Different configurations of variants may lead to systems that satisfy the same set of functional requirements, but differ in certain quality attributes (also referred to as non-functional requirements) such as reusability, modifiability and performance. Methods and tools will be developed to help the system architect make rational decisions on selecting a "good" configuration of variants for a specific product line member. In [23], we proposed a knowledge-based approach to quality prediction and assessment for product lines.

Explore an XVCL-based solution to advanced separation of concerns

Can the implementation of variants (and other concerns) be separated into different x-frames and then be composed together? In [22], we report our initial experiments on an XVCL-based solution to advanced separation of concerns. We will continue exploring this area.

Develop an integrated construction environment for product line development

We will investigate an integrated construction environment that automates the construction and evolution of a software product line, covering the whole XVCL-based development process from feature modeling to product line architecture development.

Acknowledgements

This work was supported by research grant NSTB/172/4/5-9V1 funded under the Singapore–Ontario Joint Research Program by the Singapore National Science and Technology Board and Canadian Ministry of Energy, Science and Technology. We thank Dr. Zhang Weishan and many NUS students for their contributions to the CAD product line development. We thank Ulf Pettersson of SES Systems Pte Ltd for providing us with CAD domain knowledge. We also wish to acknowledge the valuable and insightful comments from the anonymous reviewers.

Appendix. Description of essential XVCL commands

Table 2 below summarizes major XVCL commands. The reader can find the full specification of XVCL from our Website at http://fxvcl.sourceforge.net.

Table 2
Essential XVCL commands

Syntax	Command definition
<**x-frame name** = "*name*" > *x-frame body* </**x-frame**>	The <x-frame> command denotes the start and end of an x-frame. An *x-frame body* contains textual contents (e.g., program code) usually instrumented with XVCL commands to make them adaptable.
<**adapt x-frame** = "*name*"> </**adapt**>	The <adapt> command instructs the processor to: • adapt the named x-frame and its descendants, • emit/assemble the customized content into the output, • resume processing of the current x-frame after processing the named x-frame and its descendants.
<**break name** = "*break-name*"> *break-body* </**break**>	The <break> command marks a breakpoint (slot) at which changes can be made by ancestor x-frames via <insert>, <insert-before> and <insert-after> commands. The *break-body* defines the default code, if any, that may be replaced by <insert> or extended by <insert-before> and <insert-after> commands.
<**insert break** = "*break-name*"> *insert-body* </**insert**> <**insert-before break** = "*break-name*"> *insert-body* </**insert-before** > <**insert-after break** = "*break-name*"> *insert-body* </**insert-after** >	The <insert> command replaces the breakpoint "*break-name*" in the adapted x-frame with the *insert-body*. The <insert-before> command inserts the *insert-body* **before** the breakpoint "*break-name*" in the adapted x-frame. The <insert-after> command inserts the *insert-body* **after** the breakpoint "*break-name*" in the adapted x-frame. The *insert-body* may contain a mixture of textual content and XVCL commands.
<**set var** = "*var-name*" **value** = "*value*"/>	The <set> command assigns a "*value*" defined in the "value" attribute to a single-value variable "*var-name*" defined in the "var" attribute.
<**set-multi var** = "*var-name*" **value** = "*value1, value2, . . .* " />	The <set-multi> command assigns multiple values (*value1, value2, . . .*) defined in the "value" attribute to a multi-value variable "*var-name*" defined in the "var" attribute.
<**value-of expr** = "*expression*" />	The value of the "*expression*" is evaluated and the result replaces the <value-of> command. A name expression starts with "?@" and ends with a "?".
<**select option** = "*var-name*"> <**option value** = "*value*"> (0 or more) *option-body* </**option**> <**otherwise**> (optional) *option-body* </**otherwise**> </**select**>	The <select> command selects from a set of options based on the value of variable "*var-name*": <**option**> is processed, if value of "*var-name*" matches <option>'s "*value*"; <**otherwise**> is processed, if none of the <option>'s "*value*" is matched. The *option-body* may contain a mixture of textual content and XVCL commands.

