

# x-Frame Approach for Handling Variants within Concerns

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

## ABSTRACT

In this paper, we describe an XML-based language, called XVCL, for managing variants in product families. Using XVCL, we can organize product family assets and instrument them to accommodate variants. A tool that interprets XVCL and provides semi-automatic support for asset customization is also introduced. In our projects, we applied XVCL to manage variants in UML domain models and in generic architectures for product families. We have achieved simple forms of separation of concerns (in both models and architectures) and we are investigating advanced forms in current work. During the workshop, we wish to discuss the XVCL approach and compare it to other emerging techniques that lead to separating of concerns in software models, documents, architectures and code.

## 1. INTRODUCTION

Advanced separation of concerns is now emerging as a new area in software engineering field. Advanced separation of concerns suggests that concerns in different dimensions are useful for different reasons for different stakeholder, thus it's necessary for developers to be able to identify, encapsulate and integrate any kinds or dimensions of concerns simultaneously.

Much work described in the literature focuses on the separation and composition of the concerns in the class (object), function, feature, artifact and aspect dimensions. In this paper, we describe an XML-based language and tool to manage variants within concerns.

Variants are raised from the needs of product family [PARN76]. A product family (or product line) is a set of products that share a common set of requirements, but also differ in certain features. The variants within concerns are imposed, as we have to handle the variabilities among different product family members.

Like concerns, variants may cut across - the impact of variants may scatter across many modules, and may be also tangled with the impact of other variants. As the number of variants increases, the explosion of possible variant combinations and complex inter-dependencies among variants may further complicate the maintenance and evolution of the systems.

We designed an XML-based Variant Configuration Language (XVCL) [WONG01], to handle variants in a product family. With XVCL, we can organize and manage a wide range of software assets such as domain models (e.g., in UML), software documents, product family architectures, code and test cases. We use the term x-frame to refer to the software asset in a product family instrumented for flexibility and reuse with XVCL commands. The XVCL processor is a simple yet powerful tool that can customize and compose the x-frames, to produce a

specific system, member of a product family. Our XVCL is inspired by the frame technology from Netron Inc. [BASS97]. The XVCL processor is an XML implementation of the industry-proven Frame Processor.

In our approach, concerns are encapsulated in groups of x-frames. An x-frame is instrumented with XVCL commands that show how to incorporate variants within concerns. Given specific variants, custom components/systems can be generated from the x-frame hierarchy on demand. Although the XVCL was originally designed for addressing variants within concerns, the basic concept can be extended to compose multiple concerns as well.

In our projects, we applied XVCL to manage variants in UML domain models and in generic architectures for product families. We have achieved simple forms of separation of concerns and we are investigating advanced forms in current work.

In the remaining part of the paper, we will illustrate our approach using examples from our domain engineering project on Computer Aided Dispatch (CAD) domain.

## 2. RELATED WORK

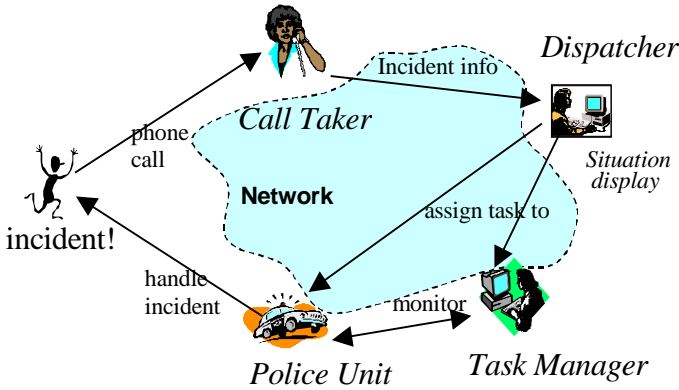
As early as in 1970's, Parnas [PARN72,76] proposed modularization, information hiding and separation of concerns principles for handling variants in a product family. Marco processors, PCL [SOMM96], application generators [BATO98], Frame Technology [BASS97], Object-Oriented framework [JOHN88], template and meta-programming techniques [CZAR00] - they all offer mechanisms to handle variants in product family.

