XML-based Method and Tool for Handling Variant Requirements in Domain Models

Stan Jarzabek and Hongyu Zhang
Department of Computer Science, School of Computing,
National University of Singapore
Lower Kent Ridge Road, Singapore 117543
{stan, zhanghy}@comp.nus.edu.sg

Abstract

A domain model describes common and variant requirements for a system family. UML notations used in requirements analysis and software modeling can be extended with "variation points" to cater for variant requirements. However, UML models for a large single system are already complicated enough. With variants - UML domain models soon become too complicated to be useful. The main reasons are the explosion of possible variant combinations, complex dependencies among variants and inability to trace variants from a domain model down to the requirements for a specific system, member of a family. We believe that the above mentioned problems cannot be solved at the domain model description level alone. In the paper, we propose a novel solution based on a tool that interprets and manipulates domain models to provide analysts with customized, simple domain views. We describe a variant configuration language that allows us to instrument domain models with variation points and record variant dependencies. An interpreter of this language produces customized views of a domain model, helping analysts understand and reuse software models. We describe the concept of our approach and its simple implementation based on XML and XMI technologies.

1. Introduction

Variant requirements arise naturally during analysis of a system family (also called a Product Line). While having much in common, members of a family also differ in functional and non-functional requirements, design decisions, runtime architectures, platforms and other characteristics. The subject of domain analysis is modeling common and variant requirements across family members. Implicit variants also arise in traditional analysis of single system requirements. Goal-Oriented requirement analysis method [12] calls for explicit representation of early decisions related to functional and non-functional requirements. These decisions affect each other and without proper analysis model, it is easy to overlook the impact of various decisions on the whole system. From Goal-Oriented perspective, at the initial development stage the system requirements space is characterized by many inter-dependent variants. Modeling a web of inter-connected goals and decisions is analogous to domain modeling. Careful analysis of goals, decisions and their outcomes gradually narrows down the space of potential requirements. This process is analogous to selection of variant requirements from a domain model in order to specify requirements for a specific member of a system family. Having said that, we shall anchor our further discussion on issues arising in domain modeling for system families. Figure 1 highlights a system family approach to software development.

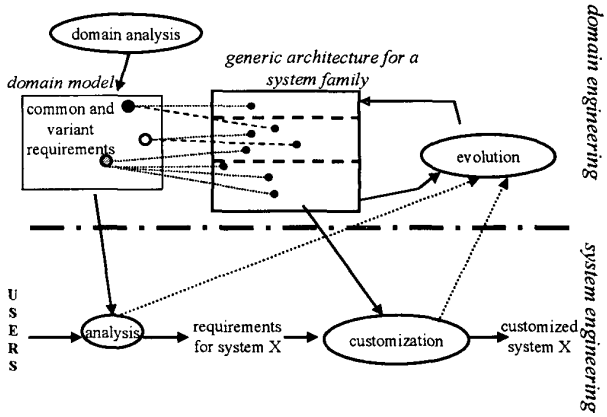

Figure 1. Domain and system engineering

Domain engineering delivers software assets that can be reused during analysis, design and implementation of family members. Major reusable assets include a domain model and generic architecture for a system family. A generic architecture defines an overall architecture for family members and provides implementation of both common and variant requirements in an adaptable form. We build systems by reusing a domain model and generic architecture. A domain model describes the scope of

1090-705X/01 $10.00 © 2001 IEEE

functionality (both common and variant requirements) that have been implemented into a generic architecture. Domain engineering artifacts evolve based on the feedback from system engineering. In practice, scoping of a system family is done before detailed domain analysis and design [7]. During scoping, we identify types of variants that are worth reusing.

Feature models [10] have been used in domain analysis to depict mandatory and variant requirements. Feature models must be complemented with other notations, for example UML, to enhance the meaning of domain concepts. Notations traditionally used in requirement analysis can be extended with the concept of a "variation point" [8] to make them useful in domain modeling. In our early work, we extended notations of DFD, ER and STD with variants [4].

