NOTE TO USERS

This reproduction is the best copy available.

UMI®

DISSERTATION

DEFINING DOMAIN-SPECIFIC OBJECT-ORIENTED MODELING LANGUAGES AS UML PROFILES

Submitted by
Emanuel Sylvester Grant
Computer Science Department

In partial fulfillment of the requirements
for the Degree of Doctor of Philosophy
Colorado State University
Fort Collins, Colorado
Fall 2002

COLORADO STATE UNIVERSITY

November 1, 2002

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY EMANUEL SYLVESTER GRANT ENTITLED DEFINING DOMAIN-SPECIFIC OBJECT-ORIENTED MODELING LANGUAGES AS UML PROFILES BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

Committee Member Dr. James M. Bieman

Committee Member Dr. Daniel E. Turk

Committee Member Dr. Gregory L. Florant

Adviser Dr. Robert B. France

Department Head Dr. Dale H. Grit

ABSTRACT OF DISSERTATION

DEFINING DOMAIN-SPECIFIC OBJECT-ORIENTED MODELING LANGUAGES AS UML PROFILES

The use of general purpose modeling languages (GPMLs) in specifying software applications is giving way to the increasing usage of domain-specific modeling languages (DSMLs). DSMLs offer a vocabulary of terms and concepts that are fundamental to the problem and solution domains, whereas GPMLs constructs are usually too generic to be directly applied in some domains.

The *Unified Modeling Language* (UML) is a comprehensive set of graphical and textual notations for modeling various views of software systems, using *object – oriented* concepts. Though the UML has been designed to satisfy the requirements of the typical software modeling tasks, there are some projects that require features beyond those explicitly defined in the UML. Application of the UML extension mechanisms on a coherent set of UML metamodels, which are defined by the requirement of a specific application domain or enterprise, results in what is termed a UML *profile*. Profiles may be used to package domain-specific modeling entities that are applied in building application models for the domain.

The objective of this research is to develop domain-specific object-oriented modeling languages that are defined in the format of UML profiles. Most DSMLs may be viewed as high level *programming* languages, because they are usually applied near the end of the design phase of application development and are generally textual models. DSML, as used in this work, are graphical in presentation, and are intended to be used at the analysis and design stages of application development.

The result of this work will be the definition of components of DSMLs that constitute the syntax and semantics of the language, and the formulation of a process for defining DSMLs in the format of UML profiles.

The main benefits that are realized from using DSML are the reuse of standardized domain artifacts, reduction in the time to deliver complete products, and more easily maintained applications.

Emanuel Sylvester Grant
Computer Science Department
Colorado State University
Fort Collins, Colorado 80523
Fall 2002

ACKNOWLEDGEMENTS

In order to express thanks and acknowledgment to those persons who have influenced and supported my life long dream of attaining this level of academic accomplishment, I must first look back to my years in elementary school. In so doing, I express thanks for the early years of academic naturing given to me by Mrs. Una Finley of Mico Practising All-Age School, Kingston, Jamaica. My next expression of thanks for the guidance and support given to me during my precious and memorable high school years is to Mr. Peter Maxwell of Kingston College, Kingston, Jamaica.

I thank the members of my graduate committee: Dr. James Bieman, Dr. Daniel Turk, and Dr. Gregory Florant for their many words of encouragement, insight, and academic and personal advice that made it possible for me to realize this life long dream. I express special thanks to my advisor, Dr. Robert France for not only his academic advisement, but more for his belief in my abilities, and years of mentoring and friendship.

To all my family and friends, who through their varied ways of support and encouragement, I express my deepest thanks and hope that I will be able to make a positive difference in the lives of others, as you have made in mine.

DEDICATION

This dissertation is dedicated to my daughter, Allison Roxanne. Since her birth she has been the dominant inspiration behind my pursuing and achieving all that I desire in life.

Fly away home...
Fly away home...
One bright morning when when my work is over man - we shall fly away home,
One bright morning when when my work is over man - we shall fly away home,
Fly away home...
Fly away home...

Robert Nesta Marley OM. 1946 - 1981 (Jamaica)

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACM	Association of Computing Machinery
AE	Application Engineer, Application Engineering
ASM	Application Specification Model
BNF	Backus Naur Form
CA	Commonality Analysis
CASE	Computer-Aided Software Engineering
CCM	Context Conceptual Model
CD	Class Diagram
CWM	Common Warehouse Meta-model
DA&D	Domain Analysis & Design
DCD	Dynamic Concept Dictionary
DCM	Dynamic Concept Model
DLE	Domain Language Engineering
DM	Domain Model
DSDR	Domain-Specific Development Rules
DSL	Digital Subscriber Line
DSL	Domain-Specific Language
DSML	Domain-Specific Modeling Language
DSPL	Domain-Specific Programming Language
DSSA	Domain-Specific Software Architecture
DSSEE	Domain-Specific Software Engineering Environment
EM	Extension Mechanism
FAST	Family-oriented Abstraction, Specification, and Translation
FODA	Features-Oriented Domain Analysis
GAL	Graphical Adapter Language
GME	Generic Modeling Environment
GPL	General Programming Language
GPML	General Purpose Modeling Language
GUI	Graphical User Interface
HTA	Hierarchical Task Analysis
ISDN	Integrated Service Digital Network
ISIS	Institute of Software Integrated Systems
IT	Information Technology
ITU	International Telecommunication Union

LIST OF ABBREVIATIONS

MDA	Model Driven Architecture
MIRG	Methods Integration Research Group
MMF	Meta-Modeling Facility
MOF	Meta Object Facility
NASA	National Aeronautics and Space Administration
NEP	Network Element Processor
OCL	Object Constraint Language
OD	Object Diagram
OO	Object-Oriented
PLA	Product Line Architecture
POTS	Plain Old Telephone Service
RIACS	Research Institute for Advance Computer Science
SCD	Static Concept Dictionary
SCM	Static Concept Model
SE	Software Engineering
SEI	Software Engineering Institute
SMI	Stanford Medical Infomatics
SPS	Service Provisioning System
SSGRG	Software Systems Generator Research Group
UML	Unified Modeling Language
VL	Visual Language
WFR	Well-Formedness Rule

Chapter 1

Introduction

The title of the dissertation, *Defining Domain-Specific Object-Oriented Modeling Languages as UML Profiles* reflects the following:

- The research work centers on issues related to the definition of analysis and design level object-oriented modeling languages for specific software domains, and
- The use of the Unified Modeling Language (UML) as the notation for such domain-specific modeling languages.

The above bulleted items form the basis for the research thesis, which is:

> **The development and use of domain-specific object-oriented modeling languages in mature and well-defined software development domains facilitates extensive reuse of modeling experiences that are intrinsically embedded in the language components, and will result in the delivery of software systems that are produced from standardized domain modeling artifacts.**

The work of this dissertation (definition of domain specific modeling languages (DSML) [25, 26, 86, 89, 91]) draws on work related to software reuse [5, 28, 39, 52, 59, 67] and modeling notations and techniques [43], and on work in the more general area of software engineering (SE) [18, 40, 51, 54, 82]. The research framework is illustrated in Figure 1.1: the DSML technology is viewed as a component of reuse technology with modeling notations and techniques, and reuse technologies being components of software engineering.

The origin of the term software engineering (SE) dates back to the 1960s and may be viewed as all the activities, processes, tools, documentations, and practices that are applied in a scientific manner in order to deliver software products that satisfy the stated requirements [85]. The opening chapter of Shaw and Garlan's text on software architecture offers

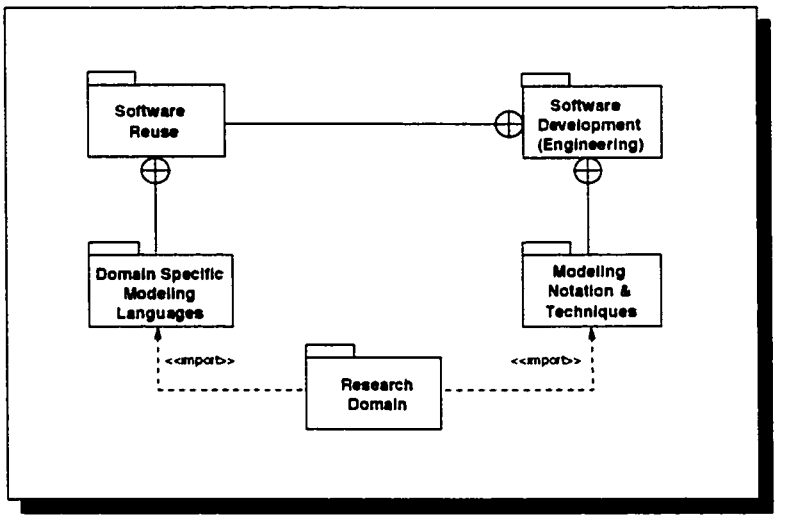

Figure 1.1: The Research Framework

an insightful chronicle of the evolution of SE [85]. The notion of re-usability is fundamental in all the established engineering fields, and is now being applied in a similar manner to software development. Software reuse in its is simplest form is the use (adaptation) of *parts* from one system in the production of other software systems that have some features in common [68]. This type of reuse is accidental (i.e. the *parts* that have been reused were not developed with reuse as an intended purpose). The other form of reuse occurs when software artifacts are developed specifically to be reused in many systems/applications. Such artifacts are developed not with a particular application in mind, but a family of applications and systems [68].

Modeling notations are fundamental to the representation of problems and solutions in an application domain and determine the types of problem and solution models that a developer may be able to create. Domain-specific modeling languages are graphical and/or textual notations that offer a syntax and semantics of terms and concepts that are intimately bound to the domain under consideration. These languages are used by developers to express models of both the problems and solutions within the domain, in a format that is native to the domain.

1.1 Problem Statement

Software developers/engineers have been grappling with the demand for more reliable, maintainable, and correct software systems that have to be developed in an environment of contracting development time [95]. Research in software development that is geared towards meeting the demands placed on the software practitioner has taken on a multi-pronged approach. A number of these approaches show promise of offering advances in software development. One such approach is the development and use of domain-specific technology in well defined and mature domains [97]. Domain-specific technology, which includes the development and use of domain-specific modeling languages, is a sub-area of the wider field of software reuse technologies (as illustrated in Figure 1.1). Reuse technology can be applied in two dimensions [94]:

1. *horizontal* (generic) reuse - occurs where architectural patterns (e.g. creation, structural, behavioral [33]), data structures (e.g. queues, stacks [60]), and algorithms (e.g. searches, sorts, list manipulation [14]) are applied in multiple application domains.

2. *vertical* (domain-specific) reuse - occurs when artifacts of software development are created for a *family of products* within a specific-domain (this is the type of reuse researched herein). Assets in this dimension are characterized by their application to a specific domain (e.g. mobile pager system [29]).

An organization that views reuse experience as an attribute of itself, rather than as an attribute of individuals, is better able to capitalize on its collective experiences during system development [38]. Such an organization is better able to *learn* from its collective experiences, and thus is in a better position to increase its productivity and the quality of its products. In this context, systematic reuse of high quality analysis and design experience during the development of software systems can lead to significant improvement in the quality of the product. The use of general purpose modeling languages (GPMLs) in specifying software applications is giving way to the usage of domain-specific modeling languages (DSMLs). This is evident from the large number of DSLs being developed and used,

such as those presented at the USENIX[1] conferences on DSL, and the ACM workshops on DSL[2]. DSLs offer a vocabulary of terms and concepts that are intimate to the problem and solution domains, whereas GPMLs constructs are usually too generic to be directly applied in expressing the problem or solution for some domains [10]. Most DSMLs examined in this research are high level textual programming languages. Such DSMLs offer very little support for modeling at the analysis, and design levels of application development.

The *Unified Modeling Language* (UML) [9, 43, 44]is a comprehensive set of graphical and textual notations for modeling various views of software systems, using object-oriented (OO) concepts. The UML is a standard modeling notation that was developed in response to the problems arising out of a proliferation of OO modeling notations, and has been accepted as the *de facto* modeling notation for object-oriented system [9].

As the use of the UML has grown, so has the need for it to satisfy the specific requirements of various user groups. The initial response to this demand was to add new features and concepts to the UML [58]. Though the UML has mechanisms to tailor the standard UML concepts to specific requirements, such mechanisms were not extensively used in the early releases of the UML specification because of limited tool support, and ambiguities in the definition of some of the UML concepts. Thus the practice of adding more features and concepts to the UML resulted in it becoming large and difficult to use.

This research aims to make an original contribution to academic work on the derivation of modeling notations that support rapid development of high quality software through reuse of domain-specific concepts. This research develops processes for defining *domain-specific modeling languages* (DSML), based on the UML. DSMLs offer application engineers a set of domain-specific modeling concepts that are native to an application domain, and incorporate analysis and design development decisions related to domain products. Armed with such languages, software application engineers are able to develop models of the problems and

---

[1]See the USENIX web site at *http://www.usenix.org*

[2]See the ACM DSL web site at *http://www-sal.cs.uiuc.edu/~kamin/dsl/*

solutions for the desired software products that are more reliable [69].

1.2 Research Objectives and Plan

The objective of this research is to develop a systematic approach to defining, domain-specific modeling languages that are constructed using the UML. These DSMLs provide domain-specific modeling constructs that are used by application engineers to create application models at the analysis and design levels. Because these constructs have in most cases, a precise basis (the formally constrained stereotypes) they may be rigorously analyzed (though not completely) by OCL analysis tools before being translated to code. A type of rigorous analysis is demonstrated in one of the case studies conducted in this research. In such an environment, *Domain-Specific Software Engineering Environment* (DSSEE) application engineers will develop models using constructs that directly reflect domain concepts as these models will incorporate expert experiences related to development in the domain (see Figure 1.2). A detailed description of Figure 1.2 is given in Section 3.2.

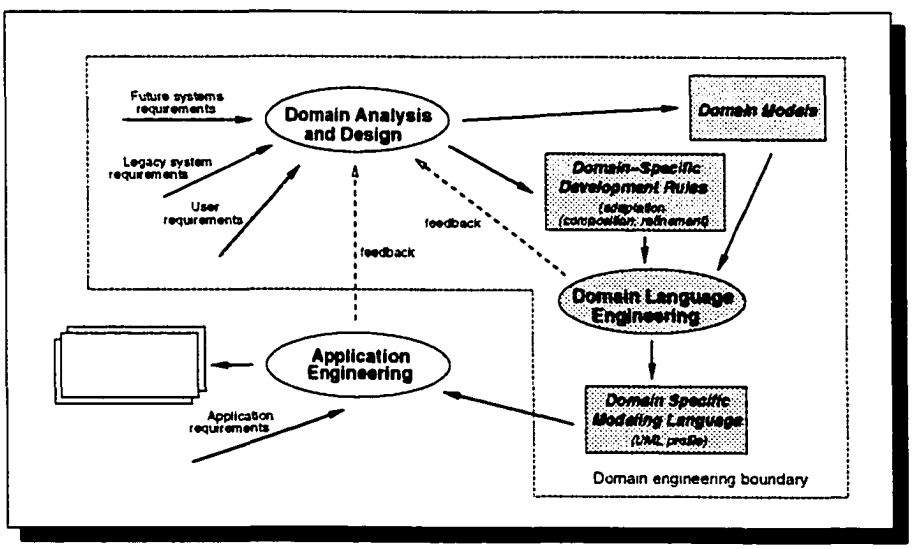

Figure 1.2: Domain-Specific Software Engineering Environment (DSSEE)

The research documented in this dissertation intends to make a contribution to work in the following specific areas:

- Definition of domain-specific modeling languages.

- Definition of precise syntax and semantics for the specialized UML model element used in defining domain-specific modeling languages.
- Specialization of UML model elements and meta-models, that define the components of domain-specific modeling languages.
- Derivation of a process for developing domain-specific modeling languages based on the UML.
- Demonstration of the development of domain-specific modeling languages so as to realize a high degree of reuse in application engineering.

The approach taken in conducting the research is targeted at producing a set of deliverables. These deliverables are directly obtained from the areas of researched identified at the end of the previous Section 1.1, and are listed below:

- A precise representation of the syntax and semantics of domain-specific modeling languages by using the UML extension mechanisms (EM), and denotational-like mappings on the UML model elements used in defining the domain-specific modeling language.
- Techniques for defining domain-specific modeling languages as UML profiles.
- Demonstration of the development of domain-specific modeling languages.
- Results from the application of the research concepts on case studies from real world domains.

In order to accomplish this task (production of the deliverables) and demonstrate the possible continuing benefits of this work the following research plan will be implemented:

1. A review of the UML concepts that are pertinent to the research work undertaken, specifically the UML *meta-modeling, model management, extension mechanisms, profile, syntax* and *semantics* concepts.
2. An analysis of previous works on domain-specific modeling languages, for the purpose of determining the essential properties of DSML.
3. Development of the deliverables of the research.
4. Presentation of ideas for future work that have been identified during the research.

There are a number of benefits that may be realized from the use of DSML technology in system development. Among these benefits are: (1) increases in reliability, maintainability, and productivity; (2) increase in the level of abstraction of the system specification; (3) and the incorporation of domain knowledge in the DSML. Albeit, the demonstration of these benefits is not within the scope of this research.

1.3 Organization and Scope

Throughout this report the term *domain-specific modeling language* will be used, rather than the generic term *domain specific languages*, except in places where referenced works use the term *domain specific languages*. This terminology expresses the focus of this research, i.e. the use of UML variants for *modeling* software system artifacts in specific domains. In this work the emphasis is on the *graphical modeling* aspects of DSML. Use of the term *modeling* refers to the representation of problem and solution entities in primarily *graphical format*.

The general approach taken in this report is that whenever a concept is introduced a definition (that supports the work) for the concept is presented. This definition may be in the form of (1) a statement that has been derived from a review of a number of published definitions, (2) a statement that has been taken verbatim from a published work, or (3) a statement that has been taken from a published work, and tailored to support the intent of the work.

1.3.1 Scope

For the purpose of consistency, the work presented here is based on the Unified Modeling Language (UML) Specification version 1.4 [43]. Errors that are encountered in this version will be noted.

The UML Specification [43] list a number of models that may be developed to present particular views of a software system. At the requirement phase the two models that are chiefly used are the *use case* and analysis *class diagram*. These are complemented at the design phase by design class diagram, the behavioral diagrams, which comprise *state chart*, *activity*, and *interaction* (*sequence* and *collaboration*) diagrams. At the implementation

phase the UML models are the *component* and *deployment* diagrams. The DSML defined in this work will be used to specify a desired sub-set of these model.

In developing DSMLs for different domains there will be requirements for different sub-sets of UML models. The goal is to use a core set of models that facilitates the implementation of DSMLs. Detail descriptions of the UML models used in defining DSMLs, in this work, are presented in Section 2.5.6. *Component* and *deployment diagrams* - are used at the implementation phase of system development, and are not considered in this work. Language components that are used to represent the use case, activity, collaboration, class, state chart, and sequence diagrams are documented herein.

1.3.2 Chapter description

Chapter 2 presents an overview of the states of research and practice that contribute to this work, and presents examples of works in some of these areas. *Chapter* 3 presents a description of DSMLs that includes an in-depth look at the use of the UML EM concepts used in defining DSMLs. *Chapter* 4 presents the processes that are used to construct DSMLs. *Chapter* 5 and 6 describes the application of the processes of Chapter 4 on real industrial domains. *Chapter*7 documents the experience gained during the development of the theory of the research and the application of that theory in the form of two case studies on real industrial problems. *Chapter* 8 is the concluding chapter of the dissertation and contains the following sub-sections:

- *Results* - a summary of the observed results of the documented work.
- *Future work* - in this area of research. This will include a look at the requirements for defining domain models, stereotypes, constraints, meta-models, for the other UML models not considered in this research.
- *Contribution* - this concluding chapter will also contain a statement of the contribution that this work has made in the research field.

The closing sections of the dissertation include a bibliography of referenced publications, and an index of terms.

Chapter 2

Background

This background chapter is to present an overview of the state of research and practice in software reuse, software development and DSMLs, illustrated in Figure 1.1, and outlined in Sections 1 and 1.2. In addition this chapter documents other works that support the thesis of this research. This chapter also introduces the main terms and concepts.

2.1 Language Definition

Various definitions of the concepts *language* and *DSML* are used in the software community. The approach taken in formulating a definition for DSML is to first present varied definitions for the term *language* so as to identify the key features of this concept. This will then lead to a definition of the concept *modeling language* that reflects on the earlier definition of the concept *language*. Finally a definition for *domain-specific modeling language* that supports the basis of the research will be presented.

The term *language* as defined in the Webster dictionary[72], is: :

> "*... a special set of symbols, letters, numerals, rules, etc., used for the transmission of information.*" [72]

A description of *language*, from the linguistic community, is based on the fundamental assumption that all languages have universal features [8]. The features of interest in this work are:

- every language has a dictionary (*lexicon*) of words and sentences, and their associated meanings, and
- every language has a grammar, that includes rules for the formation of words and sentences,

Within the discipline of computer science the definition of language is given in terms of a *simple grammar* [87]:

> Let $\mathbf{G} = \{\mathbf{V}, \Sigma, \mathbf{P}, \mathbf{S}\}$ be a simple grammar. Where
> $\mathbf{V}$ is a finite set of variables,
> Σ is a finite set of terminal symbols,
> $\mathbf{S}$ is a distinguished element called the start symbol,
> $\mathbf{P}$ is a finite set of production rules, and
> $\mathbf{V} \cap \Sigma = \emptyset$. Then
> the simple *language* of $\mathbf{G}$, denoted $\mathbf{L(G)}$, is the set $\{w \in \Sigma \mid \mathbf{S} \stackrel{*}{\Rightarrow} w\}$

In David Schmidt's work on programming language development [84] he defines a *programming* language as having three characteristics:

- *Syntax*, which defines the appearance and structure of the language constructs,
- *Semantics*, which assigns meaning to the language constructs, and
- *Pragmatics*, which specifies the usage of the language.

This notion of language is also presented in Pratt's work on programming languages [78].

The community of computer science recognizes the significance and importance of languages that are not textual based, and have identified the field of *visual languages* (VL) as a major area of study[1]. Some of the areas that VLs have been applied are in the design of: (1) digital circuits [92], (2) building structures [79], and (3) parameterized devices [75]. The specification of visual (iconic) languages is considered to be the same as for a textual language, but

> "*... whose basic expressions are small pictorial signs that refer to the elements they visually depict*" [46]

The above definitions all contain the idea of a language consisting of: (1) a set of language constructs (*symbols, words, sentences, expressions*); (2) rules (*grammar*) for composing these constructs; and (3) the meanings of the language constructs. The notion of language used in this work refers to the language constructs and rules for composing the constructs as the *syntax* of the language, and the meanings associated with the language constructs as the *semantics* of the language.

---

[1]See *Journal of Visual Languages and Computing.*

2.1.1 Language syntax

All languages have a *notation* that is used to express the constructs of the language. This notation may be textual (for textual languages), or graphical (for VLs). For textual languages the constructs may be sets of words, strings, phrases, or sentences. For visual languages the constructs may be sets of icons, pictures, drawings, etc. These sets of constructs form the *syntactic notation* of the language [46]. An example of a syntactic notation, for a simple textual arithmetic language, is: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ÷, ×, +, -}. Similarly, an example of a syntactic notation, for a visual UML state chart [43] language, is: {◎ ○ ⟵ ⟶}. The syntactic notation is used to define the *concrete syntax* of the language.

Along with its syntactic notation, a language must include rules for combining and using the constructs of the syntactic notation [46]. Such a set of rules defines the *grammar* of the language. In defining these grammatical rules it is necessary to specify the *syntactic domains* for the notation elements..

The syntactic domain of a language notation is a set of values with a common syntactic structure [84]. For example, the numerical values of the arithmetic expression are from the syntactic domain of **digits**. Similarly the 'signs' of the language is from the syntactic domain **operator**. An example of a grammatical rule for the arithmetic language is given in BNF (Backus Naur Form) notation [71], by ⟨**expression**⟩ ::= ⟨**numeral**⟩⟨**operator**⟩⟨**numeral**⟩. The complete language grammar, expressed in BNF is:

⟨**expression**⟩	::=	⟨**numeral**⟩ | (⟨**expression**⟩) | − ⟨**expression**⟩ | ⟨**expression**⟩⟨**operator**⟩⟨**expression**⟩
⟨**numeral**⟩	::=	⟨**digit**⟩ | ⟨**digit**⟩⟨**numeral**⟩
⟨**digit**⟩	::=	0|1|2|3|4|5|6|7|8|9
⟨**operator**⟩	::=	÷| × | + |−

where: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ÷, ×, +, and - are terminal symbols, and ⟨**digit**⟩, ⟨**expression**⟩, ⟨**numeral**⟩, and ⟨**operator**⟩ are non-terminal symbols.

The grammar provides an algorithm for deriving legal expressions of the language, but the process of defining a language requires a *description* of the language.

2.1.1.1 Abstract syntax

This description of the *structure* of a language is usually in a form that excludes use of the *terminal symbols* of the language. Terminal symbols are the symbols (letters, words, icons) that appear in actual expressions of the language, i.e. the elements of the syntactic notation. Such descriptions are identified as the *abstract syntax* of the language [84]. The abstract syntax of the example arithmetic language is given as:

⟨**expression**⟩	::=	⟨**numeral**⟩ | *left_paran*⟨**expression**⟩*right_paran* | *minus*⟨**expression**⟩ | ⟨**expression**⟩⟨**operator**⟩⟨**expression**⟩
⟨**numeral**⟩	::=	*zero* | *one* | *two* | *three* | *four* |...| *fifty* | *fiftyone* |...
⟨**operator**⟩	::=	*div* | *times* | *plus* | *minus*

While this approach works fairly well for textual languages, it is not as straight forward to define abstract syntax for VLs. The abstract syntax for a textual language contains structuring information about the language, e.g. with the exception of the *minus*⟨**expression**⟩, all operators must be preceded and followed by a *numeral*. It is difficult to reflect structuring information about concrete VL constructs in their corresponding abstract syntax. An example of the abstract syntax of a VL is illustrated in Figure 2.1. This diagram has been reproduced from the UML specification [43], and shows the abstract syntax of the *Core Package - Dependency*. Figure 2.1 depicts: (1) the concepts *Usage*, *Permission*, *Binding*, and *Abstraction* as specializations of *Dependency*, (2) *Dependency* as a specialization of *Relationship*, and (3) the relationship between *Dependency* and *ModelElement*. The syntactic information presented in this diagram is that a *Dependency* (be it a *Usage* or a *Permission*, or a *Binding*, or an *Abstraction* establishes an association between *ModelElements*, such that on one end of the *Dependency* the *ModelElement* plays the *role* of a *public visible client*, and on the opposite end of the same *Dependency* the *ModelElement* plays the *role* of a *public visible supplier*. Additionally the multiplicity on the associations, of Figure 2.1 specifies that dependencies must be associated with *ModelElement*. A set of UML rules - the well-formedness rules (WFR) are included to completely specify the structural definition contained in Figure 2.1.

