



ELSEVIER

Available at

www.ElsevierComputerScience.com

POWERED BY SCIENCE @ DIRECT®

The Journal of Systems and Software 68 (2003) 171–182

 **The Journal of
Systems and
Software**

www.elsevier.com/locate/jss

Handling variant requirements in domain modeling

Stan Jarzabek *, Wai Chun Ong, Hongyu Zhang

Department of Computer Science, School of Computing, National University of Singapore, Singapore 117543, Singapore

Received 23 December 2002; accepted 27 December 2002

Abstract

Domain models describe common and variant requirements for a family of similar systems. Although most of the notations, such as UML, are meant for modeling a single system, they can be extended to model variants. We have done that and applied such extended notations in our projects. We soon found that our models with variants were becoming overly complicated, undermining the major role of domain analysis which is understanding. One variant was often reflected in many models and any given model was affected by many variants. The number of possible variant combinations was growing rapidly and mutual dependencies among variants even further complicated the domain model. We realized that our purely descriptive domain model was only useful for small examples but it did not scale up. In this paper, we describe a modeling method and a Flexible Variant Configuration tool (FVC for short) that alleviate the above mentioned problems. In our approach, we start by modeling so-called domain defaults, i.e., requirements that characterize a typical system in a domain. Then, we describe variants as deltas in respect to domain defaults. The FVC interprets variants to produce customized domain model views for a system that meets specific requirements. We implemented the above concepts using commercial tools Netron Fusion™ and Rational Rose™. In the paper, we illustrate our domain modeling method and tool with examples from the Facility Reservation System domain.

© 2003 Elsevier Inc. All rights reserved.

Keywords: Domain analysis and modeling; Reuse; System families; Variant requirements; Frame technology

1. Introduction

The objective of domain analysis is to identify and model common and variant requirements for a family of similar systems. A domain model describes functionality that can be reused across family members, after possible customizations.

Feature models (Kang et al., 1990, 1998) have been widely used in domain analysis to model mandatory and variant (optional and alternative) requirements. But feature models do not tell us about the semantics of a domain. Therefore, we must also use other notations, for example UML, to enhance the meaning of domain concepts. Notations traditionally used in requirement analysis are also useful in domain analysis. Only few notations such as Object-Oriented inheritance and feature models offer a mechanism to model variants during analysis. Other notations, such as state diagrams, ac-

tivity diagrams or object interaction diagrams do not cater to variants. However, we can extend ordinary modeling notations with the concept of a variation point (Jacobson et al., 1997) to make them useful in domain modeling.

Modeling domain variants adds extra level of complexity to domain analysis, otherwise similar to requirement analysis for a single system. In our early work, we extended Data Flow, Entity-Relationship and State Transition diagram notations with variants (Cheong and Jarzabek, 1998). In later projects, we applied UML extension mechanisms (Rumbaugh et al., 1999) to model variants. Soon we realized that our purely descriptive models with variants did not scale up. During domain modeling, we face the following major difficulties:

1. One variant is often reflected in many different views of a domain (by a *domain view* we mean description of a domain in a specific notation such as use cases, activity diagrams, etc.). Tracing multiple occurrences of the same variant in different domain model views is a major challenge in domain modeling.

* Corresponding author. Tel.: +65-874-2863; fax: +65-779-4580.

E-mail addresses: stan@comp.nus.edu.sg (S. Jarzabek), zhanghy@comp.nus.edu.sg (H. Zhang).

2. One model view is often affected by many variants. The number of possible variant combinations grows rapidly.
3. Many variants are mutually dependent. Variant dependencies determine valid combinations of variants.

While each step in modeling variants may be simple, as the volume of information grows, domain models become notoriously difficult to understand. UML models of a single system may be quite complicated. With variants, models soon become unmanageable.

In our opinion, it is not possible to radically solve the problem at the model description level alone. Therefore, we changed the perception of what constitutes a domain model and how it is used. Instead of being just a set of descriptions, our domain model is flexible due to an active component that helps us in domain model interpretation and manipulation. In our approach, we start by modeling domain defaults. Domain defaults form an UML description of a typical system in a domain. Defaults are the starting point for understanding a domain itself and a range of systems in a domain. We describe variants as deltas from default requirements in so-called customizations scripts. A customization script contains commands that modify, add or delete required variants to/from those defined by defaults. We designed a tool called Flexible Variant Configuration (FVC for short). FVC interprets customization scripts and, based on selected variants, FVC promptly provides analysts with customized views of a domain model that meet the specific system requirement. FVC helps analysts understand the domain model by localizing the impact of a variant in a customization script and by producing customized model views on demand. Customization scripts as well as defaults can be easily modified, providing flexibility required during customization and evolution of a domain model.

We implemented FVC using frame technology and Fusion™ toolset by Netron Inc. (Bassett, 1997). Frame method and tools have an excellent record in industrial applications as an effective way to handle variants in reusable software. We extended application of the frame concept from programs to domain models. Our solution has a potential to apply one consistent approach throughout major stages of the development of system families. In the remaining part of the paper, we describe our solution, illustrating the principles of the approach with examples from our domain engineering project on the Facility Reservation System (FRS) family.