Recent work focuses on advanced separation of concerns. A number of approaches have been proposed to address crosscutting concerns and concern compositions. In aspect-oriented programming [KICZ97], each computational aspect is programmed separately and rules are defined for waving aspects with the base code. In multi-dimensional separation of concerns and hyperspace approach [TARR99], hyperslices encapsulate concerns in dimensions other than the dominant one and can be composed to form the complete system.

Our approach, in general, falls into the category of "generative programming" [CZAR00]. In our approach, concerns are encapsulated in groups of x-frames. A specific system, member of a product family, can be generated from the x-frame hierarchy on demand. Unlike AOP, we explicitly mark the points where code (or other reusable contents) related to variants (or aspects) can be inserted.

### 3. THE CAD DOMAIN OVERVIEW

We shall use the domain of Computer Aided Dispatch (CAD) to illustrate our approach. CAD systems are mission-critical systems that are used by police, fire & rescue, health service, port operations and others. Figure 1 depicts a basic operational scenario and roles of a CAD system for Police.



**Figure 1:** A basic operational scenario in CAD system for Police

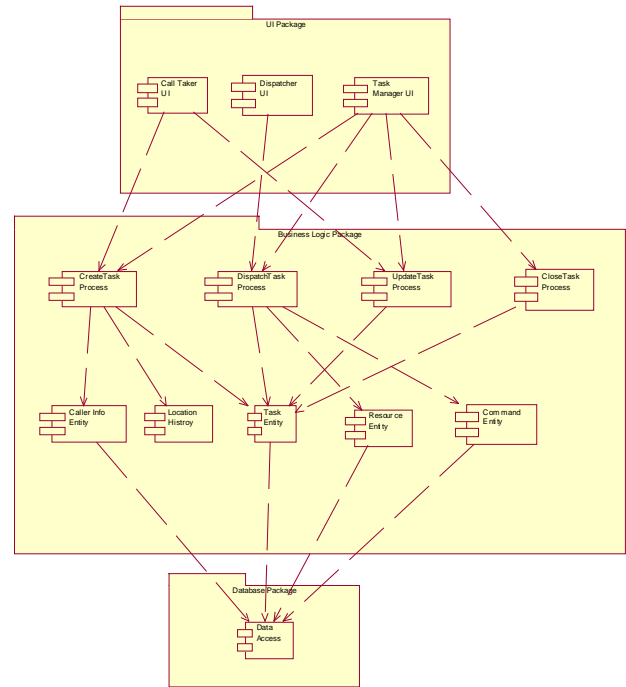
Once a Caller reports an incident, the Call Taker captures the details of the incident and the Caller, and creates a task for this incident. The Dispatcher then selects suitable Resources (e.g. Police Units) and dispatches them to execute the task. The Resources carries out the task instructions and reports to the Task Manager. The Task Manager actively monitors the situation and at the end - closes the task.

#### 3.1 The Initial CAD System

We adopt an use-case driven, architecture centric approach to develop the initial (default) CAD system. The problem and solution space are decomposed along the class boundaries, which are identified from the use case analysis and design. A three-tiered architecture style guides the realization of the use cases.

Figure 2 gives a component view of the initial CAD system. Some of the concerns are separated and localized. User Interface concerns are encapsulated in the UI classes (e.g., the CallTakerUI class contains the code related to *Call Taker UI*). Concerns related to business process (workflow) are encapsulated in the control classes (e.g., the CreateTaskProcess class localized the implementation of the *Create Task* business process). Concerns related to entities are encapsulated in entity classes (e.g., Task class contains the implementation that provides the *Task data* service). Data Access class only provides the data access service.

The initial CAD system is modeled in UML, implemented in Java, and deployed it on a CORBA-compliant component platform.