Modeling variants adds an extra level of complexity to domain analysis, otherwise similar to requirement analysis for a single system. Tracing multiple occurrences of the same variant in different domain model views and understanding how mutually dependent variants affect each other is a major challenge in domain engineering. While each step in modeling variants may be simple, as the volume of information grows, domain models become notoriously difficult to understand. The main problems are the explosion of possible variant combinations, complex dependencies among variants and inability to trace variants from a domain model down to requirement specifications for a specific member of a system family. The impact of variants on domain model views becomes unclear, undermining the very purpose of domain modeling.

Current general-purpose (i.e., domain-independent) methods for domain modeling are based on descriptive methods. We believe that the above mentioned problems cannot be solved at the domain model description level alone. Therefore, we included into a domain model an active component that helps analysts in domain model interpretation and manipulation. In our approach, a *descriptive part of the domain model* includes a set of *default domain views, feature diagrams* that describe variants, and *customization scripts* that describe variants in respect to defaults. *Default domain views* describe a typical system in a domain, expressed in UML notations. Domain defaults are the starting point for understanding the scope of a system family, i.e., the range of systems in a domain we wish to consider. *Customization scripts* specify variants as deltas in respect to domain defaults. A customization script contains commands that add or delete required variants to/from domain defaults. Our solution also includes a *flexible variant configuration* tool (FVC for short) as an integral part of a domain model. FVC helps analysts understand and manipulate the domain model descriptions. FVC is an interpreter of customization scripts. Based on selected variants, FVC promptly provides analysts with customized views of a domain model (i.e., requirement specifications for a system that meets specific variants). The FVC helps analysts explore domain defaults and variants, trace dependencies among variants, etc., enhancing understanding of a domain model. Customization scripts as well as defaults can be easily modified, providing flexibility required during customization and evolution of the domain model.

Our initial approach to handling variants in a domain model was based on Bassett's frames [1]. Frame method and tools have an excellent record in industrial applications as an effective way to handle variants in reusable software. In our domain engineering projects, we have applied frames to build generic architecture for system families. We also conducted initial experiments to apply frames in domain models. Recently, we designed a variant configuration language, called XVCL [17], which implements essential frame concepts in XML. In this paper, we describe how we applied XVCL to handle variants in UML domain models represented as XMI [13] documents. We chose XMI, as modeling tools such as Rational Rose™ adopt XMI standards as a common, exchangeable representation for software models. In this solution, an FVC is an XVCL interpreter implemented on top of JAXP [15], an open framework for parsing XML documents. The reader should notice that the very concept of our approach to supporting domain modeling is not limited to XML, JAXP or XMI.

In the remaining part of the paper, we describe our solution, illustrating it with examples from our domain engineering project in the Computer Aided Dispatch (CAD) domain.

2. Related work

System family approach has emerged as one of the most promising trends to improve software productivity and quality. In Feature-Oriented Domain Analysis (FODA) [10] mandatory and variant requirements (called features) are depicted in the graphical form as trees. By traversing the feature trees, we can find out which variants have been anticipated during domain analysis. A design space, a multidimensional space of design choices, is yet another approach to describe variant requirements [3]. Unlike feature diagrams, design spaces do not show variant types (e.g., optional, alternative and or-relationship) or depict structural relationship among variant requirements.

UML notations may be extended with "variation points" to cater for variant requirements [8]. A generic software model (analysis component) is customized by attaching one or several variants to its variation points.

In analogical domain analysis [11], one attempts to build abstract models for problems that recur in different application domains. Abstract models are then instantiated for reuse by injecting domain-specific variants into them. Domain Specific Languages (DSL) and application generation techniques provide a powerful method to deal

with variants in system families [2]. A DSL allows one to specify variants in application domain terms. Variant specifications in DSL guide generation actions that produce a custom program that meets required variants. In contrast to the above approaches, the method described in this paper is domain-independent. We concentrate on a problem of how variants affect domain views expressed in commonly used notations such as UML. Unlike DSLs, our variant specification language XVCL carries no semantics of a domain. Our tool uses simple adaptation and composition rules, rather than generation techniques, to produce customized domain model views from generic model components. We described the follow up techniques for handling variants in system family architectures in other publications [5,9]. The reader may also refer to [6] for a detailed comparative study of various approaches to handling variants in system families.