2. Related work

Background. A software system family (also called a Product Line) comprises a group of similar systems. The concept can be traced back to early Parnas work (Par-

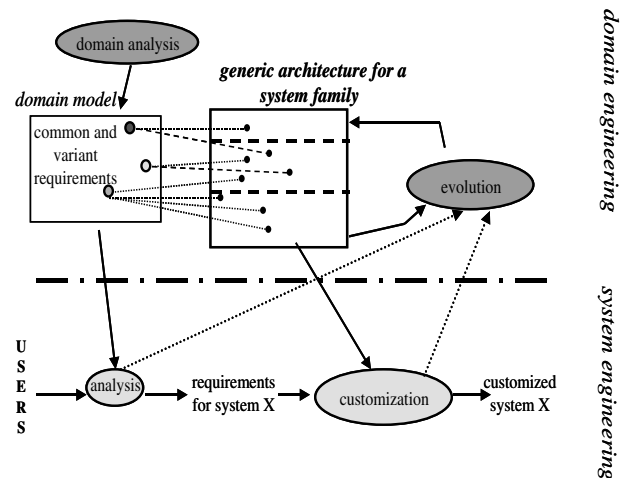


Fig. 1. Domain engineering and system engineering.

nas, 1976). Systematic support for system families by exploiting reuse opportunities emerges as a promising way to improve software productivity and quality. Domain engineering (above the dotted line in Fig. 1) is a process that leads to systematic support for a system family. Domain engineering delivers software assets that can be reused during analysis, design and implementation of family members (the system engineering process below the dotted line in Fig. 1). Major reusable assets include a domain model and a generic architecture for a system family.

A *generic architecture* for a system family defines an overall architecture for family members and provides implementation of both common and variant features. In system engineering, instead of developing family members from scratch, we build them by reusing a domain model and a generic architecture. As indicated in Fig. 1, domain engineering artifacts evolve based on feedback from system engineering.

In Domain-Specific Software Architecture (DSSA) approach (Tracz, 1995), requirements are given unique names and are qualified as mandatory, optional or alternative. In Feature-Oriented Domain Analysis (FODA) (Kang et al., 1990) and its newer version FORM (Kang et al., 1998), 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. Feature trees are organized mainly based on part-of relationship, i.e., children nodes in a tree are part-of the parent node. Feature diagrams allow us to model some of the structural dependencies among variants but must be complemented with other notations to enhance the semantics of the domain models.

The feature interaction problem in telecommunication domain was formulated in early 1990s by Zave (1993). Feature interaction occurs when one feature modifies or subverts the operation of another one. As

new features are added, it becomes increasingly difficult to keep all the interacting features under control. Research on feature interaction problem in telecommunication domain concentrated on detecting unwanted interactions. At the model level, formal specifications of feature behavior are used to analyze feature interactions. In the work described in this paper, we are also concerned with interactions among variant features but, at least so far, our objectives and methods are different from those in telecommunication domain. Our feature interactions are intentional. We attempt to create generic and flexible UML models in which variant features can occur in different configurations. Then, we develop methods to engineer variant features into the generic models. Customized models show system behavior for a required combination of variants. We are also concerned with the problem of tracing how a given variant feature affects different domain model views expressed in UML notations. To conclude, let us emphasize that there are many types of feature interactions we may be concerned with. In the Product Line context, for example, apart from behavioral interactions, we need to know how inter-related variant features interact in system code. In our approach, we apply the same technique, namely frame technology (Bassett, 1997), to describe how variant features affect (and interact with each other) in both software models describing system behavior and in code.

In the scenario-based requirement engineering method CREWS–SAVRE (Sutcliffe et al., 1998), alternative (variant) scenario paths are generated from generic use cases. Constraints and rule-based frames are used in scenario generation and validation, respectively. In CREWS–SAVRE, the generic use case is represented by actions and action-link rules, while in our approach we introduce variability to UML models and apply frame technology to automate the configuration of the variants.

Catalysis (D’Souza and Wills, 1998) proposes a UML-based approach for object and component-based development. To model a domain, Catalysis includes the concept of a model template: a generic piece of model where placeholders (marked up by angle brackets “ $\langle \rangle$ ” in UML diagrams) can be substituted to cater to specific applications. Our approach also supports variable substitution and in addition allows us to specify configuration of variants and model customization process. Instead of customizing UML models manually, we attempt to automate the customization process with the help of the FVC.

Jacobson et al. (1997) formulated the problem of modeling variants independently of a particular notation used, in terms of general modeling components, (i.e., any reusable analysis elements). Variants are exploited at variation points in modeling components. A variation point identifies one or more locations at which

the variation will occur. An analysis component is customized (specialized) by attaching one or several variants to its variation points. Authors described a number of mechanisms for modeling variants and customizing components.