**Figure 2:** The component view of the initial CAD system

#### 3.2 Variants in CAD Product Family

At the basic operational level, all CAD systems are similar - basically, they support the dispatch of units to handle incidents. However, considered the product family situation, there are also differences across CAD systems. The specific context of the operation (such as police or fire & rescue) results in many variations on the basic operational scheme. Here are some of the variant requirements in the 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 merged (played by one person). This "Call Taker and Dispatcher roles" 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 Unit) 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.

Feature diagrams [KANG90] are often used to represent the common and variant requirements in a domain. Figure 3 shows an excerpt from the CAD feature diagram. We use extensions described in [CZAR00]. The legend in Figure 3 explains notations. Mandatory requirements appear in all the instances of a

parent concept. Variant requirements only appear in some of the instances of the parent concept being described. 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.

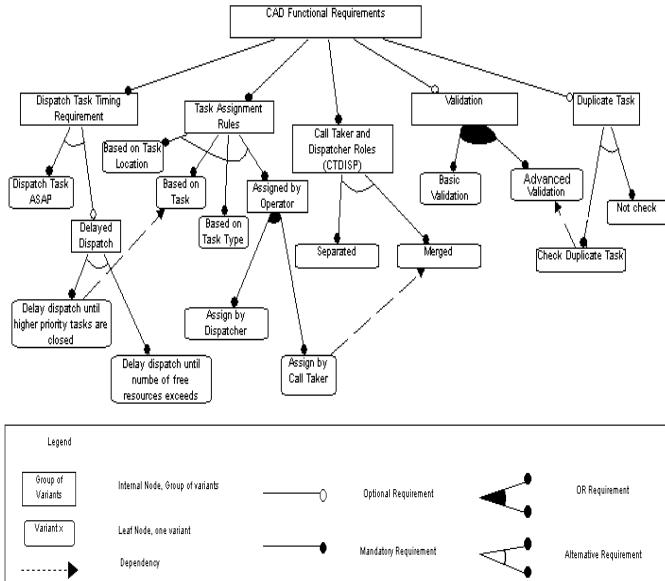


Figure 3. CAD feature model

#### 4. XVCL: AN XML-BASED VARIANT CONFIGURATION LANGUAGE

XVCL [WONG01] is a simple markup language based on XML conventions. We use XVCL to organize domain assets and to instrument domain defaults with variants. Table 1 lists some of the major XVCL commands. We use the term *x-frame* to refer to domain defaults instrumented with variants marked as XVCL commands. An *x-frame* can be processed by the XVCL processor.

XVCL Command	Description
<X-FRAME name="name"> </X-FRAME>	Denotes an x-frame.
<X-PACKAGE name="name"> </X-PACKAGE>	Denotes a group of related x-frames.
<COPY x-frame="x-frame"> <i>customization commands</i> </COPY>	Includes a copy of the specified <i>x-frame</i> after applying <i>customization commands</i> to the <i>x-frame</i>
<INSERT-BEFORE name="breakpoint"> </INSERT-BEFORE> <INSERT name="breakpoint"> </INSERT> <INSERT-AFTER name="	Allows insertions of fragments of information at the <i>breakpoint</i> . The inserted content can be placed before, after the <i>breakpoint</i> , or replace the

"breakpoint"> <INSERT-AFTER>	existing content at the <i>breakpoint</i> .
<BREAK name="breakpoint"> </BREAK>	Specifies a <i>breakpoint</i> in an x-frame body, where customizations may occur.
<SET name="varname" value="varvalue"> </SET>	Declares an XVCL variable <i>varname</i> with value <i>varvalue</i> .
<VAR name="varname"/>	Denotes an XVCL variable <i>varname</i> .
<SELECT option="variable"> <OPTION value="value"> </OPTION> <OTHERWISE> </OTHERWISE> </SELECT>	Selects one of many customization options based on the <i>value</i> of the <i>variable</i> .