3. CAD domain overview

We shall use a domain of Computer Aided Dispatch (CAD) to illustrate our approach. CAD systems are used by police, fire & rescue, health service, port operations and others. Figure 2 depicts a basic operational scenario and roles in a CAD system for Police.

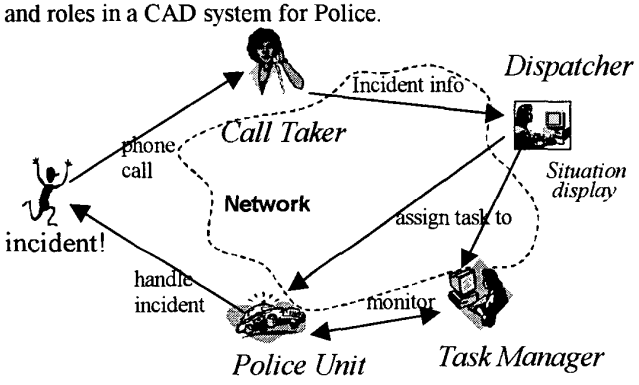

Figure 2. A basic operational scenario in CAD system for Police

A Call Taker receives information about an incident and informs a Dispatcher about the incident. The Dispatcher examines the *Situation Display* that shows a map of the area where the incident happened. Then, the Dispatcher assigns a task of handling the incident to a Police Unit, for example, this might be a police car that is closest to the place of an incident. The Police Unit approaches the place of incident and handles the problem. The Police Unit informs the Task Manager about the progress of action. The Task Manager monitors the situation and at the end - closes the task.

At the basic operational level, all CAD systems are similar - basically, they support the dispatch of units to handle incidents. However, there are also differences across CAD systems. The specific context of the operation results in many variations on the basic operational theme. Here are some of the variant requirements in CAD domain:

1. *Call Taker and Dispatcher roles* (referred to as CT-DISP variant). In some CAD systems, Call Taker and Dispatcher roles are separated (played by two different people), while in other CAD systems the Call Taker and Dispatcher roles are played by the same person. The CT-DISP variant has impact on system functionalities. For example, in the former case, the Call Taker needs to inform Dispatcher of the newly created task, but in the latter case, once the Call Taker creates a task, she/he can straightway dispatch resources (e.g., Police Units) for this new task.
2. *Validation* of 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 info) 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.

Figure 3 shows an excerpt from the CAD feature diagram [10].

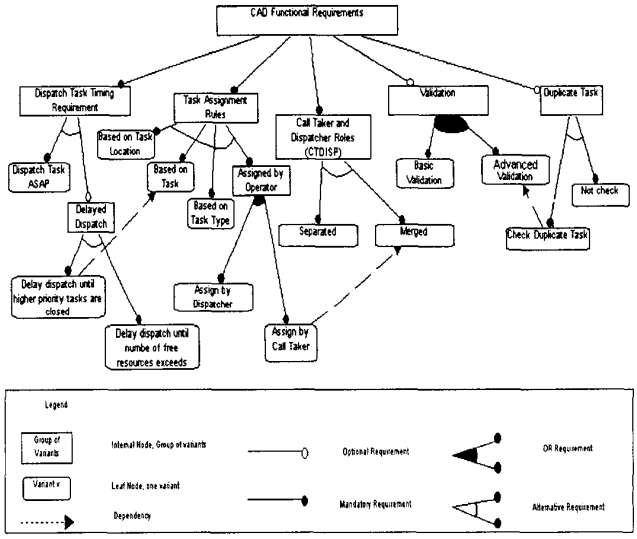

Figure 3. CAD feature model

The legend in Figure 3 explains notations (we use extensions described in [6]). Mandatory requirements appear in all the instances of a parent concept. Variant requirements only appear in some of the instances of the parent concept. Variant requirements are further qualified as optional, alternative and or-requirements. An alternative describes one-of-many requirements. For example, the "Call Taker and Dispatcher roles" requirement described above has two alternative variants: "Separated" and "Merged". An or-requirement describes any-of-many requirements. For example, 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 or neither of them.