At this point it is useful to introduce the term $meta-model$, and state its relationship to the term abstract syntax. This is necessary because of the possibility of confusion in

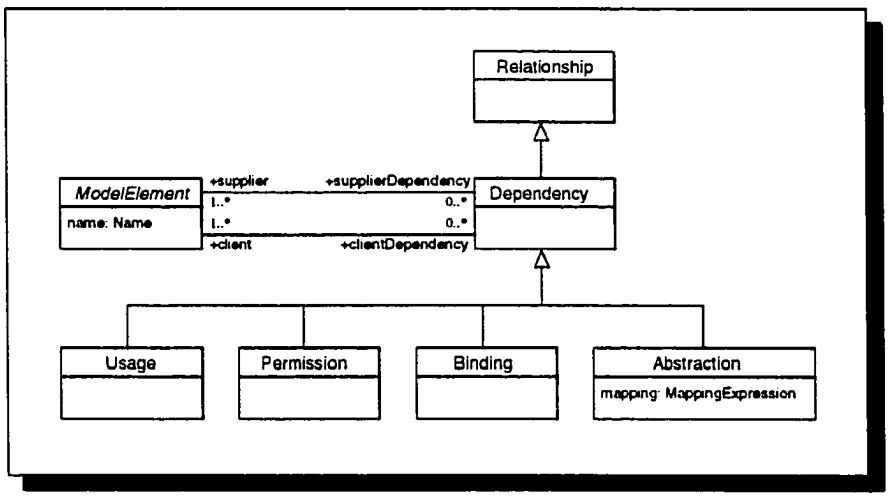

Figure 2.1: UML Core Package - Dependency abstract syntax [43]

the usage of both terms. The UML views meta-model as: (1) an abstract syntax, (2) well-formedness rules (WFR), and (3) semantics [43]. In Section 2.1.1 it is shown that the syntax of the UML is given my its abstract syntax and well-formedness rules. Thus it can be deduced that the UML meta-model is the UML language. In this work a slightly different interpretation of meta-model is taken - Meta-model is viewed as the syntactic graphical representation of DSMLs. In addition to the syntax of a DSML being captured in the meta-model it is also specified in the stereotypes of the DSML textually.

2.1.2 Language semantics

Whereas language abstract syntax deals with the structure of the language constructs, *language semantics* deals with the *meanings* ascribed to the language constructs. An understanding of the semantics of any language is necessary for the purpose of being able to define, use, and modify the language [22]. Though extensive work has been done on the definition of semantics for textual languages, a general framework for specifying VL semantics is not available. There are a variety of semantic formalism (denotational [84], structured operational [90], action [44], and algebras [7]) that have been developed for textual languages over many decades, but it is only recently that research has focused on formal semantics for VL [22]. Martin Erwig [22] states that a possible reason for this

situation is the absence of an equivalent notion of *abstract syntax* for VLs.

Though this lack of abstract syntax for VLs may be true in general, within the VL sub-domain of *software modeling languages* the modeling notation in use (the UML) is undergoing standardization. This standardized modeling notation, along with its abstract syntax provides the basis for research on the definition of semantics for VLs [24, 42, 81].

Similar to the identification of a syntactic domain for the definition of the structural features of a language, the identification of a *semantic domain* is crucial to the definition of the semantic features of a language. A semantic domain is a set of concepts whose *meaning* is known and well understood, and for which there is a *mapping* from the syntactic constructs of the language to the concepts of the semantic domain [81]. Consequently, the view taken herein is that the semantics of a language is comprised of a semantic domain and a mapping from the syntax of the language to the semantic domain. This is the *denotational* approach to semantic definition [84], and is also used in the Meta-Modeling Facility (MMF) [13] approach to engineering object-oriented design languages. An example of this mapping between the syntax of a language and the semantic domain of the language is shown in Figure 2.2

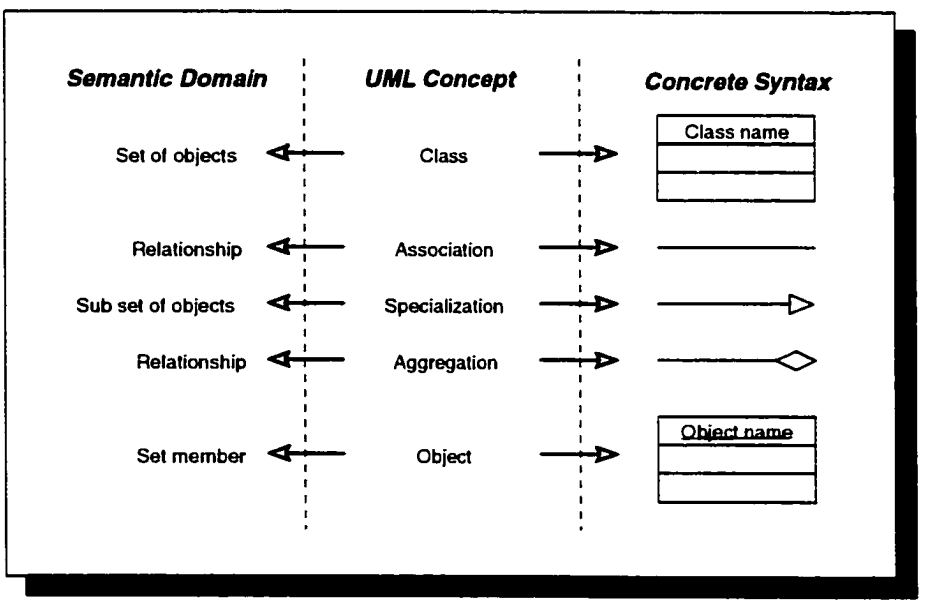

Figure 2.2: Semantic domain mapping

Figure 2.2 illustrates the mapping for a few UML constructs from the syntactic representation to the semantic domain. There can be more than one semantic domain for the syntax of a language. For the purpose of this work a language is defined as:

- *syntax*, which includes the notation of the language, and the rules for composing compound constructs from the basic constructs of the language, and
- *semantics*, which represent the meanings of the basic and compound constructs of the language.

This definition is general enough to be applicable to both textual and visual languages. In the following section this definition will be refined so as to be specific to software *modeling languages*, and then defined more specifically for software *domain-specific modeling languages*.

In this research work the attempt to define precise (formal) semantics for DSML constructs is **limited** to the precision offered in the UML specification in conjunction with any additional mapping (denotation) from the model elements to formal concepts in a semantic domain. Because of this limitation the semantic definitions for DSML model elements may not be in a one-to-one conformance with the specified textual specification of the elements meaning. This is because it may not be possible to map all textually specified *meaning* to a formal concept. Hence the precise (formal) semantics may be supplemented with textual description. Additionally, the UML and mapped semantics may be further constrained, but the mapped semantics and constraints cannot violate the UML specification semantics. The practice of supplementing the formal semantics with textual description is currently how the UML semantics is defined.

In this work the following views on semantic definition for DSML are held:

- The UML *class* and it associated *objects* concepts are the fundamental modeling concept used in all UML models of DSML.
- A DSML *class* is a *set* whose members are termed *objects* and these *objects* are characterized by having unique identifiers (a requirement of set membership).
- A *universal* DSML class characterizes all objects in the universe, and all classes defined in a DSML are sub-classes of the *universal* DSML class.

2.2 Domain-Specific Modeling Language

Domain-specific modeling languages (DSMLs) are a subset of software visual modeling languages that are characterized by having a vocabulary of terms and concepts that are fundamental to the problem and solution domains under consideration. Thus, DSMLs are identical, in definition, to that of software modeling languages, but with the added feature that the terms and concepts of the language have little applicability outside the specific domain.

The following sections (2.2.1, and 2.2.2) presents overviews of two DSML [70, 89]. From the overviews inferences will be drawn as to how closely these DSLs are in-step with the definition for DSML that has been accepted in this research. In addition, a summary of the benefits, requirements, and development issues associated with using DSMLs will be presented.

It must be again noted (as in Section 1.3) that the term *domain-specific modeling language* is explicitly uses herein, while the reviewed literature and previous work may use the term *domain-specific language*.

2.2.1 Jargon for domain engineering

Lloyd Nakatani [70] proposes the use of *jargons* in place of formal DSMLs because of their: (1) ease of definition, (2) composability, and (3) multi-purpose modeling capability.

The Webster's New World College Dictionary [72] defines the word *jargon* as:

> "*The specialized vocabulary and idioms of those in the same work, profession, etc., as of sportswriters or social workers...*"[2]

Though the word (jargon) is not explicitly defined in [70] the above definition aptly describes its usage in Nakatani's *et al.* work. The authors describe how their work with the *Family-oriented Abstraction, Specification, and Translation* (FAST) [15] process precipitated the invention of *jargons*. *Jargon* was invented because of the need to overcome the problem of

[2]This is not the only definitions for the word *jargon* in [72].

the long development time for producing a DSL, which caused a bottleneck in the overall software development cycle. The authors had worked extensively with the FAST process, and found that the time and skill that is required to build DSLs in the FAST process was not productive. Consequently they developed *jargons*, and have successfully incorporated this new form of DSL into the FAST process. A detailed description of the FAST process may be obtained from [15, 96]. Suffice to say at this point that FAST is made up of two main processes as itemized below and presented in Figure 2.3:

1. *Domain Engineering* (where software family members are analyzed to determine their commonalities), and
2. *Application Engineering* (where a model of the software family members is specified in terms of one or more *jargons*).

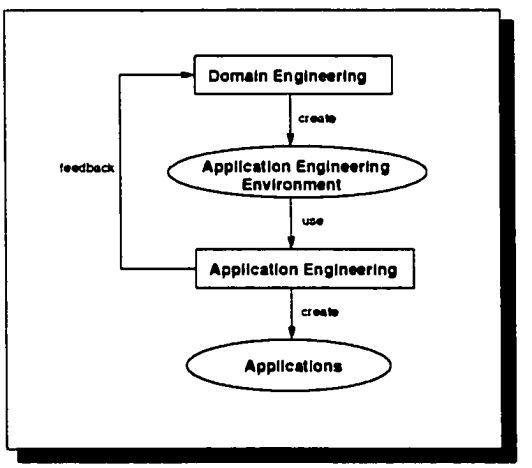

Figure 2.3: The FAST process [15]

Jargons are implemented from a single abstract syntax whose form is given as:

$$;term(note_1 \mid note_2 \mid \ldots \mid note_n)[memo]$$

where: (1) ';', '|', '(', ')', '[', and ']' are *jargon* meta-characters,
(2) *term* is the name of the *jargon* expression, and is user defined,
(3) *memo* is the information that is the focus of the expression, and
(4) $note_i$ $(i = 1 \ldots n)$ are either attributes of the *memo*, or parameters used to control evaluation of the expression.

This abstract syntax (*WizTalk*) is but one component in the application engineering environment (*InfoWiz*). The other components are, the *InfoWiz* generic interpreter, the

Fit programming language, and the API functions. The *WizTalk* abstract syntax is used to model document formats, algorithms, and hierarchically structured information. The *InfoWiz* generic interpreter is used to process any *jargon*, which has been customized with a semantics of the user defined *jargon*.

The semantics for the *jargon* is defined by a set of *actions*, which has a one-to-one relationship with the expressions of the specific *jargon*. The file that contains the actions is called a *wizer*, and a number of these files may be loaded by the interpreter in a single execution. The interpreter processes a *jargon* model by parsing it in a top-down, left-to-right, depth-first order; and executes the action corresponding to the expression at each node of the parse tree. The API functions allows the *actions* to interface with the *InfoWiz* interpreter.

The *Fit* programming language is used to write *actions* with the support of the API function library. The *Fit* programming language, and *InfoWiz* are proprietary properties of Lucent Technologies[3], but *Fit* is available in the public domain.

An example[4] of a *jargon*, an associated action, and their execution are given by:

---

[3]See Lucent's web-page at *http://www.pacelinetech.com*

[4]Taken from [70] with changes.

The *jargon* expression is given by:
`greeting[World]`

where: (1) **greeting** is the name of the *jargon* expression,
(2) `World` is the memo for the expression **greeting**, and
(3) the expression is held in the file `greeting.doc`.

The Fit function that defines the semantics of the *jargon* expression is given by:

```
A_greeting
    WizOut ''Hello to the whole '' GetWizMemo
```

where: (1) `A_greeting` identifies the *jargon* expression for the action,
(2) `WizOut` and `GetWizMemo` are *Fit* API library functions, and
(3) the action semantics is held in a *wizer* file `hello.w`

The line command to invoke the *InfoWiz* interpreter, which (1) accesses the action in the `hello.w` *wizer* file, (2) processes the contents of the `greeting.doc` *jargon* file, and (3) generates the output is:

```
localhost> wiz -w hello.w greeting.doc
```

and the output is:

```
localhost> Hello to the whole World
```

Jargon DSLs support the decomposition (divide-and-conquer) principle of software development. The domain under consideration is partitioned into two regions: *commonalities*, and *variabilities*. In the case study given in [70] the configuration control domain region of commonality is its reconfiguration algorithms, and the region of variability is its configurations; as determined in the FAST commonality analysis process.

In the concluding section of [70] Nakatani points out that the *InfoWiz jargon* concepts could be implemented with XML [47] instead of *WizTalk*, and Java [66] (or Perl [93] or Python [64]) used in-place of *Fit* for the host programming language.

2.2.2 A DSL for video device drivers

The second DSL reviewed is from the work of Scott Thibault *et al.*, [89]. Here the authors argue that a '*safer*' software architecture that is based on DSLs has as its primary aim, the achievement of applications in a shorter development time. DSLs produce safer applications because such applications are less error-prone than those developed using general

programming languages (GPL). This occurs because DSLs abstract key concepts, thus allowing the developer to focus on the issues that are pertinent at the particular phase of the development cycle. They identify increases in *flexibility*, *productivity*, and *reliability* as additional benefits that are realized from the usage of DSLs. But these benefits are offset by the difficult and costly task of building DSLs.

Thibault *et al.* [89] comments on two approaches to implementing DSLs, namely:

1. Those that are based on *compiler* technology, where there is a translation from the DSL to a target GPL. Such DSLs are difficult to write, require expert knowledge in language development, and are difficult to extend with new constructs or change existing ones. But on the positive side, these DSLs are very efficient.

2. Those that are based on *interpreter* technology are easy to write, require little knowledge of language development, are easier to extend and maintain. The negative side of these DSLs is that they are inefficient.

In order to take advantage of the best qualities of both DSL implementation technologies the authors propose a framework for the development of application generators that reconcile both approaches. Their framework relies on the program transformation technique, *partial evaluation*, which is suited to automatically transforming interpreters into compilers. An explanation of partial evaluation will not be presented here, but it is sufficient to say that it is a process by which known information about a programs input is exploited so as to evaluate sections of the program in advance (see [53] for details).

The framework proposed by Thibault *et al.* for the development of application generators is a two level (A and B) structure as presented in Figure 2.4 [89].

The lower level (**B**) is the definition of an abstract machine; "... *whose operations can be viewed as generic components that capture important operations of the domain.*" [89] The upper level (**A**) is made up of a definition for a micro-language (a DSL) in terms of the abstract machine, and this DSL provides an interface to the abstract machine. In this framework partial evaluation is applied at two points in the process, i.e. once at each level. Partial evaluation is applied to map a micro-program (DSL program) into an abstract

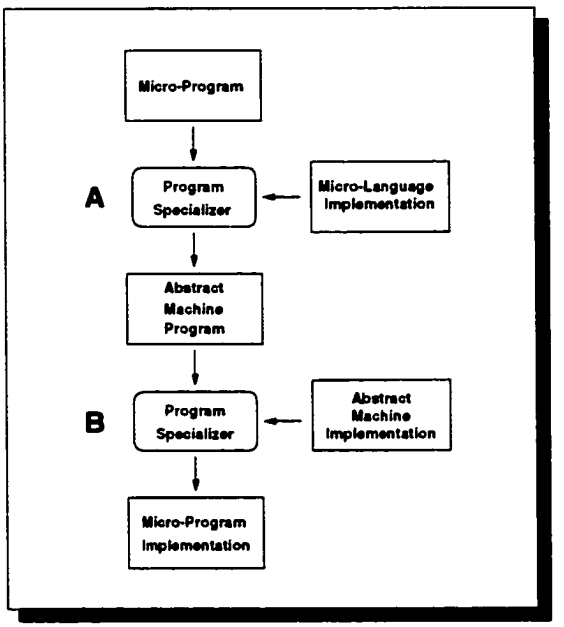

Figure 2.4: Generic program instantiation [89]

machine program. It is also used to map an abstract machine program into an efficient micro-program implementation.

The micro-language of Figure 2.4 is a "... *small restricted DSL which formally defines the program family ...* " [89] An instantiation of this program family is specified by a micro-program. The *program-specializer* (a partial evaluator) at level **A** accepts, as input, the micro-program and outputs an abstract machine program. The implementation of the abstract machine is a parameterized library of fundamental *operations* of the domain, with the parameters representing the variations of the operations. An abstract machine program has a set of parameters that satisfies the particular required functionality of the operation. The *program-specializer* (another partial evaluator) at level **B** eliminates the generality from the selected library function by using the supplied parameters. The result is the *micro-program implementation.*

The DSL (micro-language) captures the variations in *what is done* in the domain, and the abstract-machine captures the variations in *how things are done* in the domain. The program-specifiers (partial evaluators) provide a mapping from the generic programs of the

domain to a specific abstract machine program (level **A**), and a mapping from the generic abstract program to a specific micro-program implementation (level **B**). The DSL acts as an interface to the abstract machine library, i.e. DSL constructs reference the functions of the library.

The DSL designed for the *video device driver* domain is called *Graphical Adapter Language* (GAL), and the following paragraphs offer a brief description of: (1) the inputs to its creation; (2) its level of abstraction; (3) its level of restriction; and (4) definition of its variability, readability, and maintainability.

In the description of the definition of GAL the authors wrote about the variabilities of the domain, which were obtained from documentation on existing video cards, and inspection of video device driver code. One example of variability that was identified is that of the number of ways of communicating with the various cards studied. The commonalities of the domain are expressed, in GAL, as elements of the abstract machine program.

A second main input to the definition of a DSL is the domain knowledge of the abstract objects (concepts), and terminology used in the domain. In the definition of GAL the authors gained this domain knowledge from domain experts, and natural language application and system specifications. The authors identify this input as an important one, "*... because it leads to a more abstract user-level DSL.*" [89] Examples of domain concepts identified in this example are *clocks*, *ports*, and *registers*.

Once the domain concepts were listed, the attributes of these objects that relate to variabilities within the domain are then identified. From this, DSL declarative statements are defined to express values of those attributes that vary within the domain. Relationships between objects are translated into *reference relationships* in GAL. An example of this concept is given by *registers* being defined by reference to *port* definitions.

Though the authors were able to identify the patterns and define the abstract machine for the video-card device driver domain, relatively easy, they encountered some difficulty as a result of having to deal with code that was written by a number of programmers with different styles. Thus, it was necessary for the domain analysis process to be extremely detailed in determining similarities in functionalities of control structures examined.

Thibault *et al.* identified the provision of a high level of abstraction as an important goal behind the creation of a DSL, and this is their intent on focusing on the level of abstraction that can be achieved from the the abstract machine program in GAL. They set out to include information in GAL, which is used exclusively at the analysis level of system/application development. An example of this abstraction is demonstrated in their use of the concepts *fields* and *parameter* to eliminate the $low - level$ (implementation) procedural nature of of operations that execute *bitwise shift* and *logical operation.* In developing GAL the developers went through a series of iterative revisions of the language and the abstract machine program. They had to focus on the abstraction of the language during the many revisions so as to ensure that the *correspondence constraint* between the language and the abstract machine program was maintained.

An example of the application of this DSL , as taken from [89], is from the domain of video device drivers for video adapters. This domain supports twenty five different types of video cards, with over twenty four different models for each card. From the domain analysis it was determined that there are enough similarities between models of a video card to implement the drivers, for each card, from a generic program, but not for multiple cards. This limitation on the genericity of the application of the DSL is due chiefly to the wider variation between the different video cards.

The abstract machine for the video device driver domain was defined before work began on the associated DSL. The abstract machine was built from an analysis of the implementation of existing device drivers, and was refined during development of the DSL. Three patterns were identified from the existing drivers, and implemented in the abstract machine. These patterns are: *operational, combination of operational* and *control.*

The *operation pattern* corresponds to basic atomic operations of the domain abstract machine. These patterns appeared as either repeated fragments of code that differ in the type of data processed, or as fragments of code that execute the same type of operation but vary in the performance of the operation. Modules from the first instance exist in code module libraries or are defined as macros. An example of the second occurrence is identified in the frequently used code fragments that read and write data from and to the video-card.

There are a number of schemes for communicating with the cards and three such schemes were identified and captured in a single abstract machine as:

This input/output (I/O) instruction is parameterized by flags, which specifies the scheme that is to be used, i.e. indexed, paired, or PCI.

```
write_port(port_number:  integer,
           index:  integer,
           indexed:  boolean,
           pair:  boolean,
           pci:  boolean)
```

Reproduced from [89].

Combination of operation pattern is made up of combinations of operations that have no commonalities between the family members. One such example is the sequence of shift operations and logical expressions used on the video-card memory registers. The developers found no common set of combinations between the different device drivers, but were able to identify a set of operations from which any instance of these register operations could be constructed. Following is an example:

This register combination operation maps the value 'no' to an appropriate register in order to select the clock speed.

```
outb(0x3C2, (inb(0x3CC) & 0xF3) | ((no ≪ 2) & 0x0C));
outb(OTI_INDEX, OTI_MISC);
outw(OTI_INDEX, OTI_MISC | ((( inb(OTI_R_W) & 0xDF )
            ((no & 4) ≪ 3)) ≪ 8));
```

Reproduced from [89].

The *control pattern* is characterized by code fragments that exhibit a common control structure and contains combinations of previously identified domain operation patterns. Such control patterns may be viewed as framework structures as described in Section 7.5 of [82]. An example of a control pattern is given by:

This function saves, restores, or sets the clock value of the video-card.

```
Bool SelectClock(int clk, int val)
{
     switch (clk) {
     /* Save the clock value */
     case CLK_REG_SAVE:
        ...
        I/O and logic operations
        ...
        break;
     /* Restore the clock value */
     case CLK_REG_RESTORE:
        ...
        I/O and logic operations
        ...
        break;
     default:
     /* Set the clock value to 'val' */
        ...
        I/O and logic operations
        ...
     }
}
```

Reproduced from [89] with changes.

The authors introduce the concept of a *high−order* abstract machine operation, which takes in sequences of abstract machine operations as parameters. These parameters correspond to the fragments of code that make up the combination of abstract machine operation patterns. One such high-order abstract machine operation for the *SelectClock* operation is given by:

This *high−order* abstract machine operation corresponds to the abstract machine `SelectClock`.

```
clock_change(save_clk:  instructions,
             restore_clk:  instructions,
             set_clk:  instructions)
```

Reproduced from [89] with changes.

In the concluding section the authors outline the results of their work with the GAL do-

main specific language. They identify benefits of: (1) an increase in abstraction in using GAL; (2) the possibility of automated program analysis; (3) broad reuse; and (4) increased productivity; as being realized in the GAL application framework. The authors theorize that their approach to DSL development

> "*... would benefit from a generator-specific reuse method that would allow interpreters and abstract machines to be constructed from reuse composable parts.*" [89]

The following observations are based on the content of sections 2.2.1 and 2.2.2:

- Both DSLs are textual, and are easily defined, maintained, and translatable into program code.
- Both DSLs may be applied in a software development paradigm where the problem is decomposable into smaller units, such that solutions are easily defined. These small solutions are then composed to form a solution for the larger problem.
- Using both DSLs is much simpler than most other DSLs because of the absence of needed knowledge on language construction, and compiler technology.
- Both DSLs distinguish between the syntax (textual constructs) of the language and the language semantics, though the definitions of both aspects are not the focus of the respective works.
- Both DSLs are designed for application at the implementation phase of software development.
- Both lack a formal definition of their abstract syntax, because of the inclusion of terminal symbols (language tokens) in the given definitions.

The above observations are typical of most DSMLs that are currently in use. These DSMLs offer no support for system analysis and design, as they are chiefly intended to aid the implementation phase of system development, vis-á-vis abstracting away some of the repetitive programming tasks.

2.3 DSML Summary

A summary of DSML set the stage for the work that is introduced in the following chapters.

2.3.1 Requirements for DSML

1. The first requirement is that the domain of concern must be one that is *mature, well-defined*, and is comprised of a *usable* set of *current* and *future applications* [91]. This requirement is based on the necessity that the vocabulary, syntax, and semantics of DSL terms remain as stable as possible. If the domain is not mature and well-defined then the frequent introduction of new concepts would result in frequent and costly changes in the DSML. This would nullify benefits of DSML technology. The domain may consist of a set of current applications that will act as a *seed pool* for the building of the domain vocabulary, syntax, and semantics. There must also be a need for many future applications of the domain, without this need there is no point in defining a DSML for the domain.

2. A second requirement is that the DSML must encompass all known and relevant terms of the domain. Such terms are the names of all domain-specific concepts, objects, relationships, operations, functions, etc. that may appear in any domain application specification at the analysis, and design of the domain phases of development.

3. A third requirement for DSML is that it offers a level of abstraction for the domain concepts at the analysis and design stages. Domain abstraction is chiefly concerned with the mapping of the various external views of the domain applications to a set of functional representations [89]. The relationships that exist between the levels of abstractions can be defined and codified in the DSML.

4. A fourth requirement for DSML is that it should be scalable. With the passage of time there will be new user requirements or the domain may be widened to include features and concepts that were not a part of the initial set of requirements, features and concepts. The DSML for such domains must be able to be extended to accommodate these new requirements, features and concepts.

The work documented, herein, will use the UML: (1) notation, (2) model management concepts, and (3) extension mechanisms to define DSMLs that have a precise syntax and

semantics, and conform to a more strict set of requirements - as documented in Section 2.5.

2.3.2 Benefits of DSML

There are a number of functional and non-functional benefits that may be realized from the use of DSML technology in application/system development. In the framework of this research the following non-functional benefits may be realized:

- Increased *reliability*, and *maintainability* arising from the reuse of high-quality architectural and design patterns that are defined in the DSMLs.
- Increased *productivity*, arising from a reduction of the overall time to deliver a complete application/system.