Table 1. A list of XVCL commands as XML tags

Our XVCL is inspired by the frame technology from Netron Inc. [BASS97]. Frame method and tool have an excellent record in industrial applications. A quantitative study has shown that Frame technology can lead to reduction in time-to-market (70%) and project costs (84%) [BASS97]. Frame technology is tightly coupled with COBOL. Our XVCL supports all major frame commands and provides extensions for distributed component-based systems written in Java. Being based on XML, XVCL is an open, extendable and easy to use language.

XVCL helps us organize product family assets and instrument them for flexibility, to achieve systematic and effective reuse. We applied XVCL to manage variants in software assets such as UML domain models and product family architecture. In domain models, an *x-frame* contains fragments of use case, activity diagram or object collaboration models. In product family architecture, an *x-frame* may contain a component (or part of it such as method, class, declarations of data structures) or a connector (defined, for example, in IDL).

We organize *x-frames* into a hierarchy that shows how to build complex *x-frames* out of simpler ones. We distinguish basic *x-frame* from composite *x-frame*. A basic *x-frame* is treated as atomic, while a composite *x-frame* is the composition of many basic *x-frames* and other composite *x-frames*. For example, a basic *x-frame* may contain a cohesive method, a view of UML model or a test case. A composite *x-frame* may contain a class, a UML model or a test plan.

Like in aspect-oriented programming [KICZ97], we also attempt to isolate different computational concerns (both functional and non-functional) into separate *x-frames* to achieve separation of concerns. For example, data *x-frames* only contain the implementation of the entity concerns. Workflow (Business Process) *x-frames* only contain the code related to flexible workflow concerns. Logging *x-frames* only contain the code related to the Logging concern. Different views of a domain model are also localized in separate UML *x-frames*. The composition of *x-frames* also achieves composition of multiple concerns.

In the rest of the paper, we will show how we apply XVCL in handling variant with concerns, using examples from the CAD domain engineering project.

## 5. APPLYING XVCL IN HANDLING VARIANTS IN CAD DOMAIN

### 5.1 Handling Variants within Concerns

To handle variants in CAD product family, we instrument each class in the initial CAD system (as shown in Figure 2) with XVCL commands. Figure 4 shows the x-frame for the *CreateTaskProcess* class in Figure 2. The *CreateTaskProcess* x-frame only pertains to the concern of *Create Task* business process.

```
<x-frame name="CreateTaskProcess">
  <set name="ERRMSG" value="Error in Creating the Task!"/>

  <copy package="Common" x-frame = "Header"/>
  package BusinessLogic;

  public class CreateTaskProcess {
    private Task aTask;
    private String szErrMsg = <var name="ERRMSG"/>;
    ...
    public CreateTaskProcess() {
      ... // Class Initialization code
      return;
    }
  }
  <copy package = "CreateTaskFrames" x-frame =
  "GetCallerInfo"/>

  <copy package = "CreateTaskFrames" x-frame =
  "GetTaskInfo"/>

  <break name ="Validation"/>

  <copy package = "CreateTaskFrames" x-frame =
  "SaveTask"/>
</x-frame>
```

```
<break name="CT-DISP">
  <copy package = "CreateTaskFrames" x-frame =
  "InformDispatcher"/>
</break>
}
```

Figure 4. The *CreateTaskProcess* x-frame

```
<x-frame name="GetCallerInfo">
  public int GetCallerInfo(String szPhoneNumber) {
    // Create a Caller Object
    Caller aCaller = new Caller(szPhoneNumber);
    // Prepare the Caller Info
    String sCallerInfo = aCaller.name + aCaller.ID + aCaller.address;

    return sCallerInfo;
  }
</x-frame>
```

Figure 5. The *GetCallerInfo* x-frame

The *CreateTaskProcess* x-frame is a composite x-frame, which is composed by many basic x-frames, such as *Header* (contains reusable program header for program description, version info, copyright, etc.), *GetCallerInfo* (for the *Get Caller Info* function, as shown in Figure 5), *SaveTask* (for the generic *Save Task* function), etc. The composition of x-frames is achieved by the `<copy>` commands, which indicate the composition points. When the XVCL processor encounters the `<copy>` command, it will import the specified x-frame from the corresponding x-package, customize it and include it into the composite x-frame. In this example, all x-frames that are related to *Create Task* are stored in the *CreateTaskFrames* x-package. x-frames that can be reused by all other x-frames are stored in the *Common* x-package. Figure 6 illustrates the composition of the *CreateTaskProcess* x-frame.