Techniques such as application generators, frame technology (Bassett, 1997) and Aspect-Oriented Programming (AOP) (Kiczales et al., 1997) offer mechanisms for handling variants at the program level to enable reuse of code, but they have not been applied to address variants at analysis and design levels to enable reuse of models.

3. Facility reservation system family

We shall use FRS family to illustrate our domain modeling approach. FRSes are used at universities to reserve tutorial rooms and lecture halls, at companies to reserve meeting rooms and equipment of various types, at hotels to reserve rooms and conference facilities, etc. Users can manage their own reservations with the FRS (e.g. add, delete and modify their own reservations). Although the main purpose and the core functionality are similar across the FRS family, there are many variants on the basic theme. Here are some of the variant requirements in the FRS domain:

1. Most often, users manage their own reservations with an FRS. In some companies, however, users may send reservation requests to a *middleman* who makes reservations for them.
2. In some FRS systems, the facility’s record can only be deleted after the reservation on the facility is completed; in other FRSes, the facility record and associated reservation records can be deleted together; while in yet other FRSes, the facility record cannot be deleted at all.

The descriptive part of the FRS domain model consists of feature diagrams (Kang et al., 1990), domain defaults modeled in UML, domain defaults instrumented with variants and customization scripts. Domain defaults describe a typical system in a domain. Our FRS default model covers functionalities shown in Fig. 2. Defaults provide a convenient starting point to understand a domain and serve as a reference model for variants. We write customization scripts to specify how variants affect defaults.

In the paper, we discuss use case and workflow views of the FRS domain model. Of course, a domain model is not limited to use cases and workflows. Other UML notations such as state diagrams, object collaboration diagrams and sequence diagrams and non-UML notations such as Data Flow Diagrams can provide yet other views that are useful in a domain model (Tracz, 1995).

Make a Reservation
Delete a Reservation
Modify a Reservation
Search/Retrieve Reservations
Add a Facility
Delete a Facility
Modify a Facility
Search/Retrieve Facilities

Fig. 2. Functionalities covered by FRS defaults.

In this paper, however, we shall focus on use case and workflow domain model views.

Feature diagrams (Kang et al., 1990) are often used to classify variants and describe some of the dependencies among them. Fig. 3 shows an excerpt from the FRS feature diagram. We use extensions described in Czarnecki and Eisenecker (2000). The legend in Fig. 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 example of an optional requirement is

payment option (called FACILITY_CHARGE in Fig. 3). An alternative describes one-of-many requirements. An example of alternative requirement is deletion of reservation with re-confirmation or without it. An or-requirement describes any-of-many requirements. For example, a reservation can be made by requester himself, by a *middleman* or by both. Each logical grouping of variants may depend on other variants. For example, the reservation discount by block usage (BLOCK_RESV_DISC) variant is depends on the existence of block reservation (BLOCK_RESV) variant and the facility charge (FACILITY_CHARGE) variant. Dashed arrows in Fig. 3 indicate some of those dependencies.

4. Instrumenting domain defaults for flexibility

For different members of a system family, use cases may differ in the details of description, activity diagrams may differ in activities, decisions and control flows, entity classes may differ in attributes and methods, etc. To address variants, we instrument domain defaults for flexibility using frame technology from Netron Inc. (Bassett, 1997). Table 1 lists frame commands that the reader will find in examples below. Frame technology is supported by a comprehensive toolset Netron Fusion™, of which we used only frame processor to automate customizations required by the FVC.

We use term *f-default* to refer to domain defaults instrumented for flexibility with frame commands. An *f-default* (such as use case or workflow with variants)