The functional benefits that can be realized are:

- Increase in the level of abstraction of the application specification. A good DSML extracts out implementation details, and relies on transformation processing to go from the abstract DSML models to the concrete implementation model. In the context of this work the DSML technology will be applied at the analysis and design levels of system development.
- The domain knowledge incorporated into the DSML will be independent of any implementation programming language, i.e. it is up to a particular CASE tool to translate segments of the design DSML application specification into executable programming language code.

Though these lists of benefits are not exhaustive they are amongst the ones considered most significant in this research.

2.3.3 DSML development issues

Though DSML technology is not the "*silver bullet*" [54] it can reduce development cost (i.e. time and man-power) and increase reliability, maintainability, productivity, and verifiability; when used in a suitable domain, along with the appropriate support tools [73, 91]. But there are a number of questions that have to be answered with in applying this technology. A summary of a few of these questions that have to be answered are as follows:

1. *To what extent does the application engineer (AE) require DSML development knowledge?* It is theorized that the AE should not require any knowledge of DSML development technology. The AE's approach to using a DSML should be the same as that of the developer using any software modeling language.

2. *Can a DSML be presented as a graphical model?* Graphical models are widely used because of the ease of understanding (a picture is worth a thousand words), and offer more precision than textual descriptions. The research work is geared towards using a graphical notation for DSMLs, i.e. the UML [43]. Working in a CASE tool environment the AE will select domain terms and constructs that are represented as icons on a drawing palette. This simplifies the development process, as the syntactic checks will be done transparent to the AE.

3. *How can language extensibility be handled?* All changes to a DSML are carried out by a domain engineer, who is versed in DSML development technology. Figure 1.2 shows that feedback from domain application engineering activities is directed to the domain analysis and design process, which drives all changes to the language engineering activity.

4. *Can the DSML definition process be standardized, and thus be amenable to reuse?* The thesis of this research is that such a process can be standardized, and should address the three previously identified issues, which are related to the implementation of DSML technology. This standardization process will be presented in the form of UML profile description of DSMLs, along with a process for instantiating specific DSML models from the UML meta-models.

The solutions to the four issues of DSML development that have been listed above forms the four main deliverables herein. The research defines a UML-based model for the definition of DSMLs that is graphical, facilitates extension of the DSML in a precise manner, and permits the application engineer the ease of developing analyzable models.

2.4 Domain Analysis and Design

From an examination of the DSSEE, as illustrated in Figure 1.2, it is noted that the activity of *Domain Analysis & Design* (DA&D) is not an area that is studied in detail for this work, i.e. only the shaded icons (*domain models, domain language engineering, domain-specific rules,* and *domain-specific modeling language* are researched. Since two of these areas are output of DA&D, and one of these areas provide feedback to DA&D it is necessary to present some aspects of DAD activity. In the remainder of this report the phrase *domain analysis* will be used in place of *domain analysis & design.* This is done because the former phrase is generally used to describe both the activities of *analysis* of problem and *design* of solution. But in this work a distinction is made between the models obtained from both activities - hence *Domain Analysis & Design* in Figure 1.2.

2.4.1 Domain analysis and design definition

There are various published definitions for the term *domain analysis*, and a review of a few will lead to a composite definition that encapsulates the intended usage in this research. Guilermo Arango and Rubén Prieto-díaz define domain analysis as:

> "*... the process of identifying and organizing knowledge about some class of problems - the problem domain - to support the description and solution of these problems.*"[3]

Wilhelm Schäfer, Rubén Prieto-díaz, and Masao Matsumoto definition state that domain analysis is:

> "*... organizing descriptions of components and architectures common to the applications analyzed to those who adopt a systems taxonomy view on domains. ... from the notions of problem domain and reasoning task,* [it is] *the process of creating models that make it possible to reason about the domain, that is, to predict, to explain, or to derive facts about the domain that are difficult to observe directly.*"[83]

Kyo Kang, Shaolom Cohen, James Hess, William Novak and Spencer Peterson define domain analysis as:

> "*... the process of identifying, collecting, organizing and representing the relevant information in a domain, based upon the study of existing systems and*

their development histories, knowledge captured from domain experts, underlying theory and emerging technology within a domain." [56]

A forth definition, obtained from the Boeing, IBM and Unisys Technical Report for STARS [26] states that domain analysis is:

> "the *processes that analyze, abstract and model the characteristics of existing and envisioned application products within a domain in terms of what the products have in common.*" [26]

In the following sections 2.4.2 and 2.4.3 two industry created DA&D methodologies will be described. The purpose of this is identify the best set of sub-activities that support the above definition of a DA process for DSSEE.

2.4.2 Features-Oriented Domain Analysis of SEI

The *Features-Oriented Domain Analysis* (FODA) methodology was developed from an in-depth study of the applications of a number of successful domain analysis methodologies [2]. FODA focuses on the features of the applications in the domain, i.e. it views the domain as a collection of similar, existing or future applications. The features of FODA are in essence the user requirements as implemented for the various systems of the domain,

> "*...the feature-oriented concept of FODA is based on the emphasis the method places on identifying prominent or distinctive user-visible features within a class of related software systems.*"[4]

Arango [2] defines three phases to the FODA methodology, namely: *Context Analysis, Domain Modeling* and *Architecture Modeling.* While Krut and Zalman [4] specify only the first two phases in their report. The Krut and Zalman description is more comprehensive as Arango specifies the third phase, architecture modeling in his graphical notation, Hierarchical Task Analysis (HTA) [57], but fails to elaborate on this phase in either the HTA representation or the textual description.

The HTA of Figure 2.5 defines a *modified* structure of domain engineering in FODA. The *do entity-relationship modeling* of Arango's [2] original FODA HTA has been replace with *information analysis* (0.2.2) in Figure 2.5, and has been elaborated to specify three activities: *define entities* (0.2.2.1); *define objects* (0.2.2.2); and *define relationships* (0.2.2.3).

Also Arango's *Investigate functional and implementation constraints* had been renamed *constraint analysis* (0.2.3.4) in Figure 2.5. These changes were implemented from the descriptions obtained in Krut and Zalman [61].

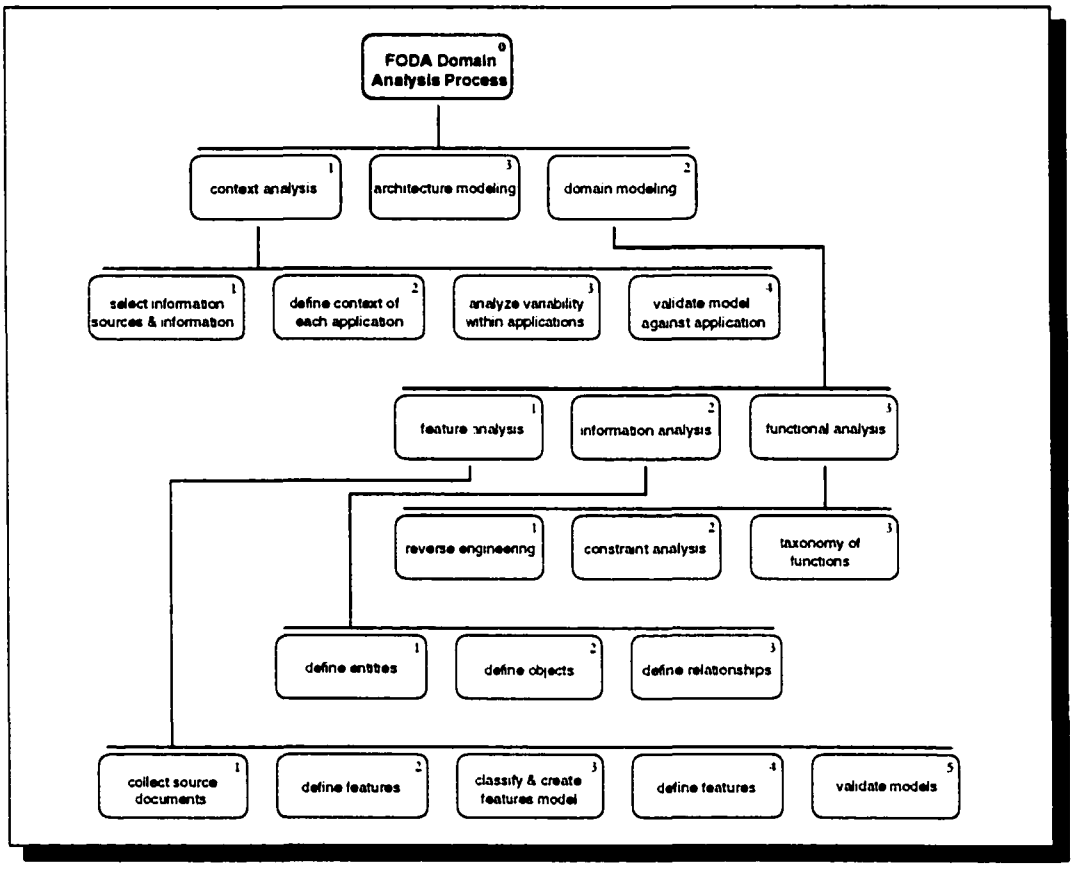

Figure 2.5: Features Oriented Domain Analysis (FODA) taxonomy

The HTA of Figure 2.5 provides no information on the inputs and outputs of the listed activities. This information is captured in Table 2.1, which is another amalgamation of the descriptions of FODA from Arango [2] and Krut and Zalman [61]. Here the phases of FODA are associated with their inputs, outputs and activities.

The outputs from the *context analysis phase* are the *context models* (*structure* and *context diagram*), and the *domain dictionary*. The *structure diagram* is an informal block diagram, which places the domain in a hierarchical mapping with its higher, lower and peer level

Table 2.1: FODA Inputs and Outputs

Phase	*Activity*	*Input*	*Output*
Context Analysis	context analysis	operating environment source document	context model structure diagram context diagram domain dictionary
Domain Modeling	feature analysis	features context diagram domain dictionary	features model context features operational features domain dictionary
	information analysis	application domain knowledge domain dictionary	information model entity relationship semantic net object domain dictionary
	functional analysis	domain technology context model features model entity relationship model domain dictionary requirements	functional model

domains. The *context* diagram is a data flow diagram that shows the communication between the domain and its external entities. Thus the context analysis defines the complete scope of the domain both in terms of its static and operational dependencies. The *domain dictionary* is a repository of all terms specific to the domain and this repository grows with the execution of each activity in the DA methodology.

The *domain-modeling phase* of the FODA paradigm comprises three activities, namely *feature, information* and *functional analysis*. The *features model*, output from the features analysis activity, is made up of *contextual* and *operational representations*. The updated domain dictionary is also an output of the features analysis sub-phase of FODA.

The *context features* describe the ways in which applications within the domain may be used. Context representation addresses issues of performance, requirements, accuracy and time synchronization [4]. The *context features* of the domain modeling phase of FODA focuses on the representation of applications within the domain as distinct from the context features at the context analysis phase, (which focuses on the interface between the domain and entities external to the domain). *Operational features* describe how an application of the domain is executed. These operational features are equivalent to the defined functions of the application. The operational features exist as components of applications of the

domain and are represented as operations in the operational model.

The outputs from the *information analysis* activity of the FODA domain modeling phase are the *information model* and the updated domain dictionary. The *information model* is made up of *object* or *entity relationship* models or *semantic nets* [4]. The tasks of information analysis are to capture, define, and represent domain data requirements and the associated domain knowledge. This activity is centered on data abstraction and data decomposition. The representation of this domain knowledge may take the form of one or a combination of the information models.

The activities of the *functional analysis* of the domain modeling phase of FODA are the abstraction and structuring of the common functions defined within the domain. Finally these functions are sequenced into an operational model [4]. While there is no specifically stated representation of the functional model, it is suffice to say the representation must capture

> "... *both the behavioral and functional relationship between the objects in the information model and the features in the features model*... [4]

Though there is no stated representation for the domain dictionary, Krut and Zalman [4] defines the function of the domain dictionary to be a tool for application developers and users to locate:

- definitions of domain terms,
- synonyms of domain terms,
- sources of domain terms, and
- hierarchical abstraction and decomposition of domain terms, definitions and features.

The contents of the domain dictionary are obtained from requirements documentation, legacy systems of the domain, user specifications and domain experts.

2.4.3 Commonality analysis of FAST

Commonality Analysis (CA) is a strategy for developing software systems that came from an industrial demand for rapid production of reliable systems. This strategy is based on three hypothesis [96], namely:

1. *The Redevelopment Hypothesis* - asserts that most software development consists of the creation of variations of already existing systems. The off shoot of this is that these systems have more in common with each other than there are differences.

2. *The Oracle Hypothesis* - assumes that it is possible to predict the changes that a system will undergo during its lifetime, i.e. system evolution is predictable.

3. *The Organizational Hypothesis* - The predictable changes that a system will under go (Oracle hypothesis) can be exploited so as to obtain an advantage in the system maintenance through its organization. This is accomplished by structuring the system in modules where changes can be implemented with minimum impact.

These hypothesis are set out in [96] from which the information for the FAST HTA (illustrated in Figure 2.6) was obtained, and also from which the overview of the CA activities and processes are taken with additional descriptions from [95]. The thick-line bold-text branch, illustrated in Figure 2.6, identifies the CA activities.

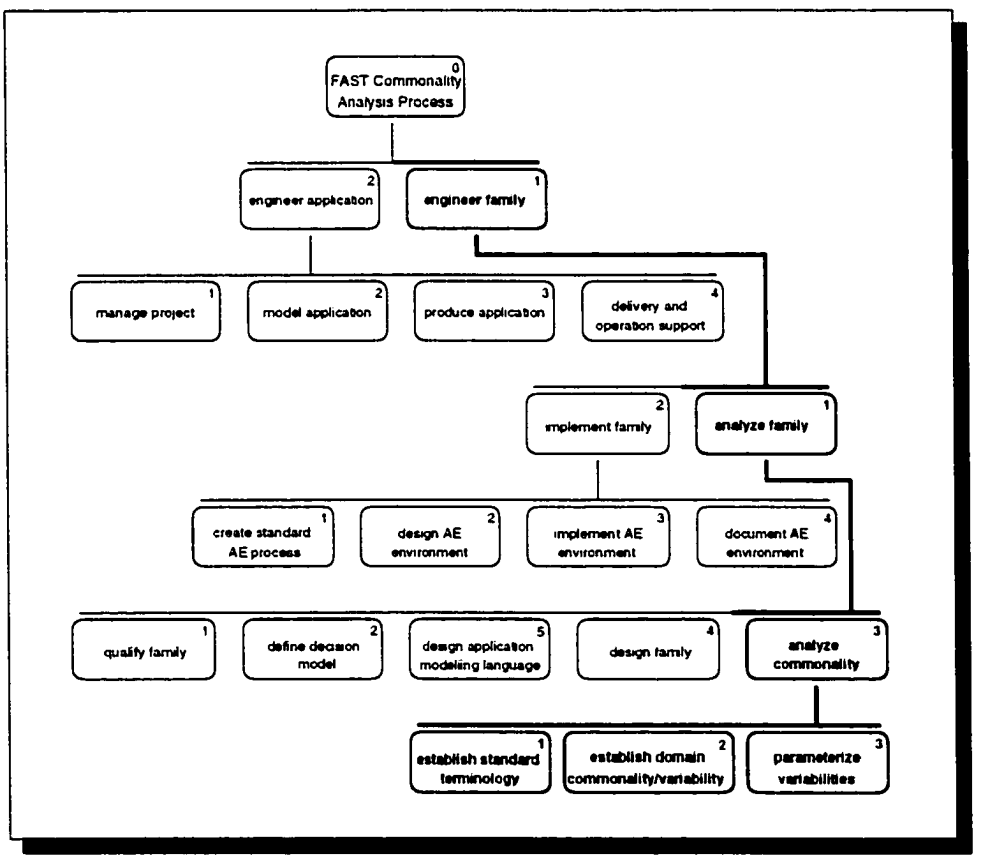

Figure 2.6: FAST taxonomy with the Commonality Analysis

The goal of CA is to be able to define a family of applications based on the commonality and variations in this family, and the success of this approach is dependent on the degree to which the developer is able to predict the type of changes (variabilities) in the family. CA is a sub-process of the FAST methodology. The process for conducting CA, in FAST, has been reproduced in Figure 2.7. The leaf sub-activities of **engineer family** (task 0.1), namely **establish standard terminology** (task 0.1.2.3.1), **establish domain commonality/variability** (task 0.1.2.3.2), and **parameterize variabilities** (task 0.1.2.3.3) in Figure 2.6 are conducted in the **Analyze** process of Figure 2.7.

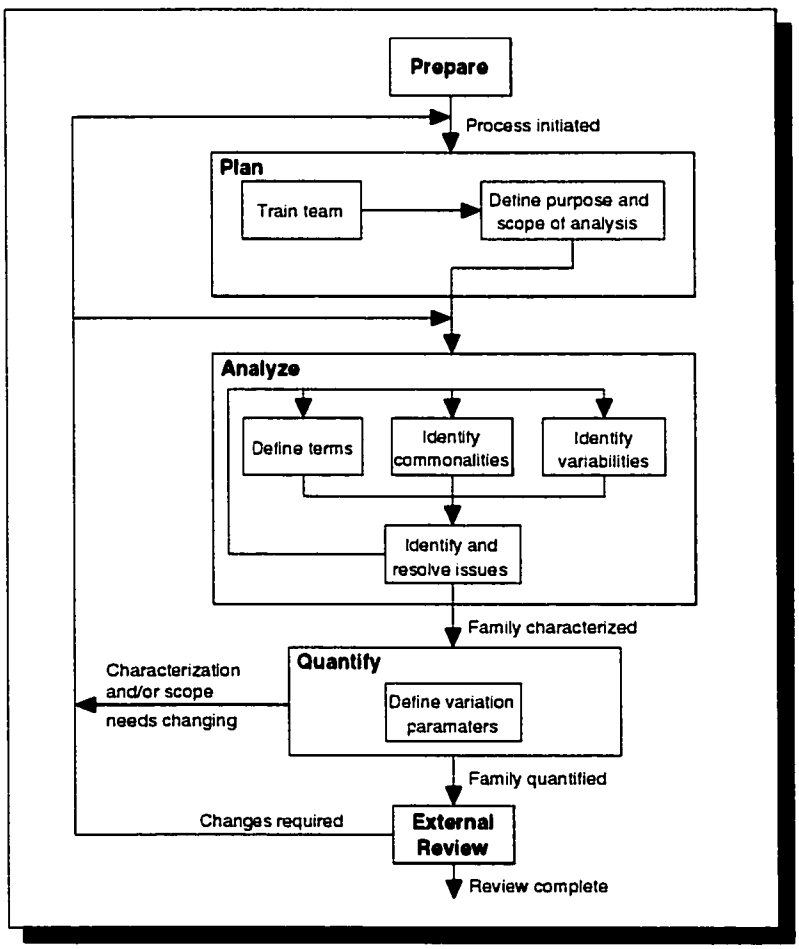

Figure 2.7: Commonality Analysis (CA) Process [95]

In CA the process of defining terms and identifying commonalities are closely related, as one can start with either of these tasks and segue to the other in a logical sequence. This

process is usually conducted in an iterative manner, i.e. the identification of a *common* activity leads to a definition of the technical terms listed in the narrative of that activity; or the defining of a technical *term* leads to the identification of a common usage or occurrence of the term. CA requires that the output of these two processes be in an agreed standard format. This requirement facilitates the systematic identification of the variabilities in the domain terms and activities.

The process of identifying variabilities involves a detailed review of the previously listed commonalities to determine whether: (1) a previous assumption actually holds true for all member of the family, and (2) there is the possibility of a common assumption changing in the future. Similar to that done for domain terms and commonalities, the domain variabilities are documented in an agreed standard format. Implicit in this formating process is the requirement for the structuring of the identified domain artifacts, i.e. the commonalities and variabilities have to be organized "*... into groups that deal with particular aspects of the domain*" [96].

In concluding this overview it must be emphasized that the outlined CA process does not identity the form of the representation of the domain terms, commonalities or variabilities identified in the process. Implicitly, there is the notion that a textual representation plays a significant part in the documentation of these artifacts of commonality analysis in FAST. Thus it is entirely left to the CA team to decide on the type of representation that will be used to record the outputs of the CA process.

2.4.4 Domain analysis and design in DSSEE

From the four definitions of domain analysis and design (DA&D) in Section 2.4.1 a composite definition for DA&D is presented that captures the key aspects mentioned in the four, and supports the intent of DA&D in the DSSEE. Thus, DA&D as specified in this work and illustrated in Figure 1.2 is defined as:

> **The process of identifying, collecting, organizing, analyzing, and representing in the form of models relevant, common, and variable knowledge about some class of problems in a domain in support of the descriptions and solutions of these problems.**

The DA&D activity used in DSSEE is a set of tasks and products derived form the FODA [56] and FAST CA [95, 96], as outlined in Sections 2.4.2 and 2.4.3, respectively. In DSSEE the FODA *architecture modeling* (0.3 of Figure 2.5) is not used, hence there will be no further mention of this sub-task. Also, the *Plan* activity of CA, as illustrated in Figure 2.7, is not discussed herein.

The *content analysis* (0.1) and *domain modeling* (0.2) tasks of FODA, along with *Analyze* and *Quantify* activities of CA (Figure 2.7) constitute the DA&A activity of DSSEE. In DSSEE *Feature Analysis* of FODA (0.2.1) and *Define terms* of CA are focused on the identification of the static and dynamic features/concepts of the domain. IN DSSEE the inputs to this activity are system requirements (legacy, user, and future as illustrated in Figure 1.2). *Legacy requirements* are those that are obtained from the documentation of existing systems in the domain. *User requirements* are those that have been defined for systems that are currently under consideration for development, and/or are in the process of being developed. *Future requirements* are those that are envisaged for systems that may be desired in the future.

Itemized below is a description of the domain analysis activities and products taken from FODA and CA that constitute DA&D in DSSEE:

- *Domain dictionary* - Both FODA and CA identify a domain dictionary as an important artifact of the DA process. The dictionary is intended to be a repository of domain terms, abbreviations, and definitions to be used by domain analysts and application developers. This artifact also facilitates communication between the various stakeholders in the domain and application development processes. Neither FODA nor CA specifies a specific format for the domain dictionary. In the UML a domain dictionary would be a type of *Artifact* as specified in Section 2.5.2.2 and would be a classified as a stereotyped $\ll$ *document* $\gg$, which is a sub-class of the stereotype $\ll$ *file* $\gg$. In Section 3.2.1.1 a format for the domain dictionary, as used in this work, will be presented.

- *Static concept model - Information analysis* (0.2.2 in the FODA HTA of Figure 2.5) in DSSEE focuses on the capturing of domain information in the form of a set of models. This activity results in the creation of a domain static concept model (SCM) that captures the commonalities and variabilities in the domain from a static perspective: This activity is specified in Commonality Analysis (CA) as *Identify commonalities* and *Identify variabilities* (see Figure 2.7).

- *Dynamic concept model - Taxonomy of functions* (0.2.3.3 in the FODA HTA of Figure 2.5) in DSSEE focuses on the capturing of the domain's dynamic (behavioral) concepts. This activity results in the creation of a domain dynamic concept model (DCM) that captures the commonalities and variabilities in the domain from a behavioral perspective. This activity is specified in CA as *Identify commonalities Identify variabilities* (see Figure 2.7).

The above bulletted models are directly associated with *domain analysis* of the problem space. The models associated with the *design* in the solution space are the set identified in Section 1.3.1, and are as defined in the UML specification. In this work there is no intent to define a development methodology, as any methodology that facilitates the creation of the domain models identified as inputs to the language definition activity will suffice. In Section 3.2.1.1, 3.2.1.2, and 3.2.1.3 examples of these models will be illustrated.

2.5 Unified Modeling Language

The *Unified Modeling Language* (UML) [43] is a visual language that is intended to be used for the modeling and the documenting of software systems' artifacts that are chiefly object-oriented in nature. The UML is partitioned into three main sub-groupings, (packages) as illustrated in Figure 2.8.

The UML specification [43] list seven design goals of the UML, and the following six are fundamental to this work:

1. "*Provide users with a ready-to-use expressive visual modeling language to develop and exchange meaningful models.*
2. "*Furnish extensibility and specialization mechanisms to extend the core concepts.*

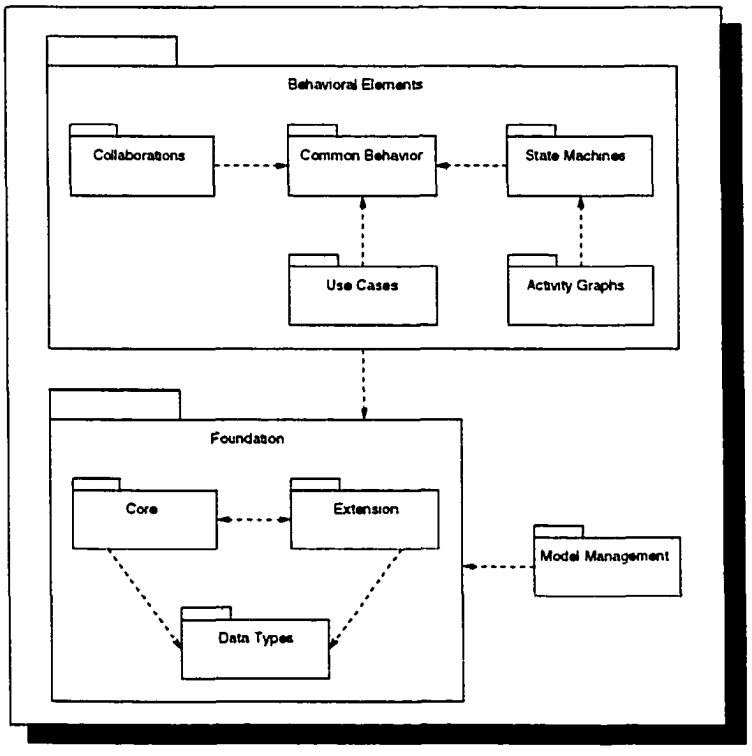

Figure 2.8: UML structure [43]

3. "*Support specification that are independent of particular programming languages and development processes.*

4. "*Provide a formal basis for understanding the modeling language.*

5. "*Support higher-level development concepts such as components, collaborations, frameworks and patterns.*

6. "*Integrate best practices.*" [43]

The UML has been primarily used as a notation for modeling single applications, and thus it is not surprising that its usage to model families of application is difficult. Despite this difficulty, the UML has been used to model reusable experience in the form of patterns and architectures (e.g. $http://st-www.cs.uiuc.edu/users/patterns/patterns.html$), but the resulting models are more representations of typical application examples, rather than true representations of product families. The UML *profile* concept is aimed at representing *dialects* of the UML language. These dialects are intended to be *enterprise-*, *project-*, or *method- specific* extensions of existing UML meta-model elements [45].