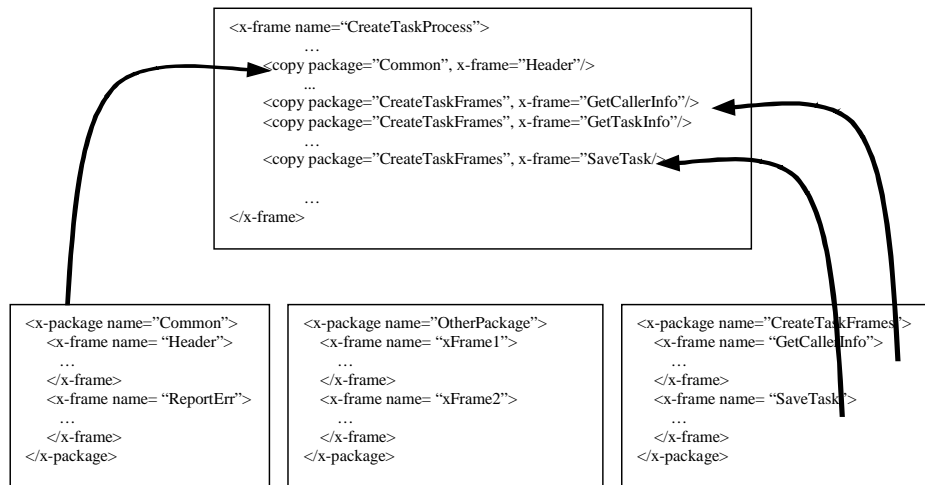


Figure 6. The composition of the *CreateTaskProcess* x-frame

A `<break>` command indicates the variation point where additional customization may occur to cater for unexpected variants. In Figure 4, the `<break name="VALIDATION">` command indicates the variation point brought up by the variant requirement *VALIDATION*. The `<break name="CT-DISP">`

command indicates the variation point brought up by the variant requirement *CT-DISP*.

XVCL variables, such as "ERRORMSG", provide another means to inject variability. XVCL variables are defined with default values,

which can be modified during program customization to fit the specific requirement.

## 5.2 Specification Frame (SPC)

In our approach, the impact of the variants is encapsulated in a special kind of x-frames called *Specification Frames* (SPC). A SPC localizes the changes required for one variant. Figure 7 shows the x-frame for the VALIDATION variant.

```
<SPC name="VARIANT_VALIDATION">
  <copy x-frame="CreateTaskProcess"
    package="BusinessLogic">
    <insert name="VALIDATION">
      public boolean Validation(Task aTask) {
        ... // Code about BasicValidation
        return ValidationResult;
      }
    </insert>
  </copy>
</SPC>
```

Figure 7. A SPC for the VALIDATION variant

The <copy> command indicates the x-frame that this variant has impact on. In Figure 7, the VALIDATION affects the *CreateTaskProcess* x-frame.

The code specified in the <insert> section can be <insert>ed into/after/before the breakpoint defined in the x-frames specified by the <copy> command. For example, in Figure 7, the VALIDATION variant is specified as “Basic Validation”. Code that meets the “Basic Validation” requirement can be <insert>ed into the “VALIDATION” breakpoint defined in the *CreatTaskProcess* x-frame during program customization.