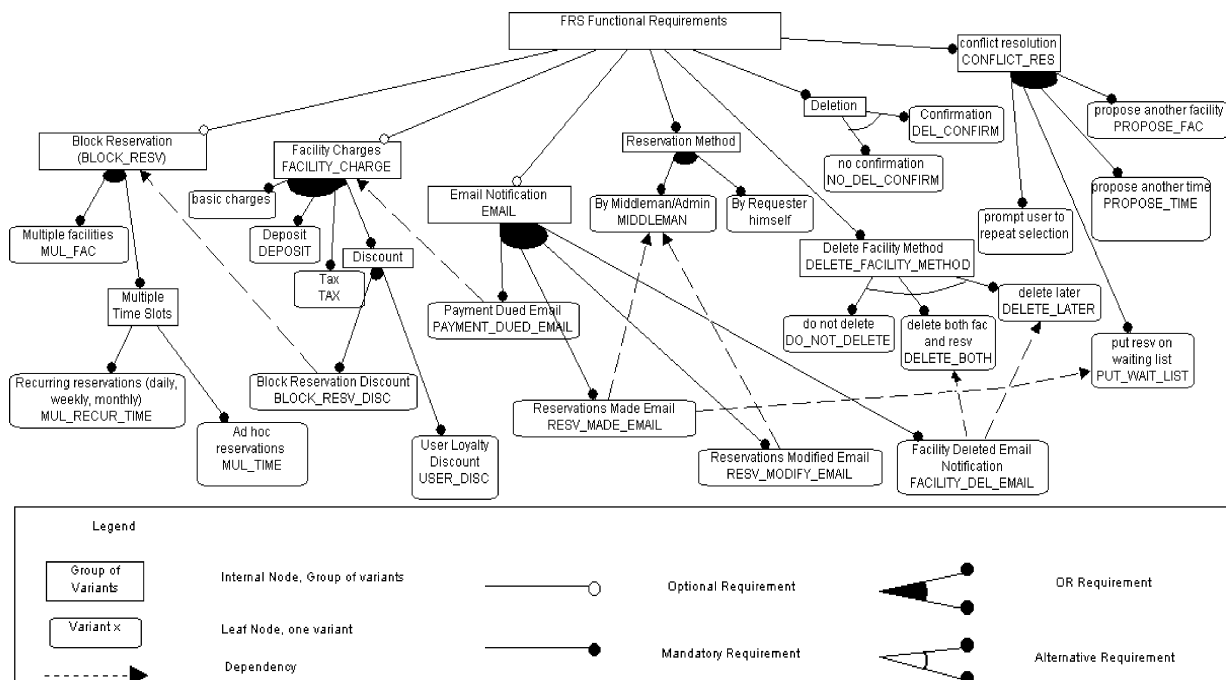


Fig. 3. FRS feature diagram with dependencies.

Table 1
Frame commands

Frame Command	Description
.COPY <i>f-default</i> .END-COPY <i>f-default</i>	When FVC encounters the COPY command, it includes a copy of specified <i>f-default</i> and applies customization commands that may be (optionally) specified.
.INSERT <i>name</i> .INSERT-BEFORE <i>name</i>	Inserts specifications into/before/after breakpoint <i>name</i> into the copied f-default.
.INSERT-AFTER <i>name</i>	
.BREAK <i>name</i>	Marks a breakpoint name in an f-default where default specifications can be modified
.SELECT <i>variable</i> .WHEN <i>value</i>	Selects one of many customization options based on the <i>value</i> of the <i>variable</i> .
.ORWHEN <i>value</i>	
.OTHERWISE .END-SELECT <i>variable</i>	
.REPLACE <i>variable</i> BY <i>value</i>	Sets the <i>value</i> of a <i>variable</i>

instrumented with frame commands can be customized by the FVC to obtain a specific domain model view. In the rest of the paper, we shall show how we instrumented default use cases and default workflows for flexibility.

5. Flexible use cases

One of the default use cases in FRS domain, called *Make Reservation*, is shown in Fig. 4. Variables, such as “FACILITY” and “RESPONSETIME” are the simplest means to inject flexibility into default models.

Fig. 5 shows f-default use case *Make Reservation* instrumented for flexibility with frame commands. Vari-

ables are assigned default values (by .REPLACE commands) that can be changed during customization.

Variants addressed in Fig. 5 include “reservations by a middleman” and “block reservation”. In the first case, only a middleman can make reservations on behalf of requesters and in the latter—users can reserve multiple facilities. The reader may find these variants in the feature diagram of Fig. 3.

A *customization script* specifies how to adapt an f-default to accommodate variants. For the MIDDLEMAN variant, a middleman must enter requester’s particulars when making or retrieving reservations. The customization script of Fig. 6 shows the impact of MIDDLEMAN variant on *Make Reservation* and *Retrieve Reservations* f-default use cases.

```

DEFAULT Make_Reservation_uc // suffix 'uc' indicates that the contents of this-default is a use case

1. Introduction

Make Reservation allows a user to select “FACILITY” and make new reservation.

2. Flow of events

    2.1 The user is identified.

    2.2 The system shows existing “FACILITY”s.

    2.3 The user selects a “FACILITY” and supplies the time period of reservation.

    2.4 If the “FACILITY” is reserved at the specified time, a warning message is prompted and user repeats
step 2.2. Otherwise, the system confirms the new reservation.

3. Special Requirement

System should respond to user input within "RESPONSETIME".

```

Fig. 4. Default use case *Make Reservation*.

```

F-DEFAULT Make_Reservation_uc // suffix 'uc' indicates that the contents of this f-default is a use case

.REPLACE FACILITY BY "meeting room"

.REPLACE RESPONSETIME BY "30 secs"

1. Introduction

Allows a user to select FACILITY and make new reservation.

2. Flow of events

    2.1 The user is identified.

    .BREAK MIDDLEMAN //a breakpoint at which variant " reservation by middlemen" can be addressed

    2.2 The system shows existing "FACILITY"s.

    2.3 The user selects a "FACILITY" and supplies the time period of reservation.

    .BREAK BLOCK_RESV_FAC //a breakpoint at which variant "select multiple facilities" can be addressed

    2.4 If the "FACILITY" is reserved at the specified time, a warning message is prompted and user repeats
    step 2.2. Otherwise, the system confirms the new reservation.

3. Special Requirement

System should respond to user input within "RESPONSETIME".