In the following sections 2.5.1, 2.5.2, 2.5.3, 2.5.4, 2.5.5, 2.5.5.2, and 2.5.7 reviews of UML concepts and features that will be used in the proof of the theory of this work are presented. The goal of this exercise is to identify, and highlight the properties of these concepts and features that are crucial to the research work.

2.5.1 Meta-Modeling

The general purpose for modeling, in any discipline, is multi-fold. some of these general purposes are:

- the ability to communicate amongst a team about a particular problem and its solution,
- document and state the boundary of some problem or class of problems, and the corresponding solutions,
- to reduce the complexity of a class of problems and the corresponding solutions that are too large to be comprehended in its entirety, and
- to test or analyze some property of a problem or solution before constructing or implementing a full scale version.

Thus, it can be inferred that models provide a means to reduce cost (be it financial, manpower, or time), and predict results (be it performance, conformance or reliability) in a virtual manner. In researching *modeling* languages it is essential that one be able to talk about the structure of the language models, i.e. be able to *describe* the models. Such descriptions may be in the form of a textual description, or (recursively) in the form of a *model*. Use of the latter format (i.e. models to describe models) is termed *meta−modeling*. The Webster dictionary [72] provides a definition of the prefix *meta-*, which states:

> *"...going beyond or higher. transcending: used to form terms designating an area of study whose purpose is to examine the nature. assumptions, structure. etc. of a (specific) field"* [72].

The UML specification [43] defines the term *meta − model* as: *"A model that defines the language for expressing a model."*

In a similar manner one could describe the model that describes a model, as a meta-model of the meta-model - this could go on indefinitely! The UML specification [43] limits the description of the UML and its meta-models to four layers. The four layers are defined as:

Table 2.2: UML four layer meta-model architecture [43]

Layer	*Description*	*Example*
meta-meta-model (level M3)	The infrastructure for a meta-modeling architecture. Defines the language for specifying meta-models.	*MetaClass, MetaAttribute, MetaOperation*
meta-model (level M2)	An instance of a meta-model. Defines the language for specifying a model.	*Class, Attribute, Operation, Component*
model (level M1)	An instance of a meta-model. Defines a language to describe an information domain	*StockShare, askPrice, sellLimit, StockQuoteServer*
user data (level M0)	An instance of a model. Defines a specific information domain.	*<Acme_SW_Share_98789>, sell_limit*, 654.56

From Table 2.2 it is noted that: (1) a language that specifies the UML, i.e. a description of the UML resides at the **meta-meta-model** level (M3), (2) the UML, i.e. the abstract syntax, well-formedness rules, and semantics, are at the **meta-model** level (M2), (3) a UML model, such as a class diagram, activity diagram, etc., exists at the *model* level (M1), and (4) the entities of a model (such as the named objects and associations of a class diagram) exist at the **user-data** level (M0).

2.5.2 UML syntax

The UML syntax is given in the form of an *abstract syntax* and a set of *Well-Formedness Rules* (WFR). The abstract syntax presents a graphical description of the structure of the language, i.e. the grammar of the language. The abstract syntax is illustrated in the UML specification [43] as a set of UML class diagrams (CD), and the WFR as a set of Object Constraint Language (OCL) statements that are supplemented by textual descriptions. The classes of the abstract syntax CD are meta-classes, and attributes of these classes are meta-attributes. In order to review the UML syntax a UML abstract syntax is reproduced in Figure 2.9, along with a sub-set of its textual description (Table 2.3), and WFR (Table 2.4); these have been reproduced from the UML specification [43].

Figure 2.9 illustrates the five abstract syntax (meta-model) concrete elements (*Parameter, Constraint, Attribute, Operation*, and *Method*), with the other elements being abstract. Concrete elements are those that can be instantiated in a model of the meta-model, where as abstract elements (whose names are presented in italic font style) are chiefly used to organize, reify, or share structural information of the concrete elements.

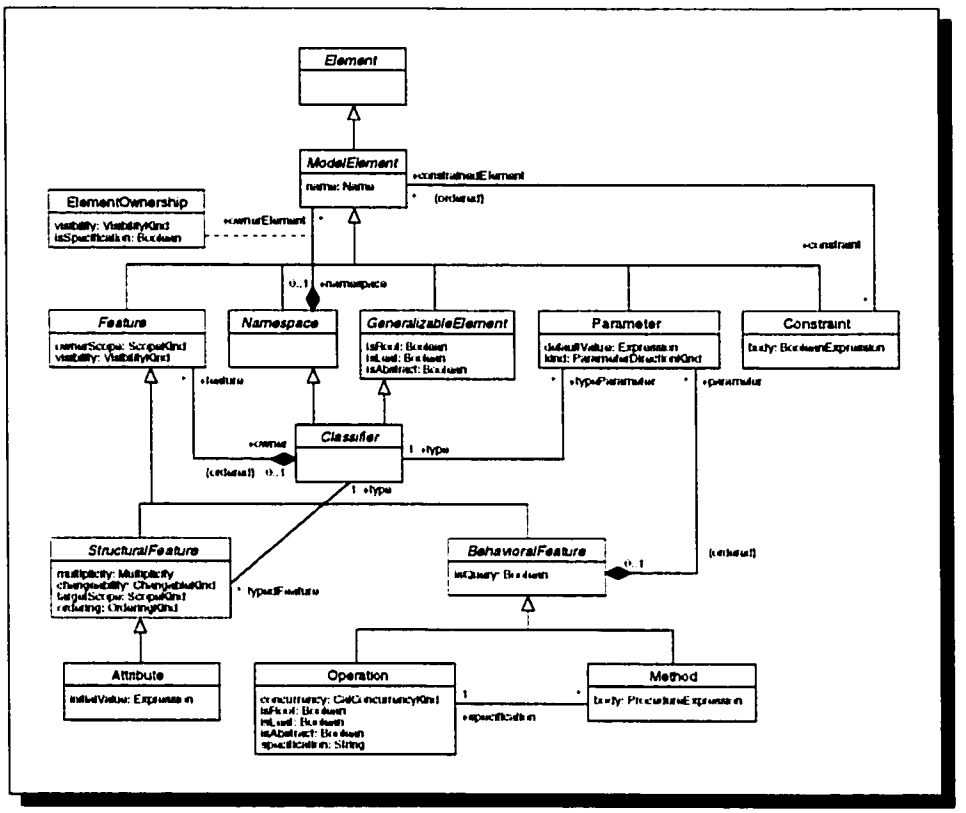

Figure 2.9: UML Core Package- Backbone abstract syntax [43]

The WFR descriptions of two meta-model elements (*Attribute* and *Parameter*) that are defined in this meta-model are presented in Table 2.3.

The abstract syntax (e.g. Figures 2.1 and 2.9) and the textual description of the abstract syntax (e.g. Table 2.3) constitute the *syntax* of the UML.

Harel *et al.* [46] views the UML WFR as a "*bootstrapping*" of the UML abstract syntax in preparation for the definition of the UML semantics. They see the WFR as constraining the syntax in such a manner as to eliminate the formation of problematic or invalid syntactic constructs, but not adding any meaning to the syntactic constructs. The UML takes a different view of the WFR. The UML terms the WFR as "*static semantics*" and defines it as being an instance of how a construct should be connected to other instances so as to be "*meaningful*". But in the opinion of this work '*to be meaningful*' does not define '*meaning*'.

Table 2.3: UML meta-model description [43]

Attribute

An attribute is a name slot within a classifier that describes a range of values that an instance of the classifier may hold.

In the meta-model, an Attribute is a named piece of the declared state of a Classifier, particularly the range of values that Instances of the Classifier may hold.

Attribute	
initialValue	An Expression specifying the value of the attribute at initialization. It is meant to be evaluated at the time the object is initialized. (Note that an explicit constructor may supersede an initial value.)

UML version 1.4 lists an association of Attribute as associationEnd. This seems to be an error, as the abstract syntax has no association end type related to Attribute. Further this association is defined in the description of StructureFeature. Its definition here is redundant. It is thus excluded from the description of Attribute in this work.

Parameter

A parameter is an unbounded variable that can be changed, passed or returned. A parameter may include a name, type, and direction of communication. Parameters are used in the specification of operations, messages and events, templates, etc.

In the meta-model, a Parameter is a declaration of an argument to be passed to, or returned from, an Operation, a Signal, etc.

Attribute	
defaultValue	An Expression whose evaluation yields a value to be used when no argument is supplied for the Parameter.
kind	Specifies what kind of a Parameter is required: Possibilities are: • *in* - An input Parameter (may not be modified). • *out* - An output Parameter (may be modified to communicate information to the caller. • *inout* - An input Parameter that may be modified. • *return* - A return value of a call.
name	(Inherited from ModelElement) The name of the Parameter, which must be unique within its containing Parameter list.
Associations	
type	Designates a Classifier to which an argument value must conform

The approach taken in this work, on the definition of semantics for domain-specific model elements, may be described as an amalgamation of both that proposed by Harel *et al.* [46] and the UML. The approach first attempts to apply the Harel *et al.* interpretation when there is adequately stated meaning of the term being defined semantically; and reverts to the UML approach to semantic definition when there is inadequate description of the term being defined.

The abstract syntax elements described in Table 2.3 have no additional description in

the WFR, i.e. the syntactic definitions of these elements, as given in Figure 2.9 are complete. The WFR for the Constraint and Method elements from the meta-model of Figure 2.9 are presented in the following Table 2.4:

Table 2.4: UML Well-Formedness Rule [43]

	Constraint
[1]	A constraint cannot be applied to itself.
	`not self.constrainedElement→includes (self)`
	Method
[1]	If the realized Operation is a query, then so is the Method.
	`not self.specification→isQuery implies self.isQuery`
[2]	The signature of the Method should be the same as the signature of the realized Method.
	`self.hasSameSignature (self.specification)`
[3]	The visibility of the Method should be the same as for the realized Operation.
	`self.visibility = self.specification.visibility`
[4]	The realized Operation must be a feature (possibly inherited) of the same Classifier as the Method.
	`self.owner.allOperations→includes(self.specification)`
[5]	If the realized Operation has been overridden one or more times in the ancestors of the owner of the Method, then the Method must realize the latest overriding (that is all other Operations with the same signature must be owned by ancestors of the owner of the realized Operation.)

	`self.specification.owner.allOperations→includesAll(`
	`  (self.owner.allOperations→select(op |`
	`  self.hasSameSignature(op))))`

[6]	There may be at most one Method for a given Classifier (as owner of the Method) and Operation (as specification of the Method) pair.
	`self.owner.allMethods→select(operation = self.operation)→size = 1`

Table 2.4 gives the WFR in both textual and the OCL. The WFRs are additional syntactic constraints that cannot be expressed in the UML graphical notation. In order for a model to be in full compliance with the UML such models must also obey the associated WFRs.

The syntax of UML is fully defined by the abstract syntax (e.g. see Figures 2.1 and 2.9), the description of the abstract syntax elements (see Table 2.3), and the WFRs (see

Table 2.4). The syntactic domain for the UML is defined as the set of two-dimensional figures containing: *lines*, *arrows*, *circles*, *boxes*, *ovals* and *textual strings*. The grammar of the UML is defined by the set of abstract syntax models as illustrated in the UML Specification document (e.g. Figures 2.1 and 2.9).

2.5.3 UML semantics

The semantics of the UML is given in a number of sections of the UML Specification [43], with the main section on semantics contained in Chapter 2, of the Specification. In Table 2.5: the first row is the description of semantics for the *Operation* element of Figure 2.9 as it is given in Chapter 2 (*UML Semantics*) of the UML Specification. The second row of Table 2.5 is the description of semantics for the same *Operation* element of Figure 2.9 as it is given in Chapter 3 (*UML Notation Guide*).

Table 2.5: UML semantics for Operation [43]

Chapter 2 - Semantics of Operation Operation is a conceptual construct, while Method is the implementation construct. Their common features, such as having a signature, are expressed in the BehavioralFeature metaclass, and the specific semantics of the Operation. The Method constructs are defined in the correspond subclasses of BehavioralFeature.
Chapter 3 - Semantics of Operation An Operation is a service that an instance of the class may be requested to perform. It has a name and a list of arguments.

Though the *Notation Guide* chapter of the UML Specification states that its section on *Semantics* is a "*brief summary*", and "*a fuller explanation and discussion of fine points*" on the semantics is contained in the *Semantics* chapter; it has to be noted (from Table 2.5) that the *full* semantics is obtained from a reading of both sections. Further, semantic is given for only the concrete meta-classes of the UML meta-models. This is inline with the approach that abstract meta-classes are for the purpose of organizing, reifying, or sharing structural information of the concrete elements.

Because the UML semantics is given in a textual format it is difficult to be precise about some aspects of this semantics, which is required for formal language definition. Consequently, a mapping from the informal (textual) semantic description to a formal semantic

description is required. This will be demonstrated in the following chapter (Section 3.3.2), where a mapping from the syntactic domain to formal semantic domain will be presented.

2.5.4 Model management

Model management in UML defines the mechanisms for structuring models, packages, and sub-systems that are groupings of other UML model elements. Thus, *Model*, *Package*, *Subsystem*, *ElementImport*, and *ElementOwnership* constitute the concrete elements of the Model Management abstract syntax, and *Namespace* being the abstract element (used for classification purposes) as illustrated in Figure 2.10.

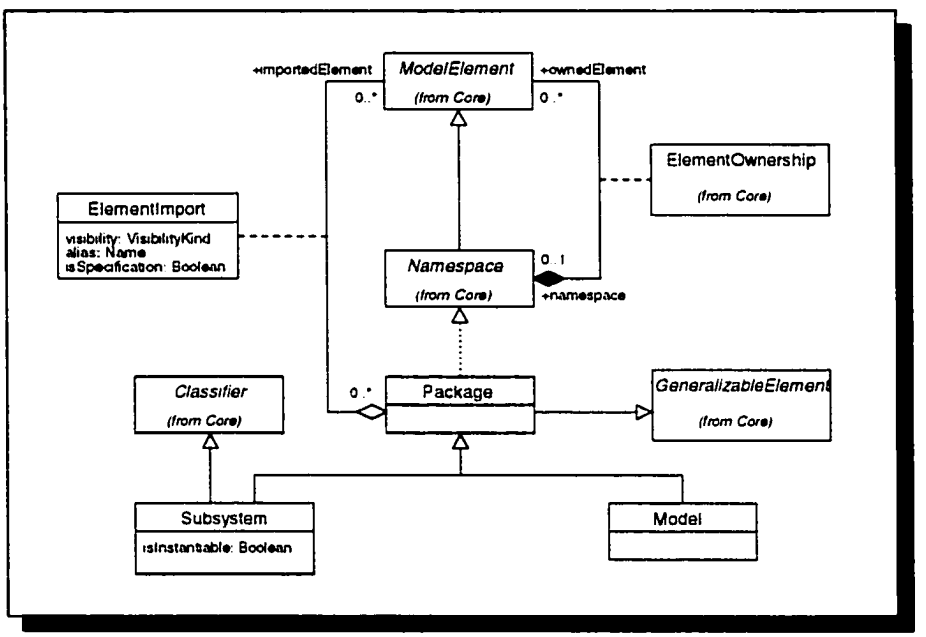

Figure 2.10: UML Model Management abstract syntax [43]

Figure 2.10 shows that *Package* is: (1) a realization of a *Namespace*, (2) a specialization of a *GeneralizableElement*, and (3) a generalization of a *Subsystem* and a *Model*. The description states that a "*Package is a grouping of model elements.*" [43], and the semantic gives its purpose as "*...providing a general grouping mechanism.*" [43]

A key feature of the Model Management construct, for this work, is the structuring capabilities that are provided. An example of this is: a Package and a Model are realizations of model elements, and a Package contains model elements, hence a Package may contain

a Model and a Package may contain a Package, this is demonstrated in Figure 2.1. This is the basis for the use of stereo-typed Package ≪profile≫ in defining DSML.

2.5.5 Extension mechanism

The UML semantics defines the *Extension Mechanism* (EM) as:

> "*... the subpackage that specifies how specific UML model elements are customized by using stereotypes, constraints, tag definitions, and tagged values.*" [43]

It further states that a "*coherent set of such extensions, define for specific purposes, constitute a UML profile* [43]. These statements suggest that the UML may be tailored for use in a manner that is specific to a particular developer's or domain's requirements. The stereotype construct is the principal extension mechanisms. It provides the means for defining new $meta - class$ elements in the UML. The abstract syntax for the EM is illustrated in Figure 2.11.

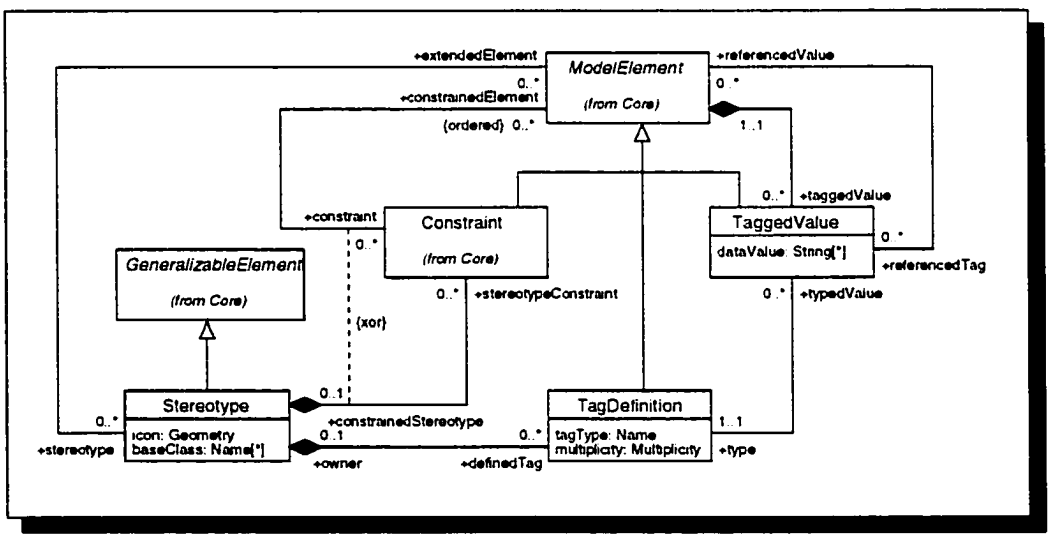

Figure 2.11: UML Extension Mechanisms abstract syntax [43]

From Figure 2.11 it may be noted that a stereotype is a specialization of *GeneralizableElement*, and may include *Constraint* and *TagDefinition*. The constraint and tag definitions constructs allows the definition of new semantics for the UML model element that has been specialized as a stereotype. A tag definition specifies new properties that

are attached to the associated stereotyped model element. While constraints are used to refine the semantics of the model element to which it is attached. The UML defines a *"fundamental constraint"* on all UML defined extensions which states that the extensions have to additive to the UML semantics, i.e. such defined extensions cannot conflict with the standard semantics. This form of extensibility is referred to as *lightweight extension* [20], as the application of the EMs preserves the semantics of the UML model elements. This contrasts with *heavyweight extension* [20] in which the EMs are applied to the UML meta-model (i.e. at the M3 level of the MOF). This type of application can result in changes to the semantics of the resulting meta-model elements, i.e. changes to the semantics of the UML model elements at the M2 level.

2.5.5.1 Stereotype

The UML specification defines a stereotype as

> *"a model element that defines additional values (based on tag definitions), additional constraints, and optionally a new graphical representation."* [43]

A stereotype specification is composed of a **Stereotype** (name), **Base Class, Parent, Tags, Constraints** and **Description**, as illustrated in Table 2.6.

Table 2.6: UML Stereotype template

Stereotype	Base Class	Parent	Tags	Constraints	Description
Stereotype name	Model element from which the stereotype is defined	Stereotype parents	Tag definition names	Constraint expressions	Textual description of the stereotype

The **Stereotype** (name) is user defined and may appear between matched guillemets in the Base Class model element icon. The name has to be unique as specified by the UML WFR [43]:

> [1] Stereotype names must not clash with any base class names.
> `Stereotype.allInstances→ forAll(st | st.baseClass <> self.name)`

The **Base Class** is the UML model element that the stereotype is defined from. The **Parent** of the stereotype is the super-class from which the stereotype has been specialized.

This Parent may be another stereotype or a UML model element. If it is a stereotype then the base classes of both the Parent and the Stereotype must be the same. **Tags** are the names of all tag definitions for the stereotype. Tag Definitions are specified in table format, similar to that of the Stereotype, and example is given in Table 2.7 and elaborated on below. **Constraints** may be expressed in either an informal or a formal notation. The Constraints allow new semantics to be attached to model elements that are tagged with the stereotype name. **Description** gives additional information about the stereotype but is not required for the definition.

Table 2.7: UML Tag template

Tag	**Stereotype**	**Type**	**Multiplicity**	**Description**
Tag definition name	Stereotype associated with this Tag	Tag type, and range of values	Number of values of the Type	Textual description of the stereotype

Tag definitions, like stereotypes, are defined at the meta-model level (M2 level) of the OMG's four layer meta-model architecture [43]. Tag definitions facilitates the specification of "*virtual*" meta-attributes. The **Tag** column of the template records the user-defined name of the Tag, and the **Stereotype** column lists the name of the Stereotype (if there is one) that the Tag is attached to. The **Type** column gives the type of the Tag and the range of values (tagged values) of this Type that are applicable for this Tag usage. Type may be a UML datatype or a meta-class. If the Type is a UML datatype then the range of values are a subset of the defined values of the datatype. If Type refers to a meta-class that is not a datatype then the associated values (tagged values) are model-level (M1 level) elements that are instances of the meta-class. The **Multiplicity** column gives the number (count) of data values, or number of model-level (M1 level) elements that can be associated with the Tag type. **Description** gives additional information about the Tag, but is not required for the definition. All model-level elements that are *branded* with a stereotype name inherits all the properties of the base-class of the stereotype, along with the tag properties of the stereotype.

2.5.5.2 Profile

A *profile* is a UML stereotyped package that contains a set of meta-model elements that have been customized by use of the EM for a specific purpose and are based on a set of requirements. Profile packages are stamped with the ≪profile≫ stereotype. A profile may only contain datatypes, stereotypes, tag definitions, and constraints. The base-classes of the profile stereotypes are include in the properties of the profile by having their names listed as the *tagged values* of the profile's tag definition {*applicableSubset*}. The WFR for these two constraints are given as:

```
self.contents→forAll(e |
   e.oclIsKindOf(Stereotype) or
   e.oclIsKindOf(Constraint) or
   e.oclIsKindOf(TagDefinition) or
   e.oclIsKindOf(DataType))
                  and
self.applicableSubset→
   includesAll(self.stereotypes→collect(baseClass))
```

respectively. The default tagged value of the *applicableSubset* meta-attribute for tag definitions is the "*whole UML meta-model*" [43]. In the work of this dissertation profiles will be used to package definitions for the components of DSMLs and for the packaging of the DSML itself. Thus, as identified in Section2.5.1 profile packages will encapsulate other profile packages that contain the components of a DSML.

2.5.6 UML models

The UML Specification [43] list a number of models that may be developed to present particular views of a software system. It is possible to list the UML models that are applicable at the three main phases of software development (requirement analysis, design, implementation). At the requirement phase the two models that are chiefly used are the *use case* and analysis *class diagram*. These are complemented at the design phase by design class diagram, the behavioral diagrams, which comprise *state chart*, *activity*, and *interaction* (*sequence* and *collaboration*) diagrams. And finally at the implementation phase the UML models are the *component* and *deployment* diagrams. The DSML defined in this work will be used to specify a desired sub-set of these model.

In developing DSMLs for different domains there will be requirements for different subsets of UML models. The goal is to use a core set of models that facilitates the implementation of DSMLs. Listed below are short description of each UML model that may be used in defining a DSML:

1. *Use case diagram* - is used at the analysis phase to present a structural view of the functional requirements of the problem domain. A use case diagram illustrates the relationships between the use cases, of a system and the actors (users) of the system, with respect to the services requested of the system by the user or the services provided by the system to the user. The UML specifies the notation for a use case diagram as a *graph* of actors and a set of use cases, where a "*... use cases is a kind of classifier...*" [43], and an instances of a use case is termed a *scenario* [62]. A scenario is described as "*... a specific sequence of actions and interactions between actors and the system under discussion*" [62]. Use case diagrams may contain four types of relationships:

   - *Association* - is the only relationship that exist between actors and use cases.
   - *Generalization* - exists between use cases and allows for the specialization/generalization between use cases.
   - *Extend* - exist between use cases, and indicates that behavior of one use case is augmented with additional behavior so as to define the second use case. This is a stereotyped relationship.
   - *Include* - exist between two use cases and indicates that the behavior of one use case is included in the specification of the other. This is a stereotyped relationship.

   Graphically, a use case is presented as an ellipse, and the use case name is contained within the ellipse.

2. *Class diagram* - is used at the analysis phase to present a view of the static entities in the problem domain, and at the design phase to present a view of the static entities

(classifiers) in the solution domain. This is the UML diagram used to captures the static information at the requirement phase of software development.

A class diagram (CD) is described as "*... a graph of Classifier elements connected by their various static relationships.*" [43]. The set of classifier elements that may be present in a CD include interfaces, packages, relationships, instances, and links, etc. The UML labels a CD that contains no classes an *object diagram* (OD).

3. *Activity diagram* - is used to present the control or data flow of a process. An activity diagram is a type of state machine, where the states are represented by actions or sub-activities, and transitions are implicitly triggered by the completion of an action or a sub-activity. An activity diagram is associated with a classifier, such as a use case (at a high level or granularity) or an operation (at a more detailed level of granularity).

   Activity diagrams are made up of a sub-set of:

   - *Action state* - is one in which there is "*... an entry action and at least one outgoing transition involving the implicit event of completing the entry action*" [43]. Action states are *atomic* states as they have no internal transitions, or outgoing transitions based on explicit events, or exit actions.
   - *Call state* - "*... is an action state that has exactly one call action as its entry action.*" [43] The representation of a call state is the same as an action state, but with the name of the operation, of the call action, included in the icon.
   - *Sub-activity state* - is one in which, upon entry a *nested* activity graph is executed. Exit from a sub-activity state occurs upon reaching the final state of the nested activity graph, or a trigger event occurs on a transition that exits the sub-activity state.
   - *Decision* - are guard conditions that indicate different possible transitions, from a single state, that depends on the Boolean condition of the owning object.
   - *Swimlane* - organize actions and sub-activities into units of responsibilities, such as organizational units.