4. XVCL: an XML-based Variant Configuration Language

To address variants in a domain model, we designed an XML-based Variant Configuration Language (XVCL for short), a simple markup language based on XML conventions [17]. We use XVCL to organize domain knowledge and to instrument domain defaults with variants. Table 1 lists some of the major XVCL commands. We use term x-frame to refer to domain defaults instrumented with variants marked as XVCL commands. An x-frame can be processed by the XVCL interpreter (i.e., the XML implementation of the FVC tool).

XVCL Command	Description
<X-FRAME name="*name*"> </X-FRAME>	Denotes an x-frame.
<DECLARATION> </DECLARATION>	Here global variables and settings are declared.
<BODY> </BODY>	Domain defaults instrumented with XVCL customization commands are defined in the body section of an x-frame.
<COPY x-frame="*x-frame*"> *customization commands* </COPY>	Includes a copy of the specified *x-frame* after applying *customization commands* inside the *x-frame*
<INSERT-BEFORE name="*breakpoint*"> </INSERT-BEFORE> <INSERT name="*breakpoint*"> </INSERT> <INSERT-AFTER name="*breakpoint*"> </INSERT-AFTER>	Allows insertions of fragments of information at the *breakpoint*. The inserted content can be placed before, after the breakpoint, or replace the existing content at the breakpoint.
<BREAK name="*breakpoint*"> </BREAK>	Specifies a *breakpoint* in an x-frame body, where customizations may occur.
<SET name="*varname*" value="*varvalue*"> </SET>	Declares an XVCL variable *varame* with value *varvalue*.
<VAR name="*varname*"/>	Denotes an XVCL variable *varname*.
<SELECT name="*variable*"> <OPTION value="*value*"> </OPTION> <OTHERWISE> </OTHERWISE> </SELECT>	Selects one of many customization options based on the *value* of a *variable*.

Table 1. A list of XVCL commands as XML tags

In the rest of the paper, we will show how we apply XVCL in domain modeling, using examples of use cases and workflows in CAD domain.

5. Modeling use cases in CAD domain

5.1. Applying UML extension mechanism

Use cases differ in many details across members of a CAD system family. We can model use case variants with <<extend>> and <<include>> stereotypes [14]. Figure 4 shows the *Create Task* use case.

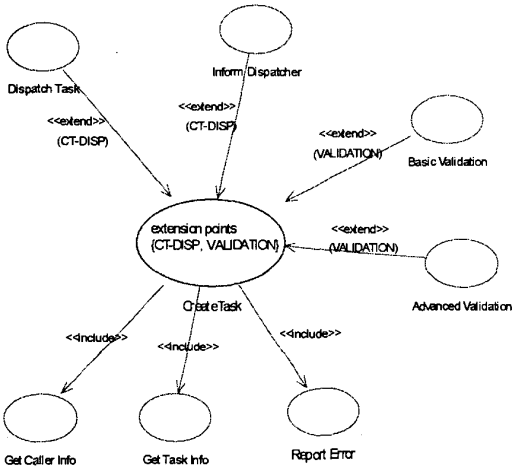

Figure 4. *Create Task* use case diagram

Create Task use case allows a Call Taker to create a task for an incident reported by an emergency caller. We have identified two variation points in *Create Task*, namely {CT-DISP} (Call Taker and Dispatcher roles) and {VALIDATION}. The <<extend>> stereotype indicates the use cases describing variant behavior associated with variation points (e.g., *Dispatch Task* and *Basic Validation*). The <<include>> stereotype indicates use cases that may be reused by other use cases (e.g., *Get Caller Info*).

A use case diagram is often complemented by description such as the one shown in Figure 5. Parameter RESPONSTIME defines required response time.

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
Call Taker login

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)
Call Taker logout
3. Special Requirement
Call Taker should respond to the emergency call within "**RESPONSETIME**".

Figure 5. *CreateTask* use case description

The *CreateTask* use case described above is rather small – it only includes two variation points. As the volume of information grows, and more variants and variant dependencies are identified, the models get complicated and become difficult to understand. For example, if two more variation points within *CreateTask* use case are identified, assuming each variation point has two possible values, there will be four more extension use cases. This will bring the total number of variant combinations up to 2^4. The exponential explosion of possible variant combination makes the manual customization (specialization) of use case model difficult. In addition, the impact of variants is not limited to use cases but spreads over other domain model views.