```

Fig. 5. f-default use case *Make Reservation*.

```

CS MIDDLEMAN // Customization Script for MIDDLEMAN variant

// customize Make_Reservation_uc

.COPY Make_Reservation_uc

.INSERT-AFTER MIDDLEMAN

If the user is a middleman, then enter the requester's information.

.END-COPY Make_Reservation_uc

...

// customize Retrieve_Reservation_uc

.COPY Retrieve_Reservation_uc

.INSERT-AFTER MIDDLEMAN

If the user is a middle man, then when searching/retrieving reservations by requester, the user enters userid of
the requester on behalf of the requester

.END-COPY Retrieve_Reservation_uc

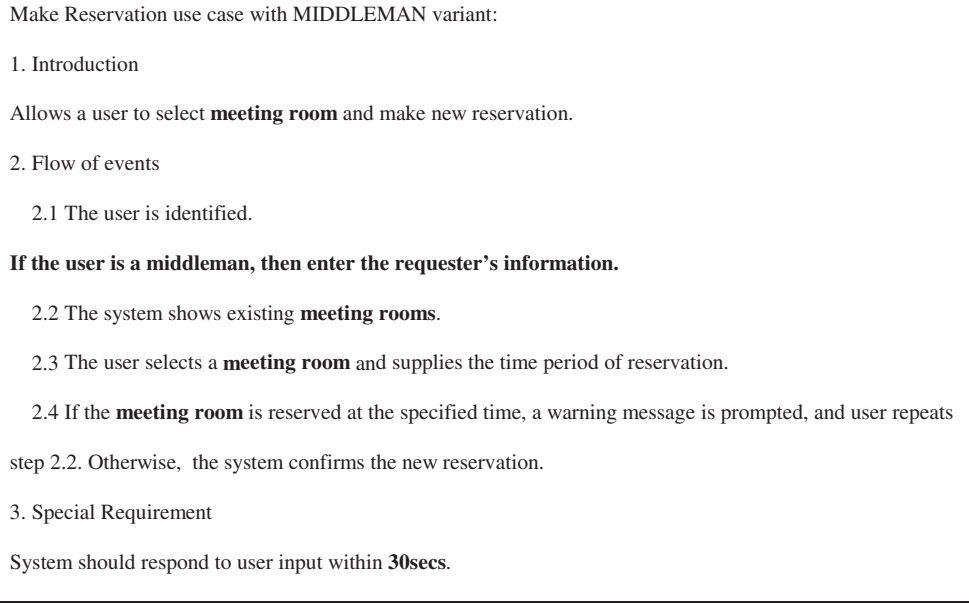
...

```

Fig. 6. A partial customization script for the MIDDLEMAN variant.

Fig. 7 shows the customized *Make Reservation* use case. Bold lines highlight the included variants.

One customization script may adapt multiple f-defaults. This allows us to systematically deal with variant

Fig. 7. A customized *Make Reservation* use case.

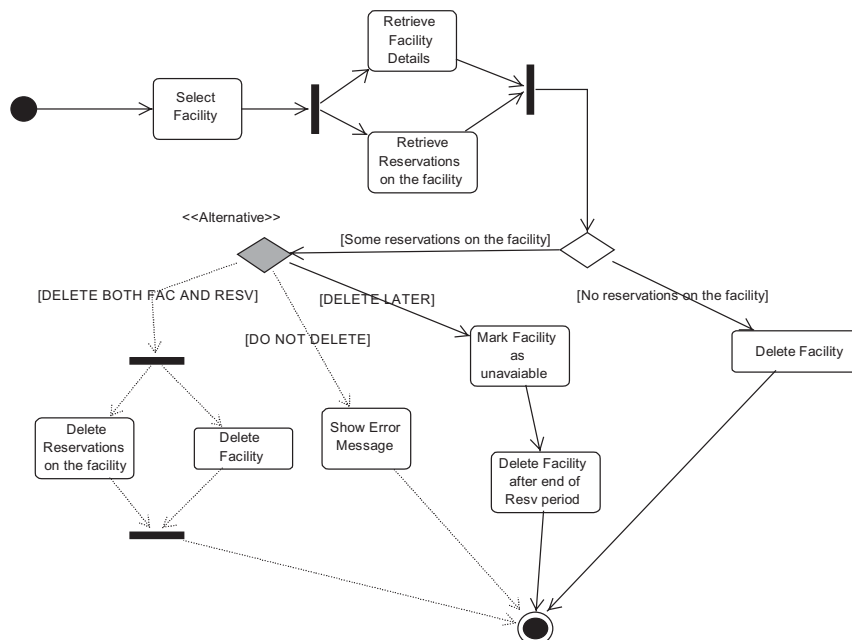
constraints and dependencies. A customization script localizes the changes required for one variant, making maintenance and evolution of the domain model easier as concerns of variants are separated.