Figure 8 gives an example of the x-frame that encapsulates the impact of the CT-DISP variant. The CT-DISP variant has impact on two x-frames: the *CreateTaskProcess* x-frame and the *CallTakerUI* x-frame. Code or x-frames that meets the requirement of “Merged Call Taker and Dispatcher Roles” can be <insert>ed into the breakpoints defined in these x-frames.

```
<SPC name="VARIANT_CTDISP">
  <copy x-frame="CreateTaskProcess"
    package="BusinessLogic">
    <insert name="CT-DISP">
      <copy package = "CreateTaskFrames" x-frame =
        "DispatchTask"/>
    </insert>
  </copy>

  <copy x-frame="CallTakerUI"
    package="UI" >
    <insert name="CT-DISP">
      public boolean AddDispatchButton(TaskInfo taskinfo) {
        ... // Code about adding a button for dispatching task
        return;
      }
    </insert>
  </copy>
</SPC>
```

Figure 8. A SPC for the CT-DISP variant

## 5.3 Composition

Once x-frames for classes and variants are separately identified and encapsulated, the XVCL processor can help us automate the

composition process (including both customization and assembly).

XVCL processor is a tool that can interpret the XVCL. During program composition, the XVCL processor reads x-frames specified in the <copy> commands, customizes them according to the instructions, and generates the executable source code. Figure 9 illustrates the composition process.

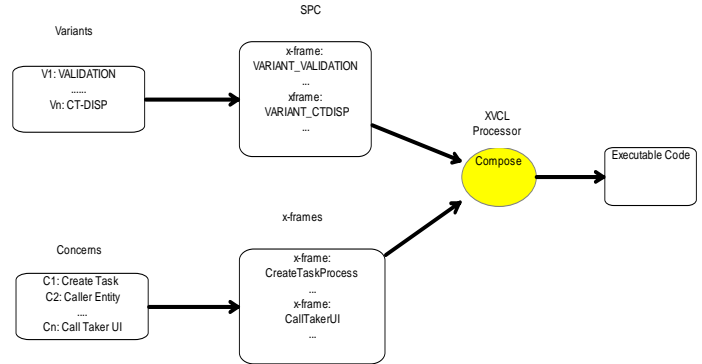


Figure 9. The composition of the SPCs and the x-frames

Figure 10 shows the generated *CreateTaskProcess* class, which meets the specific requirements (“Basic Validation” and “Merged Call Taker & Dispatcher Roles”) of a specific system, a member of CAD product family.

```
/*
 * Title: CreateTaskProcess.Java
 * Description: This is the control class for creating a task business process
 * Version: v1.0
 */
package BusinessLogic;
public class CreateTaskProcess {
    private Task aTask;
    private String szErrMsg = "Error in Creating the Task!";
    ...
    public CreateTaskProcess() {
        ... // Class Initialization code
        return;
    }

    public int GetCallerInfo(String szPhoneNumber) {
        // Create a Caller Object
        Caller aCaller = new Caller(szPhoneNumber);
        // Prepare the Caller Info
        String sCallerInfo = aCaller.name + aCaller.ID +
            aCaller.address;

        return sCallerInfo;
    }

    public int GetTaskInfo(int nTaskID) {
        ... // Code for getting task info

        return 0;
    }

    public boolean Validation(Task aTask) {
        ... // Code about BasicValidation
        return ValidationResult;
    }

    public int SaveTask(Task aTask) {
```

```

        ... // Code about saving a task
        return 0;
    }

    public int DispatchTask(Task aTask ) {
        ... // Code about dispatching a task
        return 0;
    }
}
</x-frame>

```

**Figure 10.** The generated *CreateTaskProcess* class

## 5.4 Handling variants in CAD domain model

We construct the CAD domain model by using UML [RUMB99] and its extension mechanisms. Each UML diagram represents one view of a domain model. To handle variants in the domain model, we first convert UML diagrams into equivalent textual representation. Then we instrument the XML document with XVCL commands to accommodate variants within concerns. We can then perform the same kind of composition on textual UML diagram as we have done on the Java code. Finally, we convert the generated text back to the UML diagrams.

To achieve this, we use an XMI (XML Metadata Interchange) tool Unisys Rose/XMI to convert the UML diagrams to equivalent textual representation in XML. XMI [OMG99] is a new OMG standard that combines UML and XML. XMI supports round-trip transformation of UML models from a tool (e.g. Rational Rose) to an XML file and back without loss of information.