5.2. Applying XVCL

XVCL and its interpreter help us alleviate the above mentioned problems. Based on selected variants and x-framed domain defaults, XVCL interpreter produces the customized use case model for a specific system. System analysts need only understand the customized models she/he is interested at a given moment without having to examine the entire domain model space.

Figure 6 shows the *Create Task* use case description instrumented for flexibility with XVCL commands. The <X-FRAME> tag denotes the x-frame for *Create Task* use case description. In <DECLARATION> section, the XVCL variable RESPONSTIME is defined with default value of "30 secs". The contents of the x-frame is encapsulated in the <body> section. The <COPY> command indicates the <<include>> relationship. When XVCL interpreter encounters the <COPY> 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 point where additional customizations that cater for unexpected variants may occur. In this example, the <BREAK> command indicates the variation point brought up by the optional 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.

```
<X-FRAME name="CreateTask_description.uc">
<DECLARATION>
  <SET name="RESPONSETIME" value="30 secs"/>
</DECLARATION>
<BODY>
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
  Call Taker login
  <COPY x-frame="Get_Task_Info.uc" />
  <COPY x-frame="Get_Calle_Info.uc"/>
  <BREAK name="VALIDATION"/>
  If validation failed then <copy x-frame=
  "Report_Error.uc"/> and abort the session.
  Create a new task
  <SELECT name="CT-DISP"/>
  <OPTION value="SEPARATED">
    <COPY x-frame="Inform_Dispatcher.uc"/>
  </OPTION>
  <OPTION value="MERGED">
    <COPY x-frame="Dispatch_Task.uc"/>
  </OPTION>
  Call Taker logout
3. Special Requirement
Call Taker should respond to the emergency call within
<var name="RESPONSETIME"/>.
</BODY>
</X-FRAME>
```

Figure 6. The x-frame for *Create Task* use case description

We wish to use the same approach that we applied to use case description to instrument use case diagrams for flexibility, as well. To achieve this, we convert UML use case diagrams into equivalent textual representation and instrument the text with XVCL commands for flexibility. We can then perform the same kind of adaptation on textual use case diagram as we have done on use case description. After customizations, we convert the text back to diagrams.

To illustrate this technique, we use an XMI (XML Metadata Interchange) tool Unisys Rose/XMI to convert the UML diagrams to equivalent textual representation in XML. XMI [13] is a new OMG standard that combines UML and XML and enables the exchange of UML models over the Internet. XMI supports the round-trip transformation of UML models from diagrams to an XML file without loss of information.

Figure 7 shows an x-frame for Create *Task* use case diagram depicted in Figure 4. To accommodate the "CT-DISP" variant, we instrument the use case with <SELECT> command, which indicates the places where

the anticipated customizations will occur. The <BREAK> command is used to indicate possible customization that may occur due to the VALIDATION variant.

```
<X-FRAME name="CreateTask_diagram.uc">
<BODY>

<XMI xmi.version = '1.0'>
 <XMI.header>                    // XMI Header Info
  <XMI.metamodel xmi.name = 'UML' xmi.version = '1.1'/>
 </XMI.header>

 <XMI.content>                   //XMI Content
      ...
<SELECT name="CT-DISP">
<OPTION value="SEPARATED">
  // Definition of "Inform Dispatcher" Use Case Element
<Behavioral_Elements.Use_Cases.UseCase xmi.id="UC_INFODISP ">
   <Foundation.Core.ModelElement.name>Inform Dispatcher
   </Foundation.Core.ModelElement.name>
   <Behavioral_Elements.Use_Cases.UseCase.extensionPoint />
 </Behavioral_Elements.Use_Cases.UseCase>
      ...
</OPTION>
<OPTION value="MERGED">
  // Definition of "Dispatch Task" Use Case Element
<Behavioral_Elements.Use_Cases.UseCase xmi.id="UC_DISPTASK ">
   <Foundation.Core.ModelElement.name>Dispatch Task
   </Foundation.Core.ModelElement.name>
   <Behavioral_Elements.Use_Cases.UseCase.extensionPoint />
 </Behavioral_Elements.Use_Cases.UseCase>
      ...
</OPTION>
</SELECT>

<BREAK name="VALIDATION"/>
      ...
</XMI.content>
</XMI>
</BODY>
</X-FRAME>
```