6. Flexible workflows

A workflow describes detailed flow of activities in a use case. We use UML activity diagrams to model

workflows. To handle variants in workflow, we extended activity diagrams with variation points to model alternative and optional flows of activities. Fig. 8 depicts *Delete Facility* workflow with the DELETE_FACILITY_METHOD variant.

The shaded diamond is a meta-symbol that marks the variation point. It denotes a decision related to different choices of facility deletion methods. In the diagram, the shaded diamond is stereotyped to become an «alternative» decision, whereby one of the paths from

Fig. 8. *Delete Facility* workflow with variation point.

```

F-DEFAULT Delete_Facility_wf           // suffix 'wf' indicates that the content of this f-default is a workflow

<XMI xmi.version = '1.0'>

  <XML.header>

    <XML.metamodel xmi.name = 'UML' xmi.version = '1.1' />

  </XML.header>

  <XML.content>

    ...

    <Behavioral_Elements.State_Machines.ActivityModel xmi.id = S.10001'>

    <Foundation.Core.ModelElement.name> FRS Activity Diagram </Foundation.Core.ModelElement.name>

    ...

    // Activity definitions here

    .SELECT ;DELETE_FACILITY_METHOD

    .WHEN = "DELETE_LATER"

    <Behavioral_Elements.State_Machines.ActionState xmi.id = 'G.100'>

      <Foundation.Core.ModelElement.name> Mark Facility as unavailable </Foundation.Core.ModelElement.name>

      ...

    </Behavioral_Elements.State_Machines.ActionState>

    <Behavioral_Elements.State_Machines.ActionState xmi.id = 'G.101'>

      <Foundation.Core.ModelElement.name> Delete facility after end of resv period </Foundation.Core.ModelElement.name>

      ...

    </Behavioral_Elements.State_Machines.ActionState>

    .WHEN = "DO_NOT_DELETE"

    <Behavioral_Elements.State_Machines.ActionState xmi.id = 'G.100'>

      <Foundation.Core.ModelElement.name> Show Error Message </Foundation.Core.ModelElement.name>

    </Behavioral_Elements.State_Machines.ActionState>

    .WHEN = "DELETE_BOTH"

    ...

    .END-SELECT ;DELETE_FACILITY_METHOD

    ...

    </Behavioral_Elements.State_Machines.ActivityModel>

  </XML.content>

</XMI>

```

Fig. 9. f-default workflow *Delete Facility*.

the decision point will be included in the customized diagram. The dotted arrow shows the possible path leading from the meta-decision point to another activity state. The default activity sequence is indicated by solid arrows.

We wish to use the same approach to instrument workflow diagrams for flexibility as we applied to textual notation of use cases. To achieve this, we convert UML diagrams into equivalent textual representation and instrument the text with frame commands for flex-

ibility. We can then perform the same kind of adaptation on textual workflow as we have done on use cases. After customization, we convert the text back to diagrams.

We use an XMI (XML Metadata Interchange) tool Unisys Rose/XMI to convert the UML diagrams to equivalent textual representation in XML. XMI (OMG, 1999) 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.

Fig. 9 shows an example of the XML representation of the *Delete Facility* workflow. To accommodate the anticipated “DELETE_FACILITY_METHOD” variant, we instrument the XML document with .SELECT frame commands, which indicate the places where the anticipated customizations will occur.

Fig. 10 shows a customization script for the “DELETE_FACILITY_METHOD” variant. It adapts the textual *Delete Facility* workflow as shown in Fig. 9,

specifying the ‘DELETE_FACILITY_METHOD’ variant as “DO_NOT_DELETE”.

When the FVC encounters the .COPY command, it will start processing the *Delete_Facility_wf* f-default, selecting the specific contexts that cater to the “DO_NOT_DELETE” requirement (in this case, the “Show Error Message” activity will be included in the workflow).

The customized textual workflow can be converted back to UML activity diagram by using the Unisys Rose/XMI tool. Fig. 11 shows the customized *Delete Facility* workflow in UML.

7. Flexible variant configuration tool

The purpose of the FVC tool is to help analysts understand domain model defaults and variants. Having selected variants, analysts need to obtain requirement specifications for a family member that satisfies those

```
CS DELETE_FACILITY_METHOD // Customization Script for DELETE_FACILITY_METHOD variant
...
//Customize Delete Facility workflow
.COPY Delete_Facility_wf
.REPLACE DELETE_FACILITY_METHOD BY “DO_NOT_DELETE”
.END-COPY Delete_Facility_wf
...
```

Fig. 10. A partial customization script for the DELETE_FACILITY_METHOD variant.