To conserve the space, here we do not show the example of x-frames for handling variants in CAD domain model. We refer the reader to [JARZ01] for more details.

## 5.5 Variant Dependencies

Variants may be dependent on each other. For example, in the CAD domain, the “*Check Duplicate Task*” variant in Figure 3 is dependent on the “*Advanced Validation*”. If the “*Advanced Validation*” requirement doesn’t exist, it’s not necessary to consider the “*Check Duplicate Task*” variant at all. We modeled some of the variant dependencies as dashed arrows in Figure 3.

As the volume of information in a domain model grows, the number of possible variant combinations explodes. Suppose we have  $m$  variants in requirement  $r_1$  and  $n$  variants in requirement  $r_2$ . The total number of possible variant combinations is  $m \times n$ . However, the total number of combinations is less if requirement  $r_1$  and requirement  $r_2$  are dependent.

Variant dependencies reduce the number of possible variant combinations. It is important to identify the variant dependencies so that the logical complexity of the system can be reduced. In our project, we only scratched the surface of this difficult problem. We are still experimenting with various types of variant dependencies.

## 6. CONCLUSIONS

We described an XML-based language, called XVCL, and a tool to manage variants in product family assets. With XVCL, we can organize product family assets (such as domain models, documents, test cases, product family architecture and its implementation), and instrument them to accommodate variants. In our projects, we applied XVCL to manage variants in the UML domain models and in the generic architecture for CAD product

family. We have achieved simple forms of separation of concerns and our goal is to address advanced separation of concerns.

## REFERENCE

- [BASS97] Bassett, P. *Framing Software Reuse - Lessons from Real World*, Yourdon Press, Prentice Hall, 1997
- [BATO98] Batory, D., Lofaso, B. and Smaragdakis, Y. JST: Tools for Implementing Domain-Specific Languages. *Proc. 5<sup>th</sup> Int. Conf. on Software Reuse*, Victoria, BC, Canada, 1998
- [CZAR00] Czarnecki, K. and Eisenecker, U. *Generative Programming: Methods, Tools, and Applications.*, Addison-Wesley, Reading, 2000
- [JARZ01] Jarzabek, S. and Zhang H.Y. XML-based Method and Tool for Handling Variant Requirements in Domain Models, submitted for publications, 2001
- [JOHN88] Johnson, R. and Foote, B. 1988. Designing reusable classes, *Journal of Object-Oriented Programming*, 1, 2, pp. 22-35.
- [KANG90] 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
- [KICZ97] KICZALES, et al. ‘Aspect-Oriented Programming,’ *Proc. European Conference on Object-Oriented Programming (ECOOP)*, Finland, Springer-Verlag LNCS 1241. June
- [OMG99] OMG, XML Metadata Interchange (XMI) 1.1 RTF, OMG Document ad/99-10-02, 25 October 1999
- [PARN72] Parnas, D. “On the criteria to be used in decomposing systems into modules”, *Communications of the ACM*, 15(12):1053-1058, December 1972
- [PARN76] Parnas, D. “On the Design and Development of Program Families”, *IEEE Trans. on Software Eng.*, March 1976
- [RUMB99] Rumbaugh, J., Jacobson, I. and Booch, G. *The Unified Modeling Language, Reference Manual*, Addison-Wesley, 1999
- [SOMM96] Sommerville, I. and Dean, G. PCL: A language for modeling evolving system architectures, *Software Engineering Journal*, 1996, pp. 111-121.
- [TARR99] TARR, P., OSSHER, H., HARRISON, W. and SUTTON, S. N Degrees of Separation: Multi-Dimensional Separation of Concerns, *Int. Conference on Software Engineering, ICSE’99*, Los Angeles, 1999, pp. 107-119
- [WONG01] Wong, T.W., Jarzabek, S., Soe, M.S., Shen, R., Zhang, H.Y. XML Implementation of Frame Processor, *Symposium on Software Reusability, SSR’01*, Toronto, Canada, May 2001