Figure 7. x-frame for *Create Task* use case diagram

A *customization script* specifies how to adapt an x-frame to accommodate variants. During customization, the XVCL interpreter reads an x-frame and customizes it according to the instructions. Figure 8 shows a customization script that adapts the generic CreateTask x-frame to accommodate the "separated Call Taker and Dispatcher roles" and the "basic validation" variants.

```
<customization>
  <SET name="CT-DISP" value="SEPARATED"/>
  <COPY x-frame="Create_Task_description.uc" />
    <INSERT name="VALIDATION">
       Perform basic validation checking
    </INSERT>
  </COPY>
  <COPY x-frame="Create_Task_diagram_.uc" />
    <INSERT name="VALIDATION">
      // Definition of "Basic Validation" Use Case Element
      <Behavioral_Elements.Use_Cases.UseCase
       xmi.id="UC_VALIDATION ">
      <Foundation.Core.ModelElement.name>Basic Validation
      </Foundation.Core.ModelElement.name>
      <Behavioral_Elements.Use_Cases.UseCase.extensionPoint />
      ...
      </Behavioral_Elements.Use_Cases.UseCase>
    </INSERT>
  </COPY>
</customization>
```

Figure 8. A customization script

Figure 9 shows the customized *Create Task* use case description. Customized textual use case diagram in XML can be converted back to UML use case diagram by using the Unisys Rose/XMI tool. Figure 10 shows the customized *Create Task* use case diagram in UML.

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
   Call Taker login
   (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)
   Call Taker logout
3. Special Requirement
Call Taker should respond to the call within "30 secs".

Figure 9. A customized *Create Task* use case

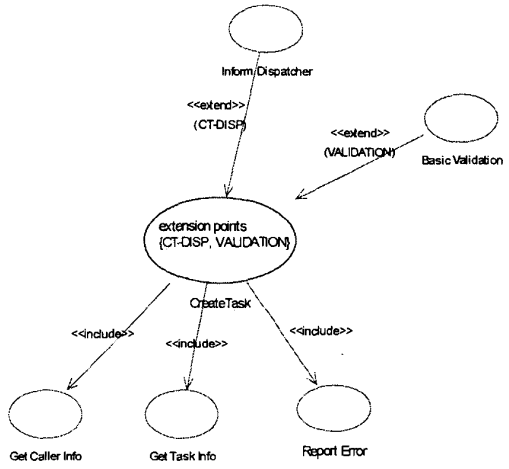

Figure 10. A customized *Create Task* use case

6. Flexible workflows

To handle variants in workflows, we extended activity diagram with variation points to model alternative and optional flows of activities. Figure 11 depicts the *Create*

Task workflow with CT-DISP and VALIDATION variants.

The shaded diamond is a meta-symbol that marks the variation point. It denotes a decision related to Call Taker and Dispatcher roles. In the diagram, the shaded diamond is stereotyped to become an <<alternative>> decision, whereby one of the paths from the decision point will be included in the customized diagram.