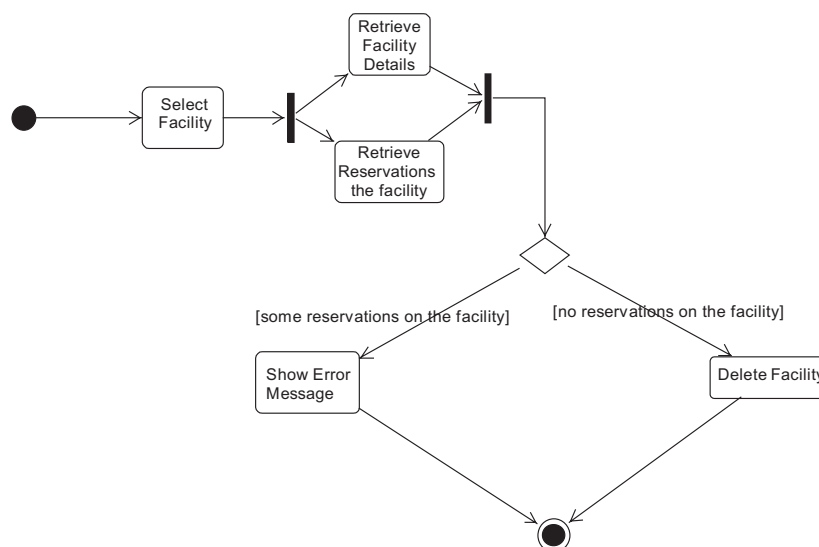


Fig. 11. A customized *Delete Facility* workflow.

variants. This is often a “try-and-error” cycle, in which analysts explore defaults and variants to learn about functional concerns described by a domain model, dependencies among variants, etc. Rather than working with models by hand, a time-consuming and error-prone process, the FVC automates the process of producing customized domain model views for selected variants (i.e., requirement specifications for a family member that satisfies those variants).

The overall process of obtaining customized views of a domain model is outlined in Fig. 12. An analyst starts by inspecting the feature diagram (such as the one in Fig. 3), to select required variants. Next, the FVC traverses the feature diagram and extracts customization scripts and f-defaults relevant to selected variants. Finally, FVC synthesizes these customization scripts and f-

defaults, performs customizations by calling the frame processor to execute frame commands embedded in f-defaults and customization scripts, and outputs customized domain model views such as use cases or workflows.

8. Dependencies between variants

As the volume of information in a domain model grows, the number of possible requirement 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, we know that sometimes not all of the combinations are valid. The total number of combinations is

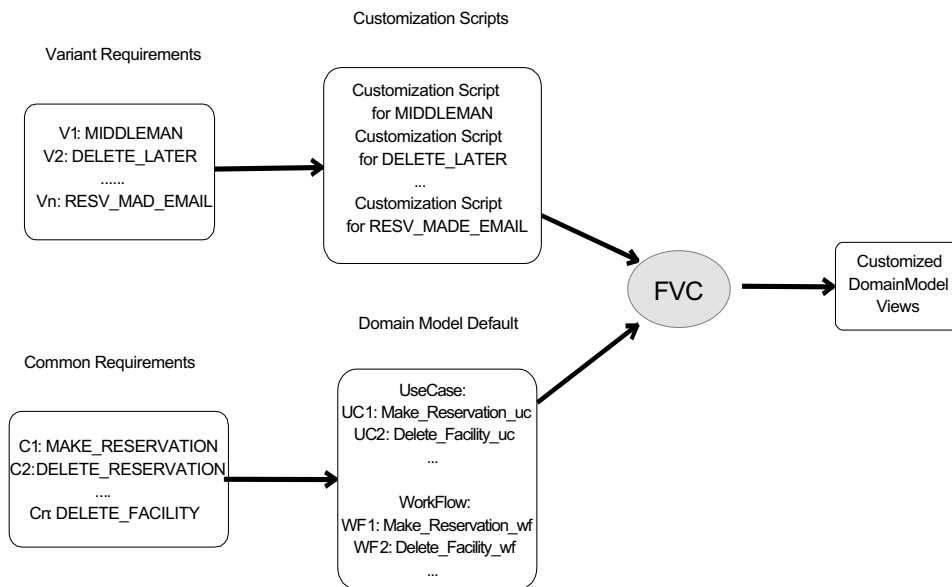


Fig. 12. An overview of the customization process.

```
CS DELETE_FACILITY_METHOD // Customization Script for DELETE_FACILITY_METHOD variant
...
//Customize Delete Facility workflow
.COPY Delete_Facility_wf
.REPLACE DELETE_FACILITY_METHOD BY "DELETE_LATER"
.BREAK FACILITY_DEL_EMAIL // An implicit breakpoint that is reserved for possible email
notification activities that may occur.
.END-COPY Delete_Facility_wf
...
```

Fig. 13. Implicit breakpoint for variant dependencies.

less if requirement r_1 and requirement r_2 are mutually exclusive. Some variants are also bound to depend on other variants. We modeled some of the variant dependencies as dashed arrows in Fig. 3. For such cases, it is important to address inter-dependent variants within customization scripts. Then, we can also automate the process of integrating customization scripts for inter-dependent variants. In our project, we only scratched the surface of this difficult problem so we shall only briefly comment on initial experiments.