- *Control icon* - is an explicit symbol for specifying certain receiving, sending, and deferring of information on transitions. They are chiefly used for enhancement of activity diagram presentation, but are not required for specification of the diagram.
- *Synch state* - is one that indicates the synchronization of parallel (concurrent) activities.

4. *Collaboration diagram* - is used at the design phase to present a view of the interaction in the solution domain. A collaboration diagram may express the interaction (1) between classifiers in the realization of a use case (similar to an activity diagram), or (2) between the classes and local variables in the execution of an operation, i.e. a finer level of granularity that at the classifier level.

   A collaboration diagram is one of the two UML interaction diagrams that are used to illustrate object and message interactions [62]. The UML describes a collaboration as "*... those Instances and their cooperation involved in accomplishing a purpose or a related set of purposes* [43]. Collaboration diagrams may be used to show a graph of either ClassifierRoles and AssociationRoles or Instances linked to each other. The order of interaction is determined by a sequence of numbers, usually starting with 1. In a procedural flow of control nested calls are represented by nested number sequence.

5. *Sequence diagram* - are used at the design phase to present "*... an Interaction, which is a set of Messages between ClassifierRoles within a Collaboration*" [43]. A key feature of sequence diagram is the capability to express timing constraints that are crucial for the modeling in real-time system domains.

   Unlike collaboration diagrams, sequence diagrams have no numbering scheme. Ordering of interactions in sequence diagrams is achieved by exploitation of 2-dimensional structuring of the diagram. The vertical dimension in a sequence diagram represents time, and the horizontal dimension represents different Instances of the interaction.

6. *State chart* - is used to present a view of the entity's behavior vis-á-vis the specification

of the entity's response to the receipt of an event. Such models are crucial in real-time reactive systems. The UML views "*activity diagrams as a special case of a state diagram*", thus some aspects of DSML definition and development that are pertinent to state diagrams are accounted for in those of activity diagrams.

State chart diagrams are "*... used to describe the behavior of instances of a model element such as an object...*" [43], by illustrating the different states that such instances may transition through as a result of discrete events that the instances is aware of.

Component and *deployment diagrams* - are used at the implementation phase of system development, and are not considered in this work. Language components that are used to represent the use case, activity, collaboration, class, state chart, and sequence diagrams are documented herein.

2.5.7 Object Constraint Language

The Object Constraint Language (OCL) is a UML defined formal language that is used to express constraints, such as: (1) invariant conditions for the system being modeled (i.e. constraints on the modeling language), and (2) to specify constraints that are application model-specific (i.e. constraints on the application models). OCL is also used to formally specify the WFRs of the meta-classes of the UML meta-models. The UML lists six purposes for the use of OCL, not including the specification of WFRs. The following four purposes will be used extensively in the definition of DSML:

- "*To specify type invariants for Stereotypes*
- "*To describe pre- and post- conditions on Operations and Methods*
- "*As a navigation language*, and
- "*To specify constraints on operations.*" [43]

The OCL is defined as a "*pure expressive language*" [43], i.e. it has no side effects on the model to which it has been applied. Evaluation of a model's OCL statements results in the return of a value, and the state of the model after the after expression has been evaluated is the same as the state before evaluation of the OCL expression. OCL is a typed language,

thus each OCL expression has a type associated with it. Comparison of expressions in OCL requires that the operand expressions be of the same type, or sub-types of the same super-type. All OCL expressions are given in the context of an instance of a particular type. All classifiers within the scope of an OCL expression are types in the expression. The OCL has a set of predefined type definitions that are a part of the OCL definition. The basic OCL types, some of their associated operations, and example values are presented in Table 2.8 (as taken from [43]).

Table 2.8: OCL basic types

Type	Operator	Value
Boolean	and, if-then-else, implies, not, or, xor	false, true
Integer	*, +, /, -, abs()	1, -5, 34, 26534, ...
Real	*, +, /, -, floor()	1.5, 3.14, 0.6765, ...
String	toUpper(), concat()	"To be or not to be..."

The OCL is used to navigate through a UML model by traversal of the model's associations, association classes, and package names.

2.6 Related Works

Though this research work is not entirely new there is no single idea in this field which dominates, and most works have had limited application outside of the respective research groups' domains. In conducting this research some of the concepts, ideas, and approaches that are presented in the works of sections 2.6.1, 2.6.2, and 2.6.2.3 will be drawn on. The descriptions of these works will include an initial identification of points of commonality and divergence between the referenced works and the documented research herein. Additional commonality, divergence points will be identified in later chapters that document the research work.

2.6.1 First class extensibility approach

In April, 1999 the OMG released a document titled *White Paper on the Profile Mechanism* [27], whose intent is to provide a "*first specification*" (requirements, properties, and

structure) for the UML *profile* concept. The idea was for the profile concept to be used in defining specializations of the M2 level; standard meta-models, such as the UML, and CWM (Common Warehouse Meta-model), which are derived from the Meta Object Facility (MOF) of the OMG four layer meta-modeling architecture (see Table 2.2). Such specializations would be defined as *Standard Profiles* and these Standard Profiles would in turn be specialized to define *End User Specific Profiles*, which would also be positioned at the M2 level. Figure 2.12 is reproduced from the white paper to provide an illustration of this notion of profiles.

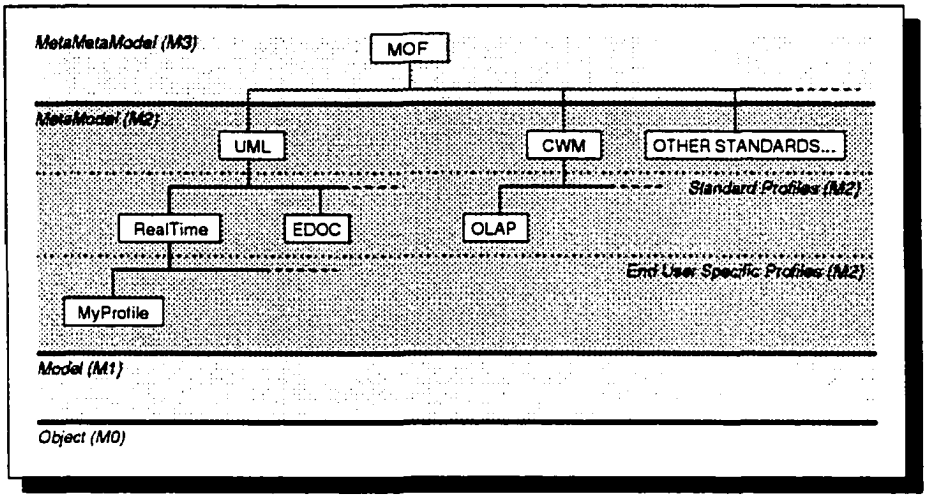

Figure 2.12: White paper profile mechanism [27]

The white paper list fifteen requirements for profiles, which include support for *heavyweight extension mechanisms*. Heavyweight extension mechanism are those that are intended to be applied at the meta-meta-model level (M3) of the MOF hierarchy, i.e. to provide the capability to define *new* meta-class at the M2 level (example, add a new model element to the UML).

Desmond D'Souza *et al.* [20] presents an approach to defining different modeling languages as profiles that is based on their work with packages and frameworks in the Catalysis [21] methodology. D'Souza *et al.* argued that a consolidation of the then current UML so as to improve its model management and structuring facilities, would result in the capability to satisfy the stated requirements for *profiles* without actually adding a new feature

to the UML.

The authors theorize that (1) application of the then existing UML mechanisms of package -import, and -generalization, along with (2) frameworks (*"to describe recurring patterns in models and meta-models"* [20]), and (3) OCL (*"to express constraints on the meta-model"* [20]) would in addition to satisfying the requirements of profile, also provide *"first-class extensibility"*. This theory is based on:

- The existing use of packages to define and structure the meta-models would also be applied to the definition and structuring of extensions to the meta-models. While, package *import* and *generalization* control the granularity of element grouping and visibility.
- Using the package import and package generalization to extend the definition of meta-model elements.
- Defining a rules for the combination of two or more elements that have been extended from the same parent element.
- Utilizing frameworks at the meta-model level to express model level patterns at the meta-model level.
- Using the OCL to describe model-level constraints at the meta-model level.

D'Souza *et al.* argued that the use of *stereotype* and *tagged value* are redundant, because of the available facilities of frameworks. While this may be true the use of stereotype, and tag definition make the work of language definition less complicated than using framework features. Since the publication of D'Souza *et al.* work the UML profile description has been changed to being simply, a stereotype of the UML package construct, that contains a set of meta-model elements that have been customized by use of the EM for a specific purpose based on a set of requirements.

The revised definition and description of profile (see the UML Specification version 1.4 [43]) have simplified much of the debate about profile by providing mechanisms (packaging and EM) that facilitate satisfying most of the requirements for profile in the white paper [27] without including these requirements as explicit features of the UML. Chief among these is the fact that profile is no longer considered a meta-model element, as was done in UML version 1.3, but is now viewed as a UML defined stereotype of *package*, i.e. the UML EM has been applied in the definition of *profile*.

Unlike D'Souza *et al.* this work will *not* consider any aspect of heavyweight extension mechanisms. Heavyweight extension mechanisms have the capability, and are intended to be use to change the semantics of the language it is being applied on. Thus, it is possible to change the semantics of the UML as documented in the UML specification by applying heavyweight extension mechanisms on it. This research will apply only the lightweight extension mechanisms, which *refines* rather than changes the semantics of the language it is applied on.

The ideas of D'Souza *et al.* on (1) using package- generalization and import to extend the definition of meta-model elements, (2) expressing patterns (domain) at the meta-model level, and (3) using the OCL to describe model-level constraints at the meta-model level will be implemented in this research work. In addition, unlike D'Souza *et al.* work, extensive use of stereotypes will be applied herein. Though it must be noted that, similar to profile, the definition and description of stereotype has undergone extensive changes from UML version 1.3 to version 1.4.

2.6.2 Other relater works

Brief descriptions of works that have similarities with the research of this dissertation are presented in the following Sections 2.6.2.1, and 2.6.2.2.

2.6.2.1 The GME approach

The Generic Modeling Environment [63, 50] (GME) is a configurable tool-set that supports the creation of domain-specific modeling and program synthesis environments. GME was developed at the Institute of Software Integrated Systems (ISIS), Vanderbilt University. Similar to the work of this report, GME specifies the configurations of the domain-specific applications through the use of meta-models. In GME a modeling paradigm (the domain-specific modeling language) is composed of syntax, semantics, and presentation information of the domain. Another similarity of the GME approach and that of this work is that GME's modeling paradigm determines: (1) which concepts will be used to construct models (commonality, and variability); (2) what relationships may exist between these concepts (associations, generalization/specialization, etc); and (3) how the concepts may be organized

and viewed by the developer (domain pattern models).

The main differences between the GME approach and this work are: (1) GME automatically synthesizes modeling paradigm from a description of the domain given in GME concepts; (2) GME interfaces with a repository of domain models that are synthesized to generate an application model; (3) GME deals with multi-aspect model integration; (4) the GME models are in the form of entity relationship diagrams; (5) the semantics of the models generated by GME is not defined in GME; and (5) GME is specifically focused on the modeling of large-scale, complex models.

2.6.2.2 The pUML approach

The *p*UML[5] proposed a re-architecture of the UML as *profiles* [12]. In this proposal the UML is composed of a "*... kernel library, which provides a collection of modeling concepts essential to the building of UML profiles*" [12]. Along with such a library the UML will have "*... an extension mechanism for constructing profiles as extensions of the kernel library or other profiles*" [12]. These components will be supplemented by "*... a constraint language for expressing invariant properties of UML models*" [12]. The *p*UML takes an approach to domain-specific language definition that is similar to this work but there are a number of fundamental differences, particular to the that of the definition of semantics for the model concepts.

The *p*UML takes a denotational approach to the definition of semantics for the domain-specific concepts contained in a profile, but their denotational mapping requires a textual description of what the semantic concepts are that the domain concepts have been mapped to. Whereas in this work the semantic domain is a mathematical one (if one exists) or a concise textual description. Another major point of dis-similarity between the two works is that the *p*UML approach seeks to $re-architecture$ the UML, whereas this work explicitly uses the UML as it is presented in the specification. Suffice to say that the *p*UML re-architectured UML would still fit into the process for defining domain-specific modeling

[5]See the web-site at *http://www.puml.org*

languages as researched herein.

2.6.2.3 Additional related works

There are a number of additional works that are related to various aspects of the research work presented in this dissertation. Though it would be impossible to document or list all of these related works a few of them are listed below, along with references:

- Model Driven Architecture (MDA)[6] - The "*Model Driven Architecture (MDA) is an approach to the life cycle integration of enterprise systems comprised of software, hardware, humans, and business practices.*" [19] MDA provides a platform-independent, vender-neutral approach to the challenge of interoperability by use of the established OMG modeling standards, such as the UML. It is expected that as new platforms and technologies are developed the MDA will enable a streamline integration of these new platforms and technologies into existing ones. It is expected that all methodologies, technologies, processes, etc. that is base on any of the involved OMG specifications will be integrated into the MDA approach to software development. Consequently work such as that of this dissertation should be able to fall within the gambit of the MDA.

- *Using OCL for Defining Precise, Domain-Specific UML Stereotypes* [35] - Martin Gogolla presents an idea for the use of stereotype in defining domain-specific UML language constructs that is similar to the approach taken in this work. His intent, like that of this work, is to produce precisely stated stereotype definition of the domain-specific constructs by way of the OCL.

- *Will Domain-Specific Synthesis Become a Silver Bullet?* [11] - In this "*Trends & Controversies*" article a number of contributors look at the growing usage of domain-specific code synthesizer (generator) systems and the benefits that may be derived

[6] See *http://www.omg.org/mda/*

from such systems. This system is of the type worked on in the case study of Chapter 6. The problems, outlined in this article are similar to those encountered in the case study, and the stated benefits of such systems are exactly those being sought after by the developers of the case study system.

- Catalysis [21] - Catalysis is an approach to component-based software development that is made up of a number of techniques and methods for developing high-integrity objected-oriented designed systems. These techniques and methods use the UML notation and has been successfully applied in a number of large-scale industrial projects. The concepts of "*Model Frameworks*" and "*Packaging*" as presented in Catalysis are similar to corresponding ideas applied in this work.

- Domain-Specific Software Architecture (DSSA)[7] - DSSA seeks to make the use of software architecture more rigorous by defining precise meanings and notations for the specification of such architectures. This is intended to lead to the use of machine-process-able languages to capture architectural specifications; tools to analyze these specifications, estimate system properties, and automate the testing of software developed from these architectures. With DSSA, it is expected that reuse and automation will be applied at multiple phases of the system development life cycle. This goal is similar to one of the many goals of DSML usage, and like DSSA, DSML depends on precise meanings of the notation and constructs used in defining components (models) of DSML.

- *Product Lines* [17, 88]. The notion of a software product line exist in an environment where an organization develops and markets a family of products with overlapping but not identical functionalities [88]. Software product line is viewed as one of the more promising proposals for improving the effectiveness and efficiency of software development [17]. One facet of software product line that is addressed in this work

[7]See *http://www.htc.honeywell.com/projects/dssa/*

is that of evolution of software and software requirements. Software product line development is geared toward domain of software and software requirements rather than single product and single product requirements.

- *Software Systems Generator Research* - An area of Don Batory's work with the Software Systems Generator Research Group (SSGRG) at the University of Texas, Austin is based on the idea that while past and current software design methodologies are specific to one-of-a-kind applications, future development methodologies will be centered around *product line architectures* (PLA), i.e. design for related families of applications[8]. The SSGRG definition of a PLA is "*... a blueprint for creating families of related applications.*"[6] This area of SSGRG's research is focused on exploring ways in which software development may be automated so as to increase software productivity, and quality. The essence of this research is to take the concept of software components, which import and export standardized interfaces to a "*logical conclusion*", a *codeless programming environment* where developers will have the:

  > "*... ability to assemble applications of a product-line quickly and cheaply merely through component composition; no source code has to be written.*"[6]

- *Stanford Medical Infomatics* - A research group at Stanford Medical Infomatics (SMI), has been carrying out work aimed at separating the modeling of software systems into *domain knowledge* and *problem solving methods*, which reason over the domain knowledge (see [34, 41, 76] and web-site $http://www-smi.stanford.edu$). Such knowledge-based systems seek to promote reusability and shareability of the given components. The SMI group developed a tool, *Protégé-2000*, whose original goal was to reduce the knowledge acquisition bottleneck (as described in [48]). Protégé-2000 is a set of tool modules which aid in the construction of domain specific knowledge bases. Though Protégé-2000 was originally developed for a medical domain it has

[8] see the SSGRG web site at: *http://www.cs.utexas.edu/users/schwartz/pub.htm*

evolved into a tool to construct knowledge bases for DSLs in other domains.

Chapter 3

The Domain Specific Modeling Language Components

3.1 Introduction

This chapter presents a description of domain-specific modeling language (DSML). The description is in the form of a presentation of the components (syntax, semantics, domain models, meta-models, and rules) of DSMLs. The architecture for DSML is presented in Figure 3.1. The *DSML profile* package of Figure 3.1 contains meta-models of the domain models that are created during the the DA&D activity. These meta-models are centered around the class diagram (CD) meta-model as illustrated in Figure 3.1. This choice of a central meta-model is based on the complete domain CD containing information from multiple views of a software system. A *complete* domain CD is one in which all classes, attributes, operation signatures, and associations that are relevant to the domain problem under consideration are included.

The *Stereotypes* sub-package of Figure 3.1 contains the domain-specific semantics as determined from the *Domain-Specific Rules* of Figure 3.2. The semantics is in the form of stereotype constraints and denotational [84] mappings. These constraints also specify how models may be instantiated from the DSML meta-models and meta-model elements (e.g. an instantiation of an application CD from the DSML CD meta-model, or an instantiation of an application model element from a stereotyped model element), by identifying the mandatory (common) elements for the application models. The broken lines of Figure 3.1 indicate that there may be other sub-packages and relationships, (i.e. additional meta-models) that are

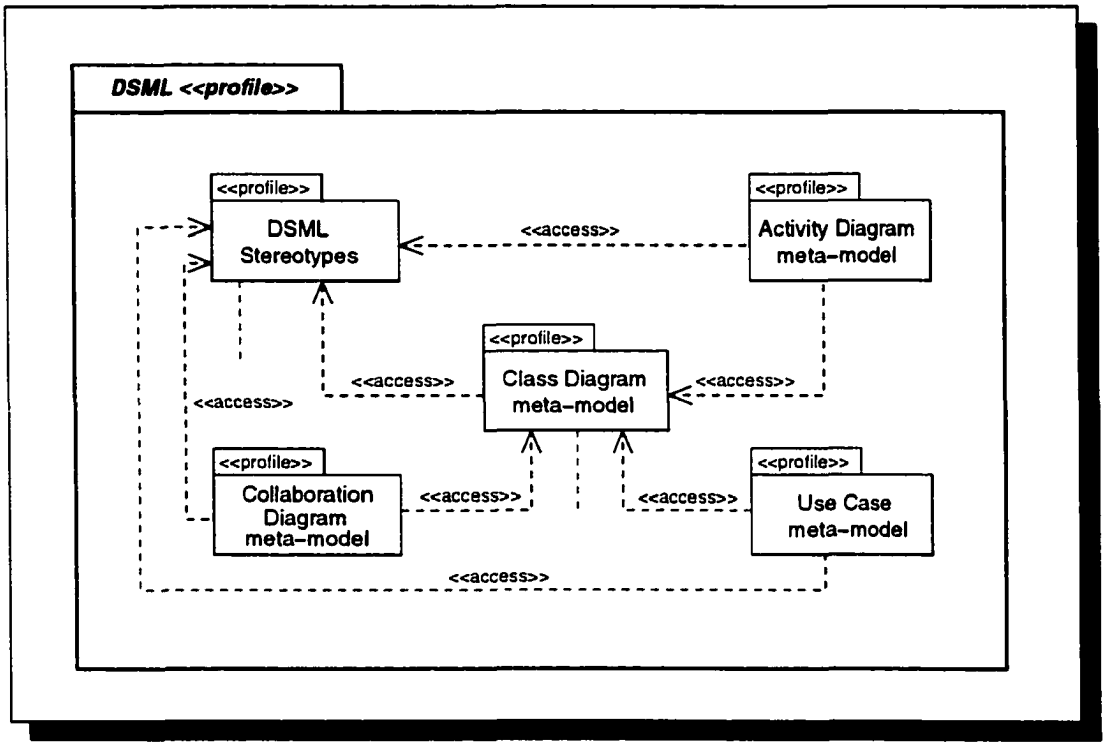

Figure 3.1: DSML architecture

not shown in this particular description.

The environment in which DSMLs are specified and used (as illustrated in Figure 1.2) is also described in this chapter. The DSML syntactic (meta-models) and semantic (stereotypes) basis are presented as the foundation for DSML definition, with the inputs to the definition process for DSML being the domain models and rules. These inputs are described in details, along with examples. The output of the domain language engineering process will be a set of meta-models that defines the syntax of the DSML, and the stereotypes that define the semantics.

3.1.1 Example domain

Starting in this chapter, and continuing through the following chapter, an example problem domain will be introduced for the purpose of illustrating components of the DSML presented. The example domain is titled *Checkin – Checkout*. Applications in this domain provide a set of services to users that allows them to *checkin* and *checkout items* from the

system. There are restrictions on the sequence of service activations, and there may be policies, and categories associated with the users, and items of the system. Examples of systems in this domain are: library, car rental, video rental systems.

Some of the main characteristics of a *Checkin-Checkout* application domain are:

- An *administrator* of the system (librarian in the library system);
- A set of *users* who will be associated with the functions of checking in and checking out (renter - customer in the car rental system);
- A set of *items*, with unique identifiers, that will be checked in and/or checked out (videos in the video rental system);
- A *description* of the *items* (title information in a library system);
- A set of *policies* associated with the *administrator*, *items*, *users*, and *Transactions* (age limit of customers in the car rental system);
- A set of *transaction* information (record of a rental receipt in the video rental system);
- A set of *categories* for *users*, *items*, and *transactions* (fictional or non-fictional in the library system);
- The relationships between the above concepts;
- A set of *add*, *remove* and *update* functions for adding, removing, and updating *items*, *descriptions*, *users*, *categories* and *policies*;
- A set of *checkin*, *checkout*, *browse*, *reserve item*, and *change reservation* functions; and
- A *checkin* activity must be preceded by a *checkout* activity.

These characteristics were determined from the requirement analysis and domain analysis that would have preceded the work studied in this research.

3.2 DSML Environment

The software development environment in which DSML is intended to be specified and used is illustrated in Figure 1.2. This figure is repeated as Figure 3.2 for ease of reference.

Figure 3.2 illustrates the operational framework of DSSEE. The broken line defines the *domain engineering boundary*, which is composed of the activities: *Domain Analysis and Design* and *Domain Language Engineering*. The inputs to *Domain Analysis and Design* are the *requirements* for *future* and *legacy* systems, and *user*. The outputs from *Domain*

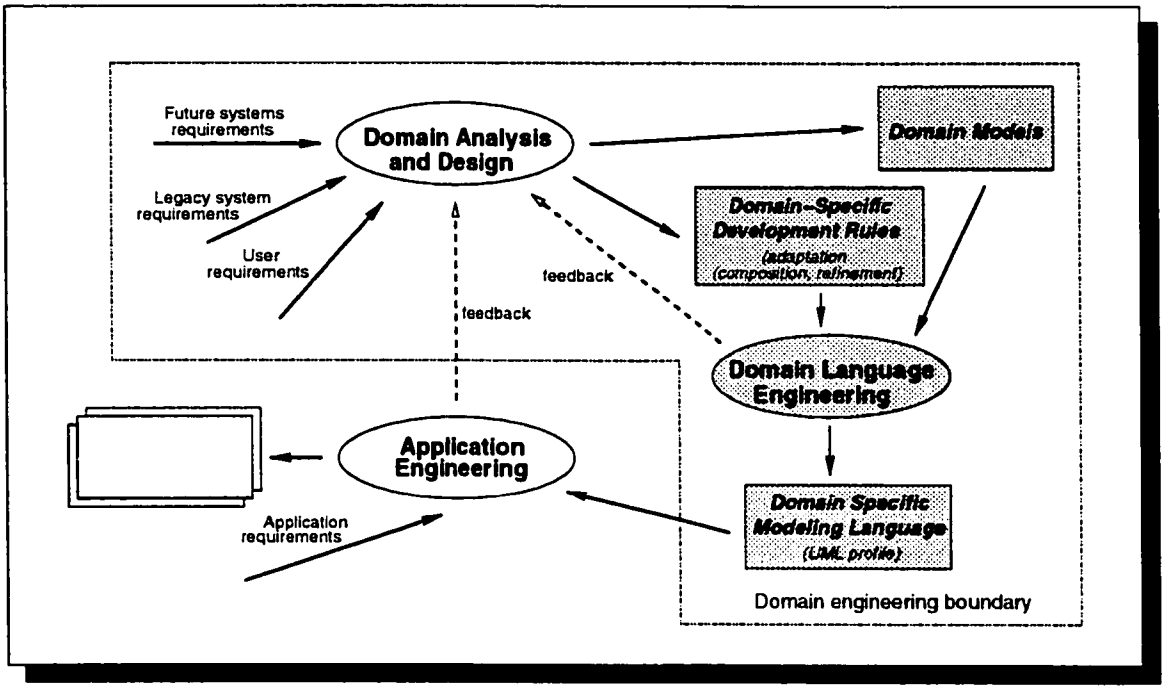

Figure 3.2: Domain-Specific Software Engineering Environment (DSSEE)

Analysis and Design are a set of *Domain Models* and a set of *Domain-Specific Development Rules*. The inputs to *Domain Language Engineering* are the outputs from *Domain Analysis and Design*, with the output being the *DSML*.

The sole activity outside of *domain engineering* is *Application Engineering*. This activity uses the *DSML* to create *Application Models*, based on the input *application requirements*. The activities of *Domain Language Engineering* and *Application Engineering* have *feedback* to the activity *Domain Analysis and Engineering*. These feedbacks are to facilitate modifications to the *DSML* that may arise from errors, inconsistencies, or omissions that have been uncovered during execution of the *Domain Language Engineering* and *Application Engineering* activities.

Only the *Domain Language Engineering* activity, of Figure 3.2, is researched in this work, along with the inputs and output of *Domain Models*, *Domain-Specific Development Rules* and *DSML*. *Domain Analysis and Design* (DA & D) will be discussed in terms of its outputs are used in the language definition process. *Application Engineering* (AE) activities

will be discussed with respect to its use of DSML in Section 4.3.