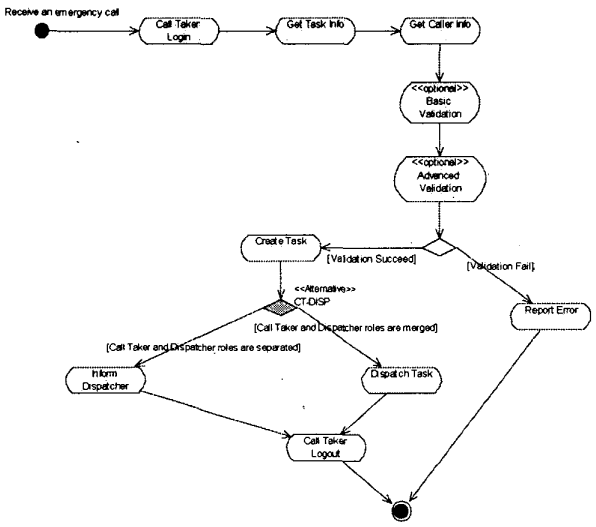

Figure 11. *Create Task* workflow with variation points

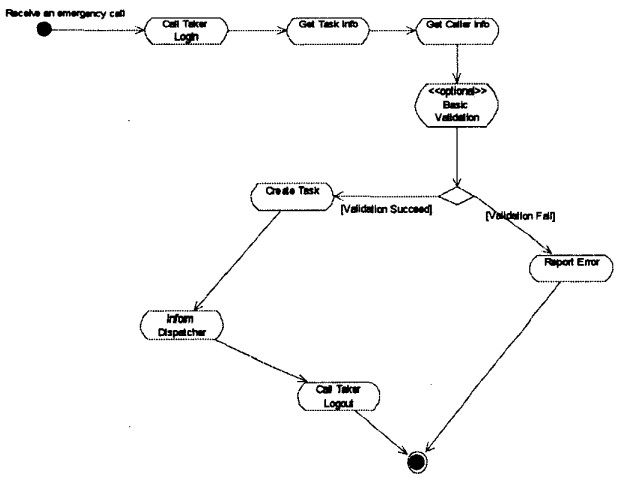

Figure 12. A customized *Create Task* workflow

We use the same approach to instrument workflow diagrams for flexibility as we applied to use cases diagrams. With an XMI tool, we convert workflow diagrams into equivalent XML representation. Then, we create an x-frame by instrumenting the XML document with XVCL commands. We can then perform the same kind of customizations as we did for use cases. Finally, we convert the customized XML document back to UML activity diagrams.

The x-frame for *Create Task* workflow contains <SELECT> and <BREAK> commands to accommodate the CT-DISP and VALIDATION variants, respectively. Both the workflow x-frame and customization script are similar to x-frames and customization scripts for use cases and we do not show them here to conserve the space. Figure 12 depicts the customized *Create Task* workflow.

7. Analysis of Results

In sections above, we described the basic technique to handle variants in domain models. Our technique is easy to grasp when we deal with small number of defaults and variants. But in reality, analysts deal with large spaces of requirements, involving many defaults and variants. By now, hopefully we managed to get the reader interested in our approach, but no doubt the reader will remain unconvinced unless we show that the method has a potential to scale up. In this section, we shall explain how we organize large volumes of domain information, describe the scope of our experimentation, identify weaknesses of our solution so far and discuss future work.

We organize domain knowledge around the feature diagram such as the one shown in Figure 3. We extended feature diagrams to provide explicit mappings between variants and customization scripts. This new structure is a form of a decision model [16] that we call a Customization Decision Tree (CDT). In our previous domain engineering projects [5], we introduced a CDT to aid in understanding a generic architecture for a system family. Here, we apply a similar idea to a domain model. By inspecting a customization script for a given variant, we can find out which domain defaults must be customized to meet the variant and how to do customization. Having selected variants, we synthesize relevant scripts to produce a consolidated customization script. When synthesizing inter-dependent variants, analyst must modify relevant customization scripts by hand. To minimize the extent of manual work, we pre-define consolidated customization scripts for typical combinations of inter-dependent variants. The XVCL interpreter selects relevant x-frames, applies customizations described in the consolidated customization script and outputs customized domain model views such as use cases or workflows.

We applied the above domain modeling technique in two projects involving Facility Reservation (FR) and Computer Aided Dispatch (CAD) system domains. We experimented thoroughly only with use case and workflow notations. In FR domain, our experimentation covered all the major functions and over 50 variants

displaying a range of mutual dependencies. In CAD domain, we modeled functions related to Call Taker and Dispatcher roles with 20 variants.

We believe our approach has a potential to reduce the complexity of a domain model and offers tangible advantages over purely descriptive domain modeling methods. At the same time, our approach and the scope of experimentation have the following limitations that we shall address in the future work:

Experiment on a larger scale

We have only experimented with selected views of a domain model and on a medium-size scale. We plan to cover a wider spectrum of modeling notations and experiments on a larger scale in future.

Address complex requirement dependencies

So far, we have been dealing with relatively simple class of functional variant dependencies. We yet have to extend research to non-functional variants. Also, other domains may give rise to different types of dependencies (such as time-based dependencies) that will require specialized approach.