We used breakpoints to control customizations related to inter-dependent variants. We define two types of breakpoints, namely implicit and explicit breakpoints. Explicit breakpoints are shown in f-defaults. Implicit breakpoints are not directly visible. They are controlled, adapted and hidden by the customization scripts. Breakpoints may sometimes obscure first reading but they allow us to better support the intended variability. As an example of addressing variant dependencies by using breakpoints, consider the “Facility Deleted Email Notification” (FACILITY_DEL_EMAIL) variant in Fig. 3. Only after the facility is deleted can we send the “facility deleted” email notification. So we say that the FACILITY_DEL_EMAIL variant is dependent on the DELETE_FACILITY_METHOD variant. Fig. 13 shows the customization scrip that contains an implicit breakpoint indicating how these dependent variants affect the *Delete Facility* workflow.

9. Conclusion

Although most of the notations, such as UML, are meant for modeling a single system, they can be extended to model variants. We have applied such extended notations in our projects. However, we soon found out that our models with variants were getting overly complicated, undermining the major role of domain analysis which is understanding. One variant was often reflected in many models and any given model was affected by many variants. The number of possible variant combinations was growing rapidly and mutual dependencies among variants even further complicated the domain model. We realized that our purely descriptive domain model was only useful for small examples but it did not scale up. In this paper, we describe a modeling method and a FVC tool that alleviate the above mentioned problems. In our approach, we start by modeling requirements that characterize a typical system in a domain (so-called domain defaults) in UML. Then, we describe variants as deltas in respect to domain defaults. The FVC interprets variants to produce customized domain model views for a system that meets specific requirements. We implemented the above concepts using commercial tools Netron Fusion™ and Rational Rose™.

We experimented thoroughly with proposed methods on the use case and workflow views of the FRS domain model. Our experimentation covered FRS functionalities listed in Fig. 2. We addressed over 50 variants. Many of these variants displayed a range of mutual dependencies, including dependencies among separate variant groups and nested variants. However, in our experiments, we only managed to handle a small subset of variant dependencies in a systematic way. We feel this problem is important but at the same time very difficult to solve.

Despite limitations, we believe our approach offers benefits of reduced complexity of a domain model, better support for customization and ease of evolution. In the future work we shall address the following issues:

Experiments 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 (such as state diagrams, object-collaboration diagrams, etc.) and experiment on a larger scale in the future.

Variant dependencies. We are experimenting with various types of variant dependencies in FRS and in Computer Aided Dispatch (CAD) domains.

Extend and integrate the proposed method with the customization of the generic architecture and program code. 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.

Acknowledgements

This project was supported by research grant NSTB/172/4/5-9V1 funded under the Singapore-Ontario Joint Research Programme by the Singapore National Science and Technology Board and Canadian Ministry of Energy, Science and Technology, and by NUS Research Grant R-252-000-066. We are indebted to Netron Inc. for letting us use their product Fusion. Earlier work by NUS student Cheong Yu Chye with frame technology on the facility reservation system contributed much to methods described in this paper.

References

- Bassett, P., 1997. Framing Software Reuse—Lessons from Real World. Yourdon Press, Prentice Hall.
- Cheong, Y.C., Jarzabek, S., 1998. Modeling variant user requirements in domain engineering for reuse. In: Proc. European-Japanese Conference on Information Modeling and Knowledge Bases, Finland.
- Czarnecki, K., Eisenecker, U., 2000. Generative Programming: Methods, Tools, and Applications. Addison-Wesley.

- D'Souza, D., Wills, A., 1998. Objects, Components, and Frameworks with UML: The Catalysis Approach. Addison-Wesley.
- Jacobson, I., Griss, M., Jonsson, P., 1997. Software Reuse Architecture, Process and Organization for Business Success. Addison-Wesley.
- Kang, K. et al., 1990. Feature-oriented domain analysis (FODA) feasibility study. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh.
- Kang, K. et al., 1998. FORM: a feature-oriented reuse method with domain-specific reference architectures. In: Balci, O. (Ed.), Systematic Software Reuse. Ann. Software Eng. 5.
- Kiczales, G., et al., 1997. Aspect-oriented programming. In: Proc. European Conference on Object-Oriented Programming (ECOOP), Finland.
- OMG, 1999. XML Metadata Interchange (XMI) 1.1 RTF, OMG Document ad/99-10-02.
- Parnas, D., 1976. On the design and development of program families. IEEE Trans. Software Eng. 2 (16), 1–9.
- Rumbaugh, J., Jacobson, I., Booch, G., 1999. The Unified Modeling Language Reference Manual. Addison-Wesley.
- Sutcliffe, A., Maiden, N., Minocha, S., Manuel, D., 1998. Supporting scenario-based requirement engineering. IEEE Trans. Software Eng. 24 (12), 1072–1088.
- Tracz, W., 1995. DSSA (Domain-Specific Software Architecture) pedagogical example. ACM Software Eng. Notes (July), 49–62.
- Zave, P., 1993. Feature interactions and formal specifications in telecommunications. IEEE Comput. XXVI (8), 20–30.