3.2.1 Domain analysis and design activity

The domain analysis and design (DA&D) 2.4 activity generates a set of domain models and rules that are used in developing DSMLs. The DA&D activity, as used in DSSEE, is a set of tasks and products derived from FODA [56] and FAST CA [95, 96], as outlined in Sections 2.4.2 and 2.4.3, respectively. In Sections 3.2.1.1, through 3.2.1.3 the outputs of the DA&D will be presented.

3.2.1.1 Domain dictionary

In the UML a domain dictionary is an *Artifact* as specified in Section 2.5.2.2 of the UML Specification [43] and would be represented as a stereotype $\ll$*document*$\gg$, which is a sub-class of the stereotype $\ll$*file*$\gg$. There is no UML defined format for a domain dictionary. In this work the domain dictionary is a three column table that lists the *name* of the domain concept, its *properties*, and *examples* of the concept, and is included in the set of domain models. Table 3.1 list a sub-set of the static concepts from the previously identified example *Checking – Checkout* domain. There are usually separate dictionary tables for static and dynamic concepts.

Table 3.1 lists the names, some properties and examples of the identified static concepts of the domain. The *Property* column identifies whether the entity is *concrete* or *abstract*, *optional* or *mandatory*, and initially identified attributes. Concepts with a concrete property are those for which there is a corresponding real world entity, while concepts with an abstract property are those that are used for classification (grouping) and usually have no corresponding real world entity. Concepts with an optional property are those that may be present in some applications of the domain and absent in others, i.e. domain variability. Concepts with mandatory property must appear in all domain applications. The *attribute* of the static concepts are listed along with a type designation. Attributes are assigned an abstract type so as not to impose a design decision at this stage of development. The

Table 3.1: Checkin-Checkout domain dictionary (static concepts)

Term	*Property*	*Example*
user	concrete, mandatory *attribute* identifier: IDENT *relationship* assist, user_cat, user_item, user_tran	customer, borrower
item	concrete, mandatory *relationship* item_cat, item_tran, item_desc, user_item	car, book, video
description	concrete, mandatory *attribute* desc_info: DESC *relationship* item_desc	title information for a book
user category	abstract, mandatory *relationship* user_cat, ucat_policy	categories of customer (preferred, delinquent)
item category	abstract mandatory *relationship* item_cat, icat_policy	categories of cars (economy, full size)
transaction category	abstract, mandatory *relationship* tran_cat, tcat_policy *generalization*	categories of transactions for vehicle rental (daily, weekly rental)
policy	abstract, optional *attribute* poly_type: POLICY *relationship* icat_policy, ucat_policy, tcat_policy, admin_policy	policy of borrow time for category of library users (professors - one year)
transaction	abstract, optional *attribute* trans_date: DATE *relationship* user_tran, item_tran, tran_cat, *generalization*	rent_vehicle, return_vehicle transactions
administrator	concrete, optional *attribute* identifier: IDENT *relationship* admin_policy, assist	librarian, vehicle rental agent
checkout trans	concrete, mandatory *attribute* return_date: DATE *relationship specialization*	borrow a book
checkin trans	concrete, mandatory *attribute* late_return: LATE charge: CHARGE *relationship specialization*	return a video
change trans	concrete, optional *relationship specialization*	change a vehicle reservation
reserve trans	concrete, optional *attribute* start_date: DATE end_date: DATE *relationship specialization*	reserve a book

relationship of the static concept list relationship names associated with the concept and *generalization, specialization* designations indicate whether the concept is a classification (*generalization*) or is classified (*specialization*).

The property column is updated as each activity in the DA process is carried out. This is indicated in the FODA *Input Output* table (Table 2.1). These properties will determine which of the UML model elements the concept will be mapped onto. The dynamic concepts of the domain are similarly documented as illustrated in Table 3.2.

Table 3.2: Checkin-Checkout domain dictionary (dynamic concepts)

Term	*Property*	*Example*
checkin-checkout	abstract, optional	car rental, library rental systems
use system	abstract, optional	borrow a book, return a video
checkin item	concrete, mandatory *specification* An item that had been previously checked out of the system is checked back in.	return a rented vehicle return a borrowed book
checkout item	concrete, mandatory *specification* An item that is available for checkout is checked out by a user.	rent a vehicle borrow a library book
browse item	concrete, optional *specification* Items may be viewed in various order.	view all books on photography
change reserve	concrete, optional *specification* An existing reservation is updated by the user who made the reservation.	cancel a book reservation
reserve item	concrete, optional *pre – condition* An item that is available for the reservation period is reserved by a user	reserve library book reserve vehicle for rental
maintain category	abstract, optional	add a new category of books
add description	concrete, optional *specification* The description to be added must not exist in the system. Only a system administrator can execute this task.	add a new book to the library add a new car model to the rental
remove item	concrete, optional *specification* The item must exist in the system and only a system administrator can execute this task.	copy of a book is removed from the library a vehicle is removed from the rental fleet

Table 3.2 presents the initial definition of a sub-set of the dynamic concepts of the example domain (*Checkin – Checkout*). The *Property* column records properties of the concepts in the form of a textual statement. The properties of *concrete* and *abstract* are the same as that defined for the static concepts in Table 3.1. The optional/mandatory property of the concept is indicated. The mandatory property of a static or dynamic concept are the concepts that are *common* to the domain. The optional static or dynamic concepts those that are variable in the domain. As other task in the DA activity are concluded more information will be added to the domain dictionary.

3.2.1.2 Static concepts model (SCM)

DA&D focuses on the capturing of domain information in the form of a set of UML models. This activity results in the creation of a domain static concept model (SCM) that captures

the static concepts and their relationships in the domain. The domain SCM of the *Checkin–Checkout* domain is presented in Figure 3.3.

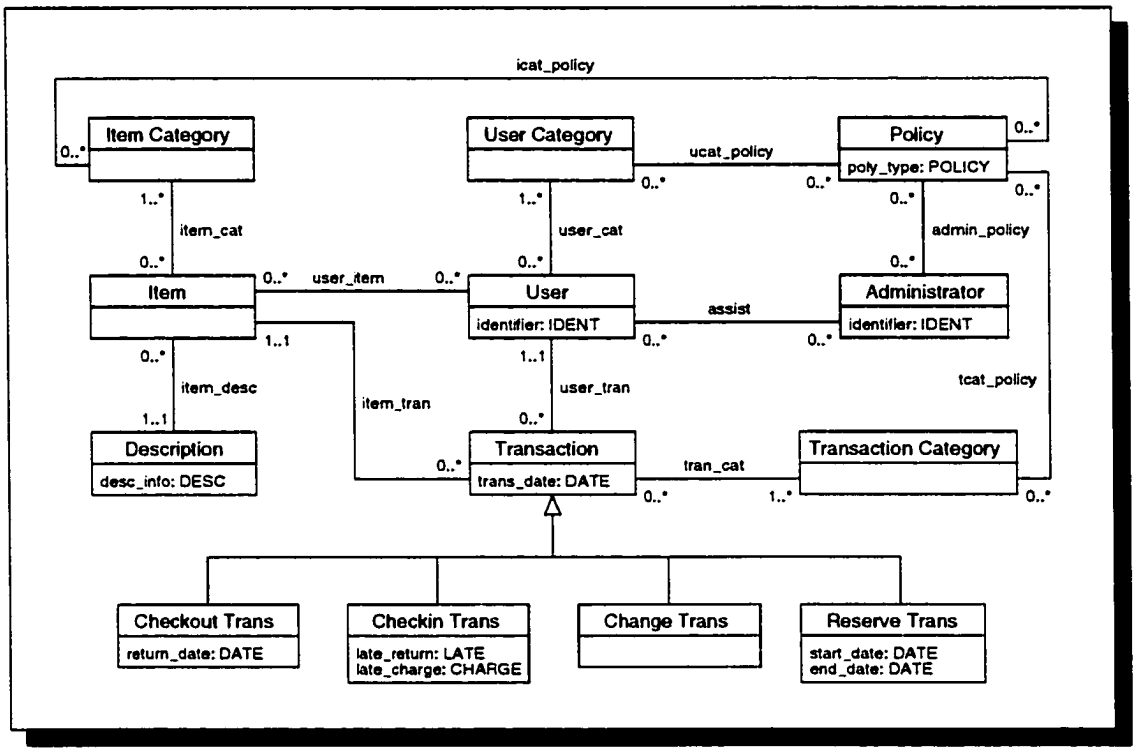

Figure 3.3: Checkin-Checkout domain static concept model (SCM)

Figure 3.3 (SCM) illustrates the static concepts of the domain and the relationships between them in the form of a UML class diagram (CD). The class constructs in Figure 3.3 represents the static concepts of the domain and not classes of objects in the solution domain. This approach is an adaptation from Craig Larman's description of *domain models* [62], where he points out that his domain model (and this work's concept model) "*... is a visualization of things in the real-world domain of interest*, [and] *not software components*" [62]. Larman also points out that his domain model (and this work's concept model) illustrates "*conceptual classes*", which are "*ideas*" or "*things*". The difference between Larman's domain model and the domain model of this work is that Larman's domain model is a model of a *single* instance of a problem, whereas the domain models in this work is a model of a *family* of application problems.

The interpretation applied to multiplicities in a SCM is that such multiplicities are multiplicities of ideas or things. Consequently the multiplicities on the association *item_desc*, between the *Item* and the *Description* concepts state that: (1) an *Item idea* or *thing* must be associated with one and only one (*1..1*) *Description idea* or *thing*; and (2) a *Description idea* or *thing* may be associated with zero or more (*0..**) *Item ideas* or *things*.

The determination as to which UML model element these entities *things* will be mapped to (classes, objects, associations, interfaces, nodes, etc) is not manifested in the SCM. This determination is made at the next stage of analysis, where a domain class diagram will be derived from the SCM. The purpose of the SCM is to identify the static concepts of the problem domain and the relationships between these concepts. Is is from this model that a mapping is made to the software concepts of the solutions space.

3.2.1.3 Dynamic concepts model (DCM)

In DA&D the *Taxonomy of functions* (0.2.3.3 in the FODA HTA of Figure 2.5) focuses on the capturing of the domain's dynamic (functional) concepts. This activity results in the creation of a domain dynamic concept model (DCM) that captures the dynamic features of the domain. These features are later categorized as being either common (mandatory) or variable (optional) features of the domain. This activity is specified in CA as *Identify commonalities* and *Identify variabilities* (see Figure 2.7). The domain DCM of the *Checkin – Checkout* domain is presented in Figure 3.4.

Figure 3.4 is a hierarchical structuring of the domain dynamic concepts in the form of a UML use case diagram. This hierarchical structuring of the use cases facilitates a high degree of reusability, as a developer can select many combinations of concepts (based on the commonality or variability) for different applications from this use case diagram. The actor *user* has access to a subset of the functional concepts, namely *Use system* that is specialized as *Checkin Item*, *Checkout Item*, *Browse. Item*, *Reserve Item*, and *Change Reserve*. While an *Administrator* of the *Checkin – Checkout* system has access to all the dynamic concepts, including those that are accessible by an *User*. In DSSEE each use case is specified in a table format as illustrated in Table 3.3.

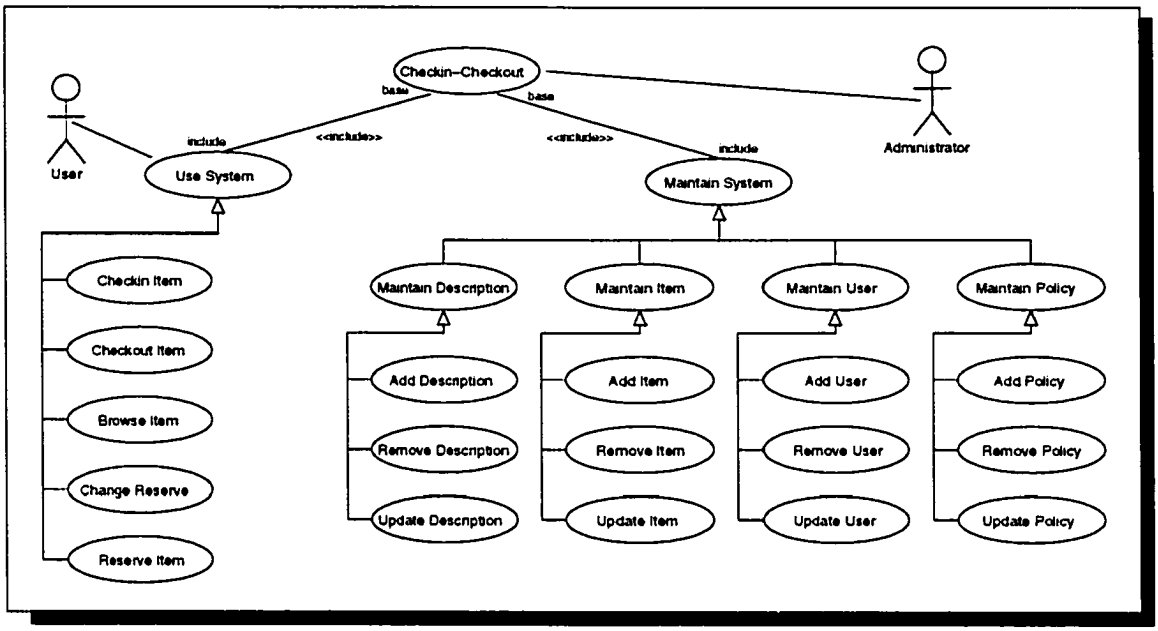

Figure 3.4: Checkin-Checkout domain dynamic concepts model (DCM)

Table 3.3: Use Case Specification

Use Case	Use Case Name
Generalization	Name of use case that generalize this use case
Specialization	Name of use case that specialize this use case
Create	Names of static concepts that are created in order to satisfy the use case
Read	Names of static concepts that are read in order to satisfy the use case
Update	Names of static concepts that are updated in order to satisfy the use case
Delete	Names of static concepts are deleted in order to satisfy the use case
Input	Inputs and types that are required from the actor by the use case
Output	Outputs and types returned to the actor by the use case
Formal description	A formal description of the use case in OCL
Informal description	An informal textual description of the use case, which is derived from the formal description

This tabular specification for use cases is an adaptation of the format used by the organization from which the case study of Chapter 5 was obtained. The use cases are first specified in a textual (informal) format, and then converted to a formal representation by translating the use case specification into OCL. The activity of formalizing the specification leads to a more precise representation of the use case. Once the formalization is complete then the informal specification will be restated from the formal format of the specification.

The process of formalization used in this work is the mapping of a concept from the

informal domain to a concept in a formal domain (a denotational mapping). This mapping creates a formal meaning for the relevant properties of the informal concept for which precision is required. This process may not always produce a complete mapping of all the informal properties to formal properties. In such cases the unmapped informal properties will be retained in terse informal statements.

Formalization of the specification of the domain use cases in OCL is given within a *context* of an OCL expression. This context defines the scope of entities that may be referenced in the OCL expression. In order for this to be accomplished the relationships between the static concepts and the dynamic concepts have to be established. This relationship model,in DSSEE, is referred to as a *concept context model* (CCM), and is an output of the DA&D activity (see Figure 1.2). The CCM for the example domain *Checkin−Checkout* is presented in Figure 3.5.

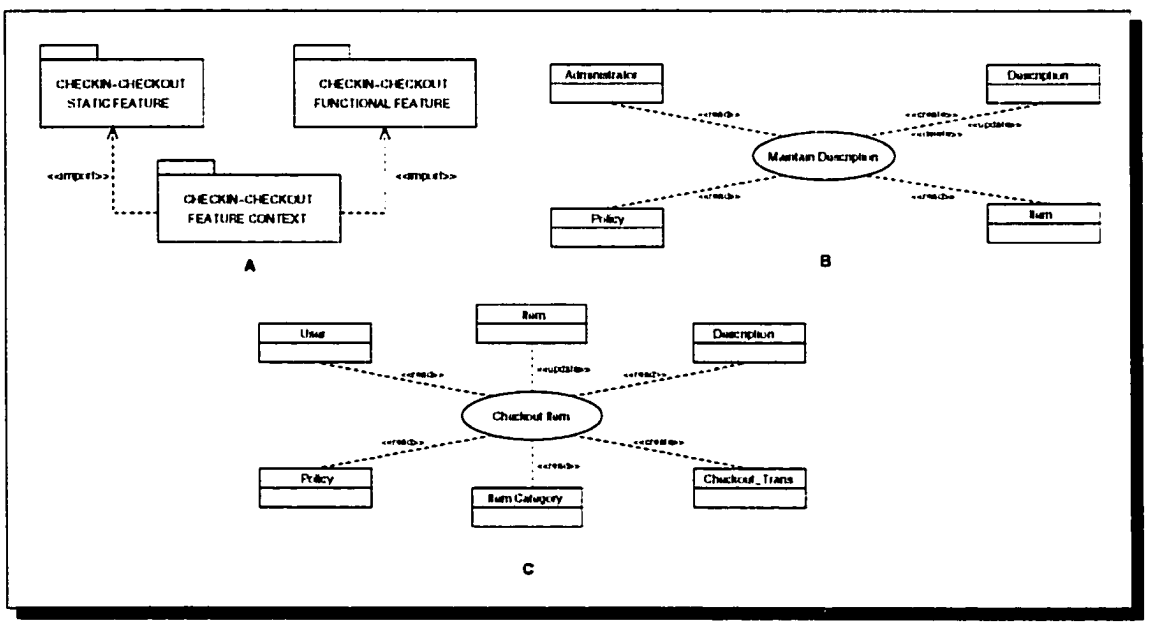

Figure 3.5: Checkin-Checkout concept context model (CCM)

Figure 3.5*A* is a package (high) level description of the relationship between the domain static and dynamic concepts that are illustrated in Figures 3.3 and 3.4, respectively. This diagram shows that the CCM package imports from the DCM and SCM packages. The

interpretation of this *package import* concept is as defined in the UML [43] and illustrated in Figure 2.10. The *visibility* of the imported model elements is *public*, and the *isSpecification* attribute is *true*.

Figure 3.5*B* is an example of the reification of the ≪`import`≫ relationship between the CCM and the SCM and DCM concept elements. The static concepts *Administrator*, *Description*, *Policy*, and *Item* are associated with the dynamic concept *Maintain Description*. The stereotype associations are taken from the use case specification of the dynamic concept. This model identifies all the static concepts that are involved with the three functional concepts *Add Description*, *Remove Description*, and *Update description*, which are sub-concepts of *Maintain Description*. Figure 3.5*C* is the context model for the *Checkout Item* functional concept. In the execution of this dynamic concept the static concepts *User*, *Description*, *Policy*,and *Item Category* are read; *Item* is updated; and *Checkout_Trans* is created. Figures 3.5*B* and *A* are contained in the ≪*concept context model*≫ package of Figure 3.5*A*.

In DSSEE use case descriptions are presented in two format, an informal and a formal version. The formal version is intended for use by language and tool developers who require a precise (non-ambiguous, consistent) expression of the use case specification. The informal version is intended to be used by anyone who requires a clear but non-formal description of the use case. The approach taken in expressing the use case specification in a formal context is an adaptation from the work of the Methods Integration Research Group (MIRG) at the Florida Atlantic University[1]. MIRG presents a specification for Fusion Operation Models [65] in terms of the the system's configuration before execution and the system's configuration after execution of the operation as unprimed and primed (') variables [32, 30] respectively, in the Z [77] notation. In this work a similar approach is taken for the formal description of the use cases, i.e. by use of the OCL's '`@pre`' (before execution state) construct. MIRG's approach with the Z notation is more extensive in its expressions than

[1]See the MIRG web-site at *www.cse.fau.edu/research/MIRG/*

in this work's OCL statements. An example of this is that there is no explicit matching of **@pre** and non **@pre** variables, i.e. statements asserting that values of variables that are not created or updated are unchanged.

An examples of a use case specification (*Remove Description*) is presented below, additional examples of use case (*Add Item*, *Add User*, *Update Policy*, *Checkout Item* and *Checkin Item*) for the *Checkin – Checkout* domain are presented in Section 3.6.6 at the end of this chapter.

Table 3.4: Use Case: *Remove Description*

Use Case	Remove Description
Generalization	Maintain Description
Specialization	
Read	Administrator, Policy
Update	
Create	
Delete	Description
Input	desc: Description, admin: Administrator
Output	result: Boolean
Formal description	**context** Remove_Description **inv:** (Description.oclIsKindOf(Class) **and** Administrator.oclIsKindOf(Class) **and** Policy.oclIsKindOf(Class) **and** desc.item→size() = 0 **and** Description→exists(d | d.desc_info@pre = desc.desc_info) **and** admin.policy→exists(p | p.poly_type = 'remove description') **and** Description→forAll(d | d.desc_info <> desc.desc_info) **and result** = TRUE) **xor** (Description.oclIsKindOf(Class) **and** Administrator.oclIsKindOf(Class) **and** desc.item→size() = 0 **and** Description→exists(d | d.desc_info@pre = desc.desc_info) **and** Description→forAll(d | d.desc_info <> desc.desc_info) **and result** = TRUE)
Informal description	Administrator, admin, must have permission to delete the Description **and** desc is the Description to be deleted **and** desc must exist in the system **and** the number of Item objects associated with desc must be zero **and** after execution Description, desc, does not exist in the system **and result** is true. **exclusive or** for a variation with no Policy concept: Administrator, admin can delete the Description **and** desc is the Description to be deleted **and** desc must exist in the system **and** the number of Item objects associated with desc must be zero **and** after execution Description, desc, does not exist in the system **and result** is true.

The specifications contain both the formal and informal descriptions of the use cases. The formal description is a post-condition of the use case. The pre-condition may be derived from the post-condition. The informal descriptions have been derived from the formal

versions. These descriptions are statement of the requirements of the use cases and are not to be inferred as design or code specifications. Variable names and literals introduced in the use case specifications are presented in `Typewriter font` style, and text from the model figures are in the **Sans Serif font** style. In OCL expressions spaces in model element or variable names will be replaced with an underscore (_), as spaces in names are not allowed in OCL expressions. The context specification for(*oclIsKindOf*) in the formal description of the use cases states that the model elements (*Description*, *Administrator*, and *Policy*) may be instances of meta-classes or instances of stereotypes of meta-classes.

The informal use case descriptions may be used to develop domain application models, but the formal format is required for the domain language definition process and incorporation of DSML into CASE tools.

3.2.1.4 Domain class diagram

From the *Domain dictionary* and the SCM a domain class diagram (CD) is created. The domain CD is developed after the domain concepts of the SCM are mapped to UML model elements. This also results in the introduction of additional concept attributes in the domain CD and domain dictionary. In Section 3.2.1.1 it was stated that the properties of the static concepts are used to determine which UML model element represents each of the static concept. This process of mapping the static concept properties to corresponding properties of UML model elements will be described in Section 4.2.1. A domain CD for the example *Checkin – Checkout* domain is illustrated in Figure 3.6.

The domain CD model of Figure 3.6 is similar to the SCM of Figure 3.3, but with additional attributes. The key difference is that in Figure 3.6 the developers have restricted the domain applications to those for which the user and administrators are of the type person.

3.2.1.5 Domain activity diagrams

For each use case definition a UML activity diagram is created. Activity diagrams for the use cases *Remove Description* (Table 3.4is illustrated below in Figure 3.7*A*. Additional activity diagrams for the use cases *Add Item*, *Add User*, *Update Policy*, *Checkout Item*, and

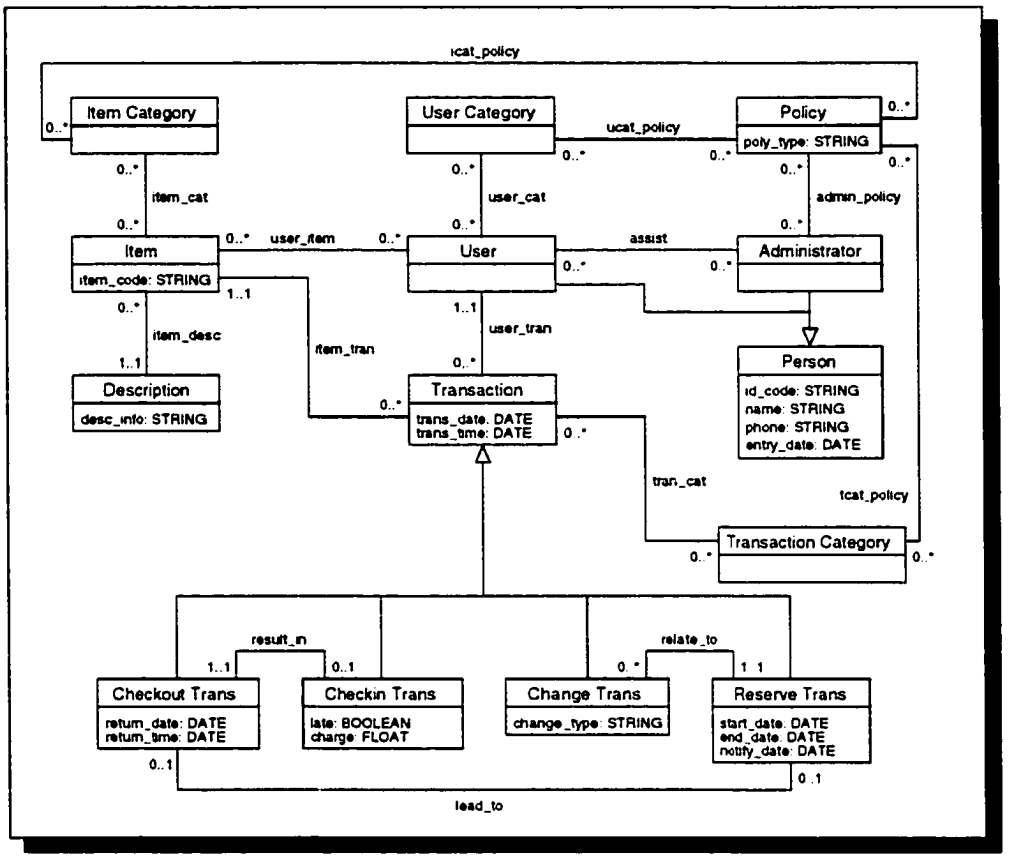

Figure 3.6: *Checkin – Checkout* domain CD model

Checkin Item (Tables 3.10, 3.11, 3.12, 3.13, 3.14, and 3.15) are illustrated in Figures 3.7*A* and *B*, 3.8*A* and *B*, and 3.9*A* and *B*, respectively. These figures are also illustrated in Section 3.6.8.