(continued on next page)

Table 2 *(continued)*

Syntax	Command definition
<while using-items-in = "*multi-var*"> *while-body* **</while>**	The <while> command iterates over the *while-body* using the values of multi-value variable "*multi-var*" defined in the "using-items-in" attribute. The *i*-th iteration uses *i*-th value of the "*multi-var*". Inside *while-body*, *multi-var* with the *i*-th value can be used as a single-value variable. The *while-body* may contain a mixture of textual content and XVCL commands.

References

[1] P. Bassett, Framing Software Reuse—Lessons from the Real World, Yourdon Press, Prentice-Hall, NJ, 1997.
[2] D. Batory, Product-Line Architectures, Invited presentation, Smalltalk und Java in Industrie and Ausbildung, Erfurt, Germany, October, 1998.
[3] J. Bosch, Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach, Addison-Wesley, MA, 2000.
[4] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, MA, 2001.
[5] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, MA, 2000.
[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, MA, 1995.
[7] I. Jacobson, M. Griss, P. Jonsson, Software Reuse: Architecture, Process and Organization for Business Success, Addison-Wesley, MA, 1997.
[8] S. Jarzabek, H. Zhang, XML-based method and tool for handling variant requirements in domain models, in: Proc. of the Fifth IEEE International Symposium on Requirements Engineering, RE'01, IEEE Press, 2001, pp. 166–173.
[9] S. Jarzabek, P. Bassett, H. Zhang, W. Zhang, XVCL: XML-based variant configuration language, in: Proc. of 25th International Conference on Software Engineering, ICSE'03, IEEE Press, 2003, pp. 810–811.
[10] G. Kiczales et al., Aspect-oriented programming, in: Proc. 11th European Conference on Object-Oriented Programming, ECOOP'97, Lecture Notes in Computer Science, vol. 1241, Springer, Berlin, 1997, pp. 220–242.
[11] K. Kang et al., Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report, CMU/SEI-90-TR-21, SEI, CMU, November, 1990.
[12] M. Minsky, A framework for representing knowledge, in: P. Winston (Ed.), The Psychology of Computer Vision, McGraw-Hill, New York, 1975.
[13] D.R. Musser, A. Saini, STL Tutorial and Reference Guide, Addison-Wesley, MA, 1996.
[14] OMG, XML Metadata Interchange (XMI) 1.1 RTF, OMG Document ad/99-10-02, 25 October, 1999.
[15] D. Parnas, On the design and development of program families, IEEE Transactions on Software Engineering 2 (16) (1976) 1–9.
[16] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language—Reference Manual, Addison-Wesley, MA, 1999.
[17] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, NJ, 1996.
[18] M.S. Soe, H. Zhang, S. Jarzabek, XVCL: a tutorial, in: Proc. of 14th International Conference on Software Engineering and Knowledge Engineering, SEKE'02, ACM Press, 2002, pp. 341–349.
[19] P. Tarr, H. Ossher, W. Harrison, S. Sutton, N Degrees of separation: multi-dimensional separation of concerns, in: Proc. 21st International Conference on Software Engineering, ICSE'99, ACM Press, 1999, pp. 107–119.
[20] W. Tracz, DSSA (Domain-Specific Software Architecture) pedagogical example, ACM Software Engineering Notes 20 (3) (1995) 49–62.
[21] XVCL homepage, 2003, Available at http://fxvcl.sourceforge.net.

[22] H. Zhang, S. Jarzabek, M.S. Soe, XVCL approach to separating concerns in product family assets, in: Proc. Third International Symposium on Generative and Component-based Software Engineering, GCSE 2001, Lecture Notes in Computer Science, vol. 2186, Springer, Berlin, 2001, pp. 36–47.

[23] H. Zhang, S. Jarzabek, B. Yang, Quality prediction and assessment for product lines, in: Proc. of the 15th International Conference on Advanced Information Systems Engineering, CAiSE'03, Lecture Notes in Computer Science, vol. 2681, Springer, Berlin, 2003, pp. 681–695.