Extend and integrate the proposed method with the customization of the generic architecture

The proposed solution offers the opportunity to use one technique throughout the domain engineering and the system engineering phases. We can extend and integrate the method with the customization of the generic architecture and the program code to cover the whole software development life cycle. We conducted initial experiments in the CAD domain and obtained encouraging results.

8. Conclusions

As the volume of information grows, domain models become notoriously difficult to understand. The main problems are in explosion of possible variant combinations, complex dependencies among variants and inability to trace variants from a domain model down to requirement specifications for a specific member of a system family. In this paper, we proposed a new approach to handle variants in a domain model. The main idea of our solution is to define default domain model views and provide a semi-automatic way to modify and extend these defaults. We proposed a *flexible variant configuration* component (FVC for short) as an integral part of a domain model. The role of FVC is to help analysts in interpretation and manipulation of domain variants. In the paper, we described a simple implementation of the above concepts using XML and XMI technologies.

Acknowledgments

The XVCL and XML-based tool for model customization were inspired by Bassett's frames. Thanks are due to NUS students Soe Myat Swe who implemented frames in XML and Ong Wai Chung who applied frames to facility reservation system models. This work was supported by project NSTB/172/4/5-9V1 funded under the Singapore-Ontario Joint Research Programme and by NUS Research Grant R-252-000-066.

References

[1] Bassett, P. *Framing Software Reuse - Lessons from Real World*, Yourdon Press, Prentice Hall, 1997

[2] Batory, D., Chen, G., Robertson, E. and Wang, T. "Design Wizards and Visual Programming Environments for GenVoca Generators," *IEEE Trans. on Software Engineering*, Vol. 26, No. 5, May 2000, pp. 441-452

[3] Baum, L., Becker, M., Geyer, L. and Molter, G. "Mapping Requirements to Reusable Components using Design Spaces", *Proc. Int'l Conf. on Requirements Engineering*, Schaumburg, Illinois, USA, June 2000

[4] Cheong, Y.C. and Jarzabek, S. "Modeling Variant User Requirements in Domain Engineering for Reuse", *Information Modeling and Knowledge Bases*, Eds. Hannu Jaakkola, Hannu Kangassalo and Eiji Kawaguchi, IOS Press, Netherlands, 1998

[5] Cheong, Y.C. and Jarzabek, S. "Frame-based Method for Customizing Generic Software Architectures", *Proc. Symposium on Software Reusability*, Los Angeles, May 1999

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

[7] Debaud, J.M. and Schmid, K. "A Systematic Approach to Derive the Scope of Software Product Lines", *Int. Conf. on Software Engineering*, Los Angeles, May 1999

[8] Jacobson, I. M. Griss and P. Jonsson *Software Reuse Architecture, Process and Organization for Business Success*, Addison-Wesley, 1997

[9] Jarzabek, S. and R. Seviora "Engineering components for ease of customization and evolution," *IEE Proceedings - Software*, Vol. 147, No. 6, December 2000, pp. 237-248, a special issue on Component-based Software Engineering

[10] Kang, K et al. "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Nov. 1990

[11] Maiden, N. and Sutcliffe "Exploiting Reusable Specifications through Analogy," *CACM*, Vo. 34, No. 4, April 1992, pp. 55-64

[12] Mylopoulos, J., Chung, L. and Yu "From Object-Oriented to Goal-Oriented Requirements Analysis", *CACM*, January 1999/Vol. 42, No. 1, pp.31-37

[13] OMG, *XML Metadata Interchange (XMI) 1.1 RTF*, OMG Document ad/99-10-02, 25 October 1999

[14] Rumbaugh, J., Jacobson, I. and Booch, G. *The Unified Modeling Language, Reference Malnual*, Addison-Wesley, 1999

[15] Sun Microsystems, Inc. *Java API for XML Processing v1.1*, December 15, 2000

[16] Weiss, D.and Lai, R. Software Product Line Engineering, Addison-Wesley Longman, 1999

[17] Wong, T.W., Jarzabek, S., Soe, M.S., Shen, R. and Zhang, H.Y. "XML Implementation of Frame Processor," *Symposium on Software Reusability, SSR'01*, Toronto, Canada, May 2001, pp. 164-172