Figure 3.7*A* illustrates the main (*common*) flow for the *Remove Description* use case as including the sub-tasks; *Administrator Logon*, *Verify Description*, *Verify Item*, and *Remove Description*. The optional (*variable*) flow along the conditional path *remove policy]* is for applications that include a *Policy* for administrators use of the system. Similarly, Figure 3.7*B* illustrates the main (*common*) flow for the *Add Item* use case as including the sub-tasks; *Administrator Logon*, *Verify Description*, and *Add Item*. The optional (*variable*) flow along the conditional paths *[remove policy]* is for applications that include a *Policy* for administrators use of the system.

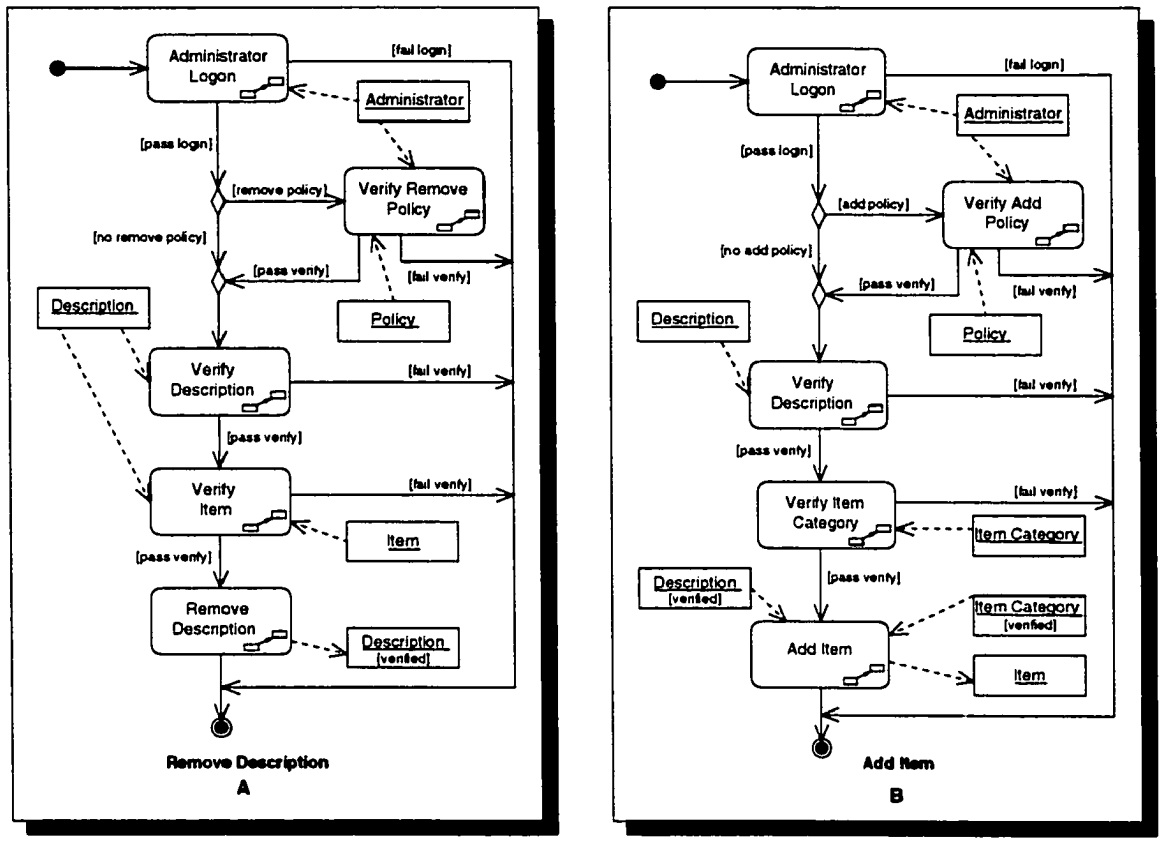

Figure 3.7: Activity diagrams - Remove Description & Add Item

Figure 3.8*A* illustrates the main (*common*) flow for the *Create User* use case as including the sub-tasks; *Administrator Logon*, *Verify User*, and *Create User*. The optional (*variable*) flow along the conditional paths *[remove policy]* is for applications that include a *Policy* for administrators use of the system. Similarly, Figure 3.8*B* illustrates the main (*common*) flow for the *Update Category* use case as including the sub-tasks; *Administrator Logon*, *Verify Category*, and *Update Category*. The optional (*variable*) flow along the conditional path *[remove policy]* is for applications that include a *Policy* for administrators use of the system.

Figure 3.9*A* illustrates the main (*common*) flow for the *Checkin Item* use case as including the sub-tasks; *User Logon*, *Verify Item*, *Verify Description*, and *Create Checkout Transaction*. The optional (*variable*) flow along the conditional paths *[checkout policy]* is for applications that include a *Policy* for users use of the system. Similarly,Figure 3.9*B* illustrates the main (*common*) flow for the *Checkin Item* use case as including the sub-tasks;

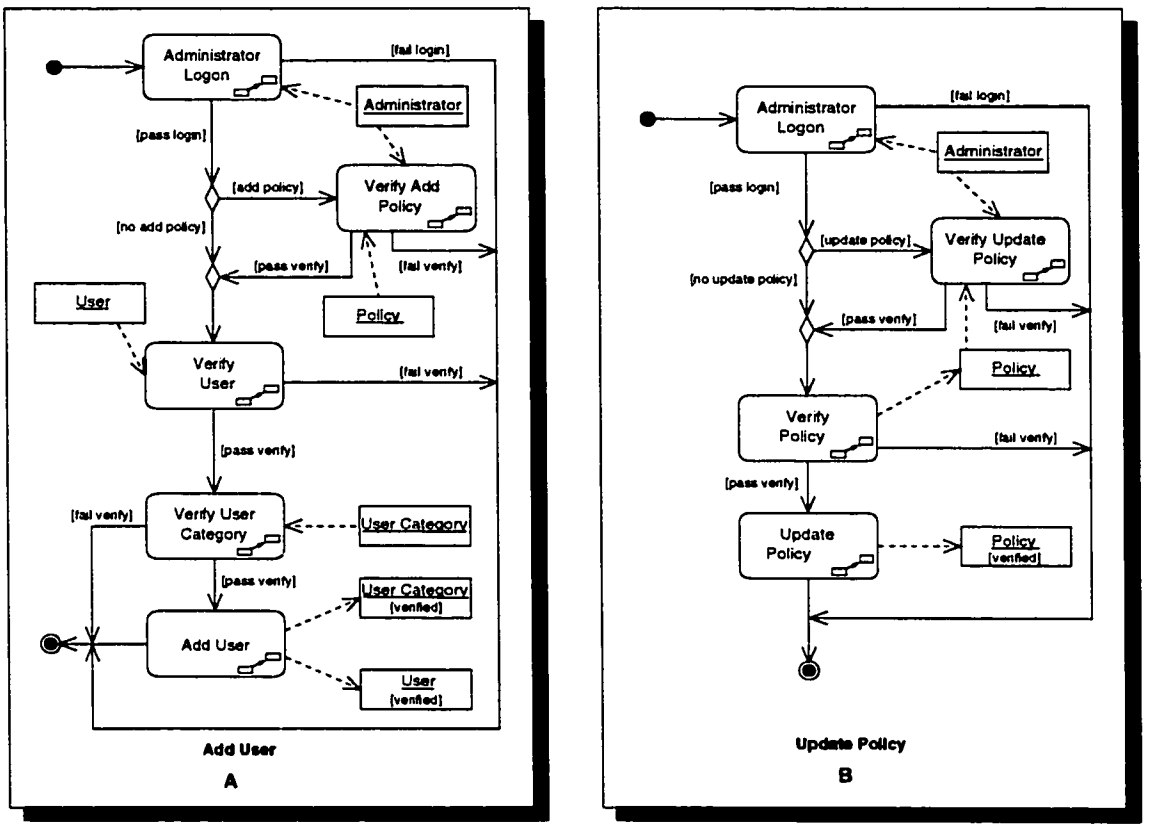

Figure 3.8: Activity diagrams - Add User & Update Policy

User Logon, *Verify Item*, and *Create Checkout Transaction*. The optional (*variable*) flow along the conditional path *[checkin policy]* is for applications that include a *Policy* for users use of the system.

3.2.1.6 Domain collaboration diagrams

For each use case definition a UML collaboration diagram is created. Collaboration diagrams for the use cases *Remove Description* (Table 3.4) is illustrated below. Additional collaborations (*Add Item*, *Add User*, *Update Policy*, *Checkout Item*, and *Checkin Item* (Tables 3.11, 3.12, 3.13, 3.14, and 3.15) are illustrated in Figures 3.10, 3.11, 3.12, 3.13, 3.14, and 3.15 respectively, and repeated in Section 3.6.9.

Figure 3.10 illustrates the collaboration between the *Description* object and the *Item* class in executing the *Remove Description* use case. The collaboration requires that the

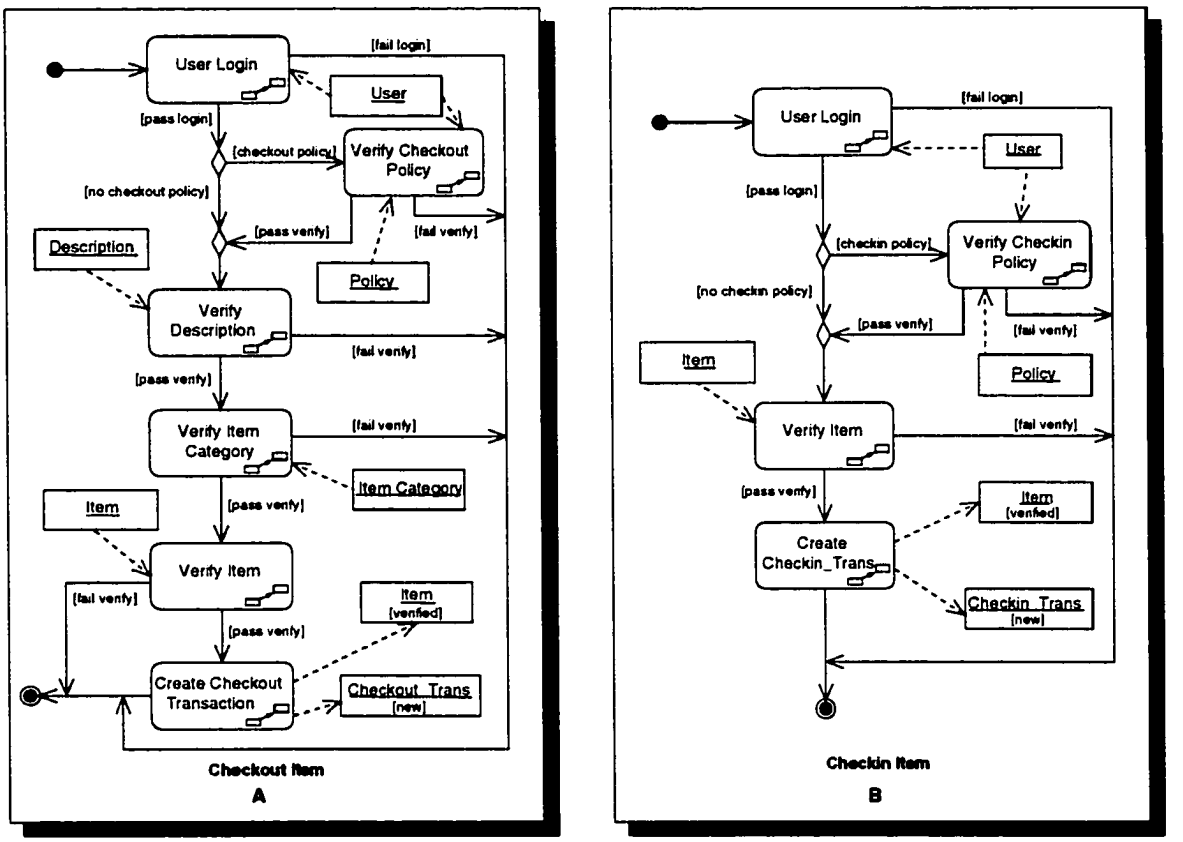

Figure 3.9: Activity diagrams - Checkout Item & Checkin Item

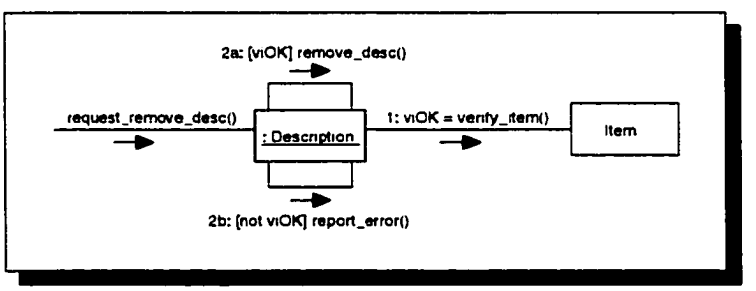

Figure 3.10: Collaboration diagram - Remove Description

Item class is accessed to verify that no *Item* object is associated with the *Description* that is to be removed. A fail condition, i.e. *Item* objects are associated with the *Description* object that is to be removed, is reported as an error (*report_error*), and removal of *Description* does not occur.

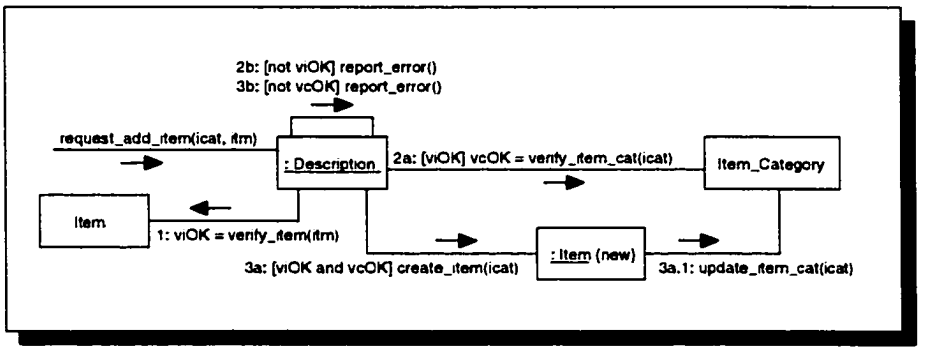

Figure 3.11: Collaboration diagram - Add Item

Figure 3.11 illustrates the collaboration between the *Description* object, *Item Category* and *Item* classes in executing the *Add Item* use case. The collaboration requires that the *Description* object verify that the inputted *Item* (*itm*) does not already exists in the system. The *Item Category* is accessed to verify that the inputted *Item Category* (*icat*) exists. *itm* is then add to the system, and the *Item Category* (*icat*) updated. A fail condition, i.e. *itm* already exist in the system, or *icat* does not exist, is reported as an error (*report_error*).

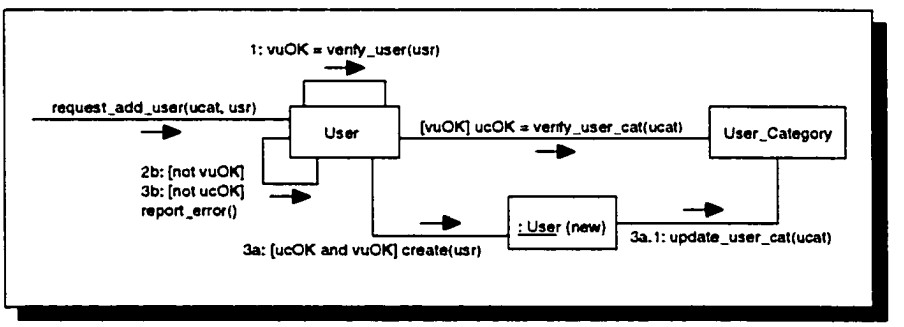

Figure 3.12: Collaboration diagram - Add User

Figure 3.12 illustrates the collaboration between the *User*, and *User Category* classes in executing the *Add User* use case. The collaboration requires that the *User* class verify that the inputted *User* (*usr*) does not exists in the system. The *User Category* is accessed to verify that the inputted *User Category* (*ucat*) exists. *usr* is then add to the system, and the *User Category* (*ucat*) is updated. A failed condition, i.e. *usr* already exist in the system or *ucat* does not exist in the system is reported as an error (*report_error*).

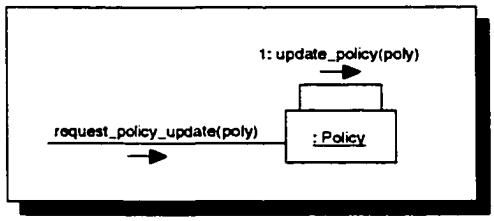

Figure 3.13: Collaboration diagram - Update Policy

Figure 3.13 illustrates the collaboration with the *Policy*, object in executing the *Update Policy* use case. The collaboration requires that the *Policy* object be updated with the new data contained in *poly*. There is no alternative or failed condition in this collaboration.

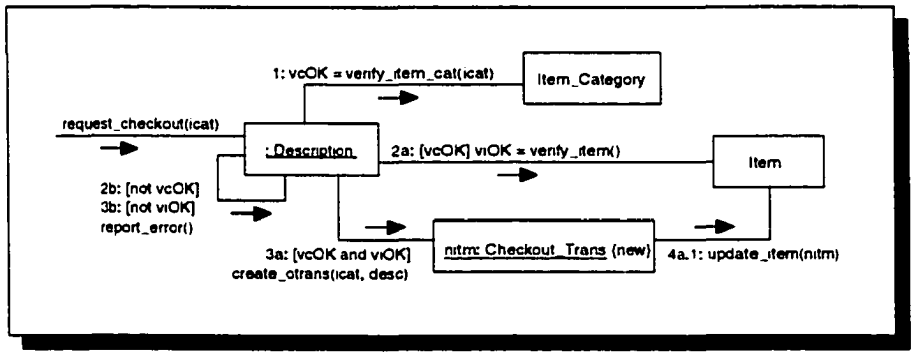

Figure 3.14: Collaboration diagram - Checkout Item

Figure 3.14 illustrates the collaboration between the *Item* an *Item Category* classes and the *Description* and *Checkout_Trans* objects in executing the *Checkout Item* use case. The collaboration requires that the *Description* object verify that the inputted *Item Category* (*icat*) exists, and the *Item* class is accessed to verify that at least one object of *Item* exist in the *Category icat* is available to be checked out. A {*new*} *Checkout Trans* object is then created in the system, and the *Item* class is updated. A failed condition, i.e. the inputted *Item Category* does not exist or no *Item* object is available to be checked out, is reported as an error (*report_error*).

Figure 3.15 illustrates the collaboration between the *Item* and *Checkin Trans* objects in executing the *Checkin Item* use case. The collaboration requires that the *Item* object request that a {*new*} *Checkin_Trans* is created in the system. There is no fail or alternate

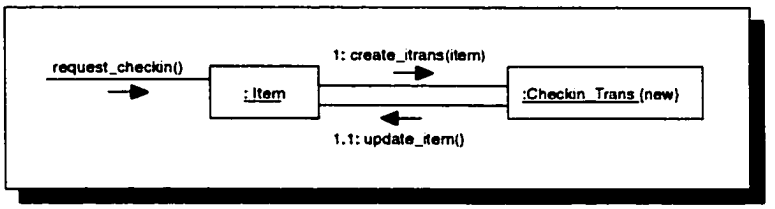

Figure 3.15: Collaboration diagram - Checkin Item

condition in this collaboration.

3.2.1.7 Domain sequence diagrams

A *sequence diagram* is an interaction diagram (like that of a collaboration) that captures timing constraints of the sequence of messages involved in the interaction. A sequence diagram is associated with an *Interaction*, and the "*...purpose of an Interaction is to specify the communication between an ensemble of interacting instances performing a specific task.*" [43] A *Collaboration* is defined as the "*... structure of the participants that play the roles in the performance of a specific task and their relationships.*" [43]. The relationship between an *Interaction* and a *Collaboration* is that a *Collaboration* is related to zero or more (0...1) *Interaction*. This leads to the interpret that a *Collaboration* is characterize by a set of *Interactions*. Hence each sequence diagram can be viewed as a *scenario* of the *Interaction*. This notion of a scenario is similar to that of Larman's description of a *scenario* as "*... a specific sequence of actions and interactions between actors and the system ... or one path through* [a] *use case*" [43].

For each use case definition a set of UML sequence diagram is created. Sequence diagrams for the use case *Remove Description* (Table 3.4) and *Add Item* (Table 3.10) are illustrated in Figures 3.16 and 3.17, respectively. Additional sequence diagrams for use cases *User*, *Update Policy*, *Checkout Item*, and *Checkin Item* (Tables 3.11, 3.12, 3.13, 3.14, and 3.15) are presented in Section 3.6.10 as Figures 3.39, 3.40, 3.41, 3.42, 3.43, and 3.44 respectively. These sequence diagrams are successful execution scenarios for each use case.

At the end of creating the collaboration and sequence diagrams, i.e. the interaction diagrams, it is now possible to determine the responsibilities for the operations of the

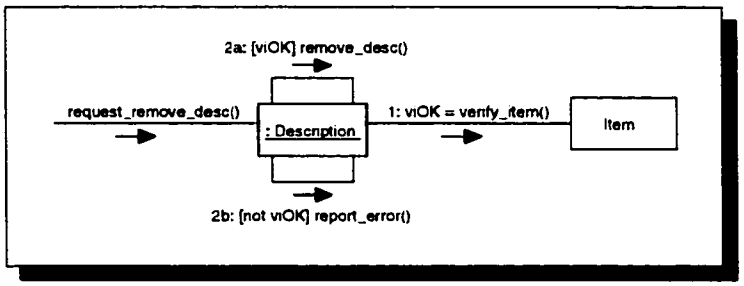

Figure 3.16: Collaboration diagram - Remove Description

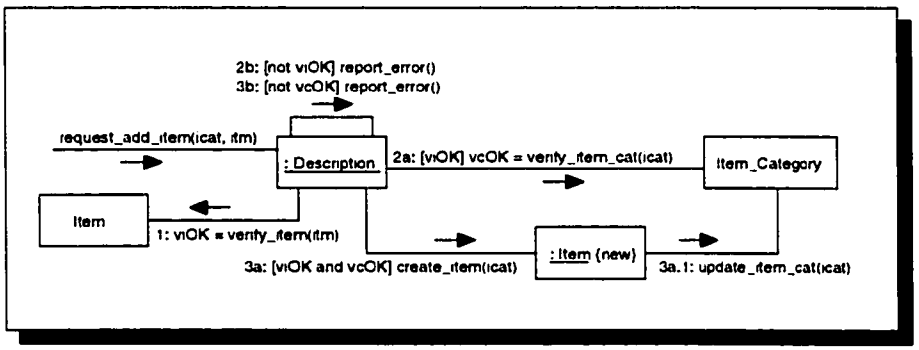

Figure 3.17: Collaboration diagram - Add Item

domain. This information is added to the domain CD to create a domain design CD, which is illustrated in Figure 3.18.

Only a subset of the domain operations are illustrated in the diagram of Figure 3.18 so as to demonstrate the relationship between the interaction diagrams and the design CD.

3.3 DSML Base

The base on which DSML are defined is taken from the specification of the UML, and its model management and EMs facilities. This concept is depicted in Figure 3.19.

Figure 3.19 illustrates the relationship between the *UML meta-model* and the researched *DSML*. DSML are derived from the UML meta-models as stereotyped ≪*profile*≫ packages. The UML meta-model base includes the entire UML specification i.e. the *Model Management*, *Foundation*, and *Behavioral* packages as depicted in Figure 2.8.

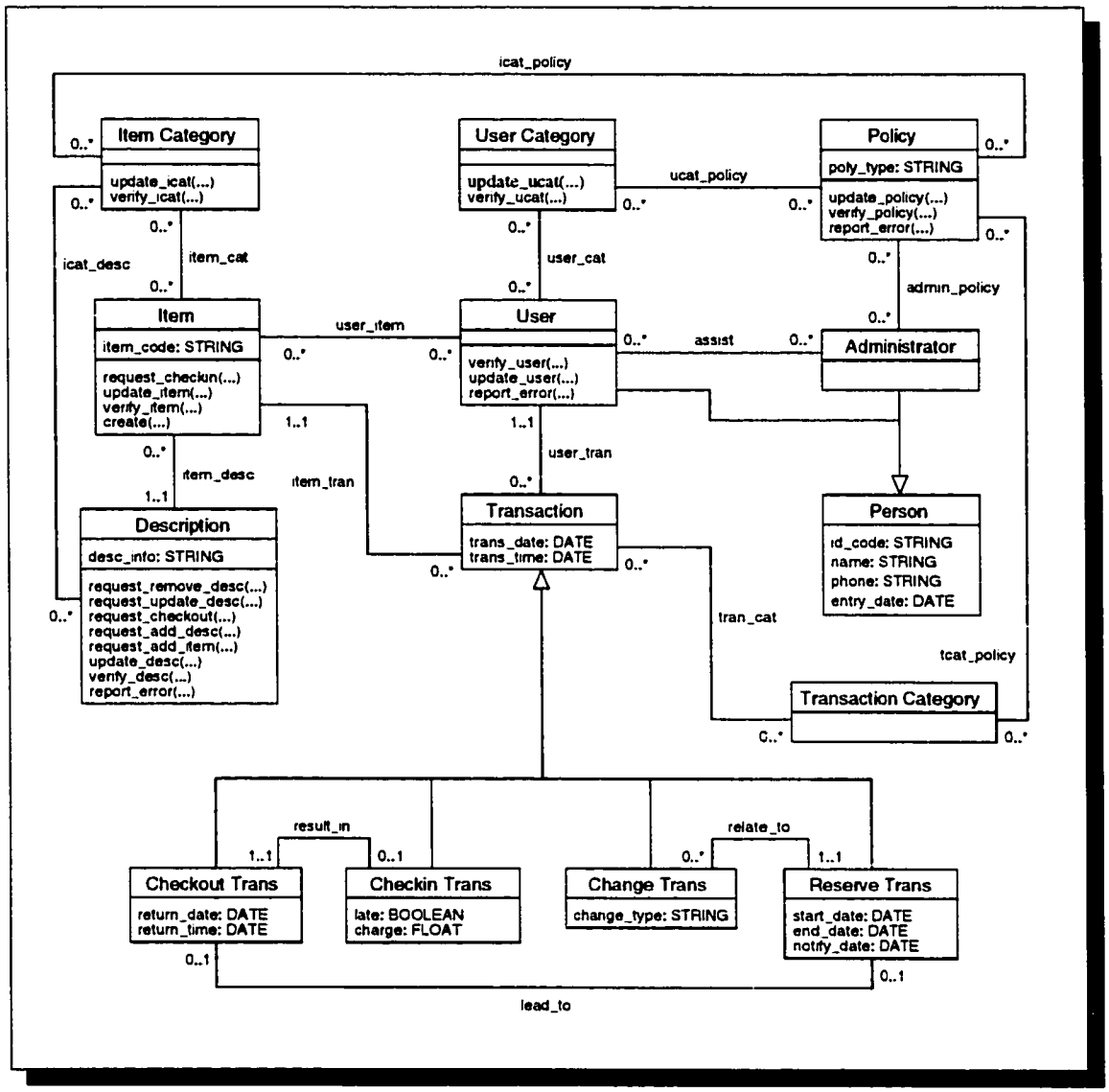

Figure 3.18: *Checkin – Checkout* domain design CD model

3.3.1 Syntactic base

The syntactic base for a DSML may be obtained from the UML syntactic base or it may be defined independently. It is presumed that the syntactic base will be that of a visual language (i.e. chiefly non-textual). The syntactic base of a DSML is defined by a: (1) syntactic domain, (2) a mapping from the syntactic domain to a set of domain constructs (icons, picture, etc.) and (3) a mapping from the domain constructs to the concepts of the

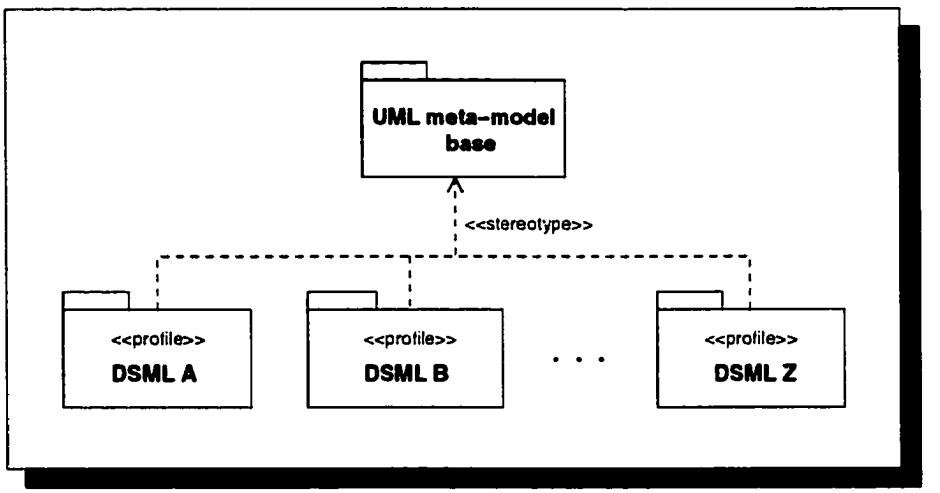

Figure 3.19: The DSML base model

domain. The syntactic domain defines the visual representation for the language, i.e. the language notation.

The UML's syntactic domain is a set of constructs from the two-dimensional space of geometric figures, and alphanumeric characters. The sub-set of figures include: lines, points, and arrows. This sub-set of figures and characters are used to define the concrete syntax of the UML, i.e. the two-dimensional icons for representing the model elements of the abstract syntax, and models that are derived from the abstract syntax. The notation of the language is the set of iconic elements (concrete syntax) that are mapped to concepts of the language. Figure 2.2 gives examples of the *concept to concrete syntax* mapping for the UML.

The syntactic domain for a DSML may be identical to the UML's, a sub-set of the UML's, or taken from a different domain. Similarly the iconic elements used to define the abstract syntax of the DSML may be identical to the UML's, a sub-set of the UML's, or from a different set of icons. The use of a syntactic domain, and syntactic elements that are different from the UML's are permitted in UML, vis-á-vis the EM stereotype mechanism via its attribute *icon* that is a of the type *Geometry* and is defined as a "*... geometric description for an icon to be used to present an image of a model element branded by the stereotype.*" [43] In the example of the example *Checkin – Checkout* domain the UML's syntactic domain and abstract syntax iconic elements will be used.

3.3.2 Semantic base

A semantic domain for the UML is not explicitly given in the UML specification [43], but this can be deduced, for each model element, from its semantic definition. The semantic definition for a *Class* states that: a *"class is a description of a set of objects..."* [43]; and an association is defined as*" a semantic relationship between classifiers."* [43], where the *"instances of an association are a set of tuples relating instances of the classifiers."* [43], and each *tuple value may appear at most once."* [43] These semantic definitions infer that UML classes may be semantically mapped to the mathematical concept of *sets*, and associations to the mathematical concept of relationships [32, 77].

The approach taken in France *et al.* [32] and adapted in this work is to map the (domain) concept to a well known and understood (ideally a formal i.e. mathematical) concept that have similar properties to that of the (domain) concept. The mathematical theory of sets and predicate logic [77] are applicable semantic domains for the UML. Examples of this mapping from a domain (the UML) concept to a well known and understood domain concept is given in Figure 2.2, where the *UML Concept* is mapped to a *Semantic Domain* concept that is mathematical.

In the definition of the semantics of DSML a mapping from the DSML concept to a well known and understood formal (mathematical) domain concept that exhibit similar properties to that of the domain concept. This is the approach outlined in the following works [12, 13, 23, 24], and applied in defining DSML. In the case where it is not possible to map a domain concept to a mathematical concept the semantics of the domain concept will be presented in the form of a concise textual description.

3.4 The Domain Rules

During software development the developer will implicitly and explicitly apply *rules* in defining the models of the application. Some of these rules are from the requirements and some are from the experience gained from previous software development projects. The characteristics of the example $Checkin - Checkout$ domain, as outlined in Section 3.1.1, forms a subset of the rules that constitute the *domain rules* (DR) of the domain. In the *Do-*

main Language Engineering activity of Figure 1.2 the *Domain-Specific Development Rules* are used to constrain the syntax and semantics of the DSML. The rules are applied to the concepts (static and dynamic) of the domain. The rules are expressed as OCL constraints in stereotypes of UML model elements that the domain concepts have been mapped to as a part of the language engineering process. Within DSSEE language engineering three types of rules are specified:

1. *Adaptation Rules* - uses information contained in the domain dictionary, to determine which UML model element a domain concept is to be mapped to, and how the inherited syntax and semantics are to be constrained. Additionally, the adaptation rules specify what denotational-like mappings are to be applied to the concepts to further refine its semantics.

2. *composition rule* - exploits the information contained in the SCM, DCM, domain dictionary, along with domain expert knowledge to determine which of these concepts must be included in an application model of the domain. Execution of the composition rules leads to the definition of models that capture *patterns* of the domain. These rules result in the definition of composition domain concepts from elementary domain concepts (UML model element).

3. *refinement rule* - exploits the information contained in the SCM and DCM, along with domain expert knowledge to determine which domain model elements may be optionally included in domain application models. The refinement rules are applied after the composition rules have been executed. The refinement rules, like the composition rules, result in further definition of composition domain concepts from elementary and composed domain concepts. The refinement rules facilitate the identification of optional UML model elements and variabilities in the domain patterns of the solution space.

These three types of rules are formally represented as OCL constraints and (initially) as a set of domain and development requirements. The *adaptation rules* are chiefly used for defining the semantics of the domain concepts, while the *composition rules* and *refinement*

rules are directed at identifying the common (mandatory) and variable (optional) concepts of the DSML. These rules are contained in the stereotypes of the DSML.

The revised presentation format for stereotype that is introduced in the UML 1.4draft version and continued in the 1.4 version [43] is used in this report, but with the table reoriented to a vertical layout from its original horizontal layout. The purpose for the re-orientation is to aid in the readability of the tables. Table 3.5 is a layout of the stereotype specification, where the constraints are expressed in OCL. The constraint entry is a key element of the stereotype definition as it expresses the semantics of the concepts, by way of the domain rules.

Table 3.5: Stereotype and tag definition specifications (textual)

Stereotype	Stereotype name
Base Class	Stereotype base class
Parent	Name of parent stereotype
Tags	Tag names for this stereotype
Icon[2]	Stereotype graphical icon name
Constraint	Stereotype constraints in OCL
Description	Textual description

Tag	Tag definition name
Stereotype	Stereotype that uses this tag
Type	Tag type and enumerated values
Multiplicity	Number of model elements or tag values for this tag
Description	Textual description

The information contained in Table 3.5 will also be represented in a graphical format as specified in Figure 3.20. In the graphical representation the tag definitions are the additional attributes of the stereotyped (specialized) base class classifier. Thus the language and application developers will have available to them two representations for the rules (semantics) of the DSML.

The stereotypes demonstrate how the UML meta-model elements are transformed into domain-specific elements by application of the EMs (*constraint*, *stereo-typing*, and *tag definition*) to define the semantics of the DSML.

3.4.1 Stereotype examples

In this section domain rules for the *Checkin−Checkout* example domain are presented as a set of UML stereotype definitions for the SCM, and DCM concepts. The meta-attributes of the stereotyped model elements are explicitly assigned a value that help in the specialization of of the model element semantic definition. The ≪*item*≫ stereotype is illustrated below

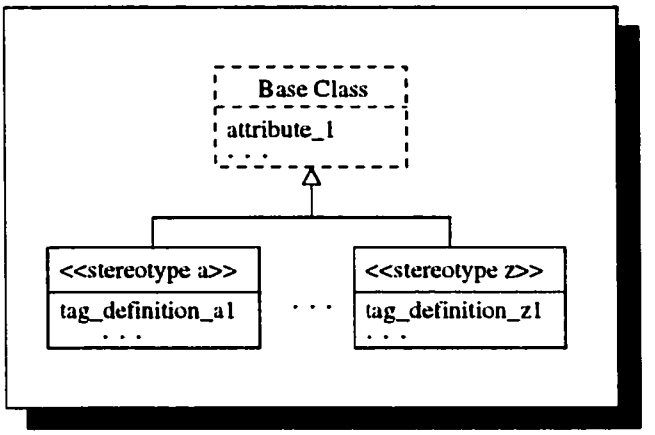

Figure 3.20: Stereotype and tag definition specifications (graphical)

in Table 3.6. Additional tables of stereotypes are presented at the end of this chapter in Section 3.6.11.

Table 3.6: Class *Item* stereotype

Stereotype	item
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪item≫ **inv:** self.isAbstract = false **and** [Stereotyped classes may have instances] self.isLeaf = true **and** [Stereotyped classes may not have descendants] self.isRoot = true **and** [Stereotyped classes may not have ancestors] self.mandatory = true **and** [Stereotyped classes must be included in domain applications] self→isUnique(self.item_code) **and** [Item objects have unique item_code values] self.≪user≫→size() = 1) **and** [An Item class associates with one User class] self.≪description≫→size() = 1 **and** [An Item class associates with one Description class] self.≪transaction≫→size() ≥ 0) **and** [An Item class associates with one or more Transaction class] self.≪item_category≫→size() > 0 **and** [An Item class associates with one or more Item Category class] self.≪item_code≫.multiplicity = 1 **and** [Stereotype class has one item_code attribute] self.≪item_code≫.changeability = frozen **and** [Stereotype class item_code attribute value may be changed] self.≪item_code≫.visibility = public **and** [Stereotype class item_code attributes is private] self.operation→includesAll(request_checkin, update_item, create_item, verify_item) [Operations associated with the stereotyped class]
Description	The *constraint* specifies the meta-attributes, multiplicities and operations of the stereotype.

Table 3.7: Tag definition *Mandatory*

Tag	mandatory
Stereotype	item
Type	UML::Datatype::Boolean
Multiplicity	1
Description	Indicates whether a stereotyped class of this type must be included in a domain application model.

The meta-attribute *mandatory* of the stereotype ≪item≫ has been introduced as a tag definition, which is now available to be applied to any defined stereotype of the domain language. The semantics of the stereotype (≪item≫) is defined by the inherited semantics of its (1) *Base Class*, (2) uniqueness (self→isUnique(self.*item code*)), and (3) the semantics of the associated operations (*request checkin*, *update item*, *create item* and *verify item*).

Precipitating from the definition of stereotype for the SCM concepts, stereotypes for the operations of the static concepts (that are realized as classes) are also defined. The textual stereotypes may also be represented in a graphical format. This graphical format illustrates the stereotypes as realized specializations of the UML model element base classes. This is the principle identified in Section 3.4 and illustrated in Figure 3.20. Graphical examples of the stereotypes for the example $Checkin - Checkout$ domain are presented in Figures 3.21, 3.21, and 3.23.

The *mandatory* meta-attributes of Figures 3.21 and 3.22, and *mandatory* and *actor* of Figure 3.23 have been introduced for the $Checkin - Checkout$ domain. These meta-attributes are the tag definitions that are used in conjunction with the stereotype definitions for the domain. The stereotypes are presented as $sub - types$ of their respective *base class*, thus they inherit the base class' semantics, attributes, and associations as defined in the UML semantics and meta-model (abstract syntax).

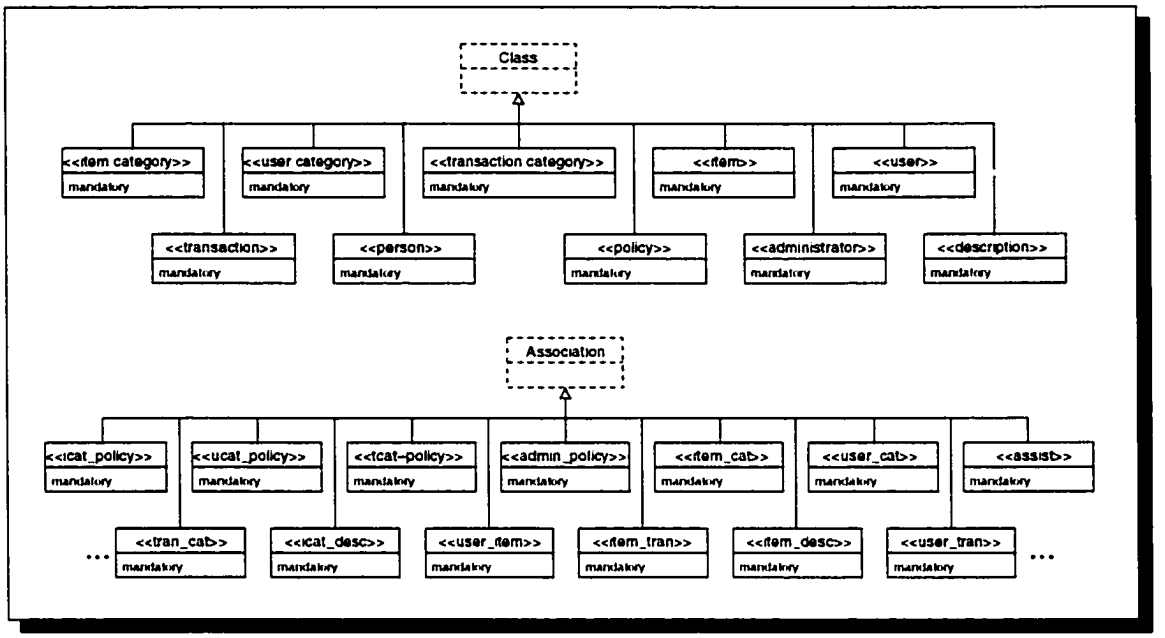

Figure 3.21: Graphical class and association stereotypes

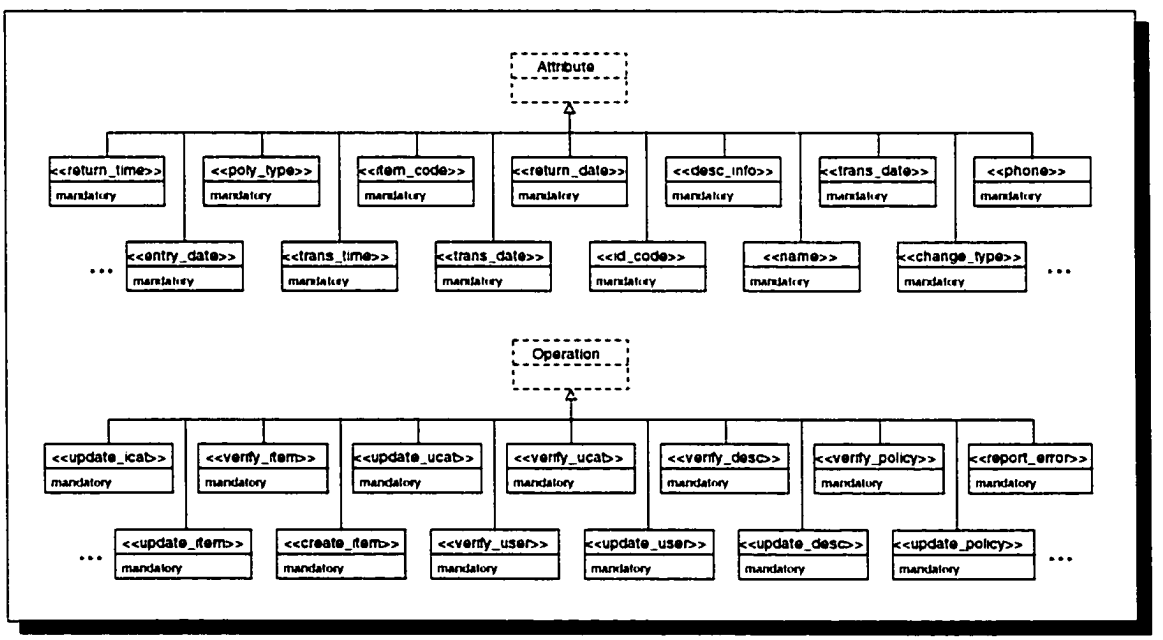

Figure 3.22: Graphical attribute and operation stereotypes

3.5 The Domain Meta-Models

In DSSEE a set of meta-models are created for each of the SCM, DCM, and domain UML models that have been created. These meta models are specifically for the *Language Engi-*

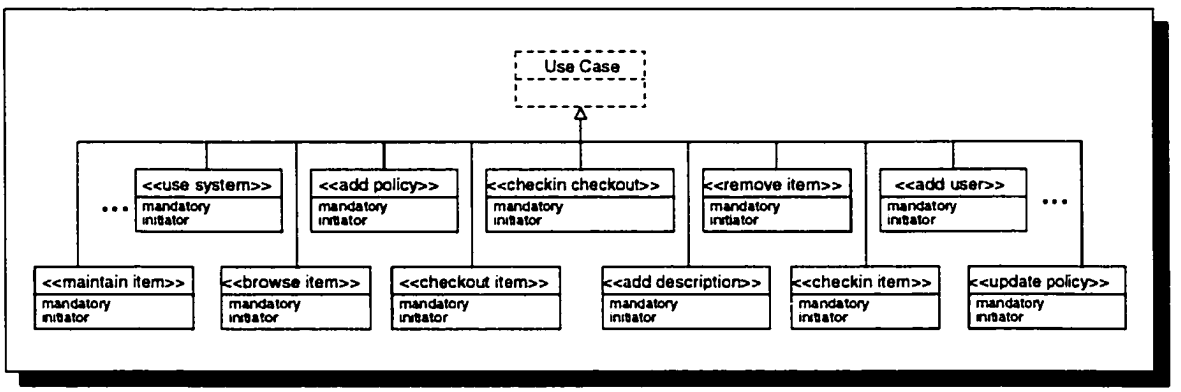

Figure 3.23: Graphical use case stereotypes

neering process, and are not intended to be explicitly used in *Application Engineering*. The meta-models are intended to be used by tool and language developers. Language developers will use the meta-models in defining the syntax of the language, and tool developers will use them to facilitate automatic generation of application model components within the tool environment. Without a CASE tool facility that has incorporated the DSML an application engineer may work directly with the domain models to develop application models. The meta-models are made up of (1) graphical representations of the domain stereotypes, (2) the associations between the stereotypes, and (3) the multiplicities on the associations. These components of the meta-models must fully comply with the syntactic and semantic definitions of the UML for the respective models and model elements used. The multiplicities in a meta-model stipulates the number of model elements that may appear in an associated application model. This is so because the instances of the meta-model elements (stereotypes) are the stereotype *base class* model elements. An instance of *≪item category≫* (Figure 3.24) is a class of type *≪item category≫*. This is consistent with the principle that the instances of the meta-model (instances of the stereotype) resides at the model level (MOF level M1 of Table 2.2) and the meta-model elements (stereotypes) reside at the meta-model level (MOF level M2 of Table 2.2).

The naming convention taken in presenting the meta-models is to use the underscore character instead of space for attributes, associations, and operation stereotype names. Class names may contain spaces. All stereotype names are presented in lower case letters.

The meta-model for the SCM is illustrated in Figure 3.24. A sub-set of the meta-models for the example *Checkin – Checkout* domain appears in Figures 3.6.12, 3.6.12, 3.6.12, 3.6.12, 3.6.12, and 3.6.12 of Section 3.6.12 at the end of this chapter.

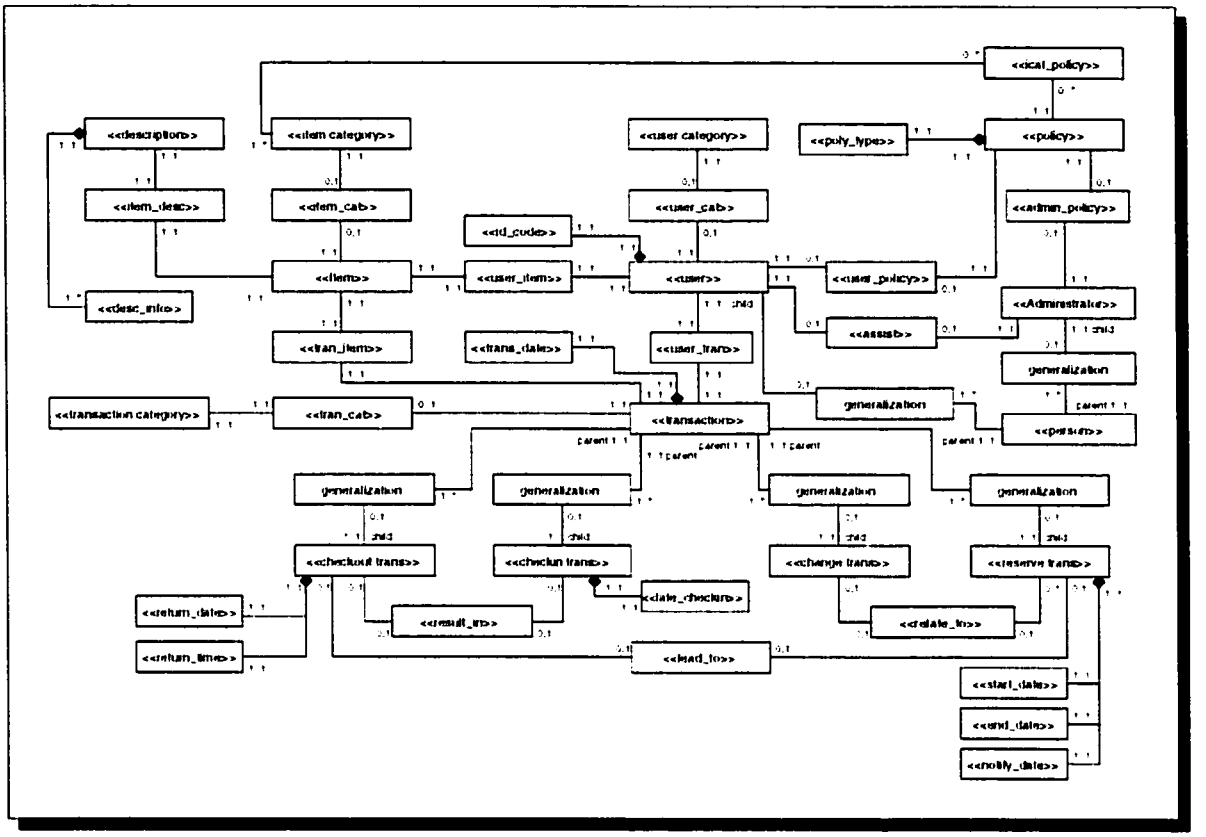

Figure 3.24: SCM meta-model

The SCM meta-model of Figure 3.24 is a graphical *description* of the SCM of Figure 3.3.

3.6 Checkin-Checkout Specification

The following sub-sections contain the complete specification for the example domain *Checkin – Checkout* domain. The specifications for the various domain models were developed under using the process presented in Chapter 4.

3.6.1 Example domain

Some of the main characteristics of a *Checkin-Checkout* application domain are:

- An *administrator* of the system (librarian in the library system);
- A set of *users* who will be associated with the functions of checking in and checking out (renter - customer in the car rental system);
- A set of *items*, with unique identifiers, that will be checked in and/or checked out (videos in the video rental system);
- A *description* of the *items* (title information in a library system);
- A set of *policies* associated with the *administrator*, *items*, *users*, and *Transactions* (age limit of customers in the car rental system);
- A set of *transaction* information (record of a rental receipt in the video rental system);
- A set of *categories* for *users*, *items*, and *transactions* (fictional or non-fictional in the library system);
- The relationships between the above concepts;
- A set of *add*, *remove* and *update* functions for adding, removing, and updating *items*, *descriptions*, *users*, *categories* and *policies*;
- A set of *checkin*, *checkout*, *browse*, *reserve item*, and *change reservation* functions; and
- A *checkin* activity must be preceded by a *checkout* activity.

3.6.2 *Checkin – Checkout* domain dictionary

The domain dictionaries of Table 3.8 and 3.9 contain the names, properties, and examples of the terms used in the domain. The (Table 3.8) is updated during the DA&D activity, as more information becomes available. This would include new classes and additional attributes that are introduced during the design phase. Initially, the SCD would contain only the static concepts from the analysis phase.

Table 3.8: Checkin-Checkout domain static concepts dictionary (SCD)

Term	*Property*	*Example*
user	concrete, mandatory *attribute* identifier: IDENT *relationship* assist, user_cat, user_item, user_tran	customer, borrower
item	concrete, mandatory *relationship* item_cat, item_tran, item_desc, user_item	car, book, video
description	concrete, mandatory *attribute* desc_info: DESC *relationship* item_desc	title information for a book
user category	abstract, mandatory *relationship* user_cat, ucat_policy	categories of customer (preferred, delinquent)
item category	abstract mandatory *relationship* item_cat, icat_policy	categories of cars (economy, full size)
transaction category	abstract, mandatory *relationship* tran_cat, tcat_policy *generalization*	categories of transactions for vehicle rental (daily, weekly rental)
policy	abstract, optional *attribute* poly_type: POLICY *relationship* icat_policy, ucat_policy, tcat_policy, admin_policy	policy of borrow time for category of library users (professors - one year)
transaction	abstract, optional *attribute* trans_date: DATE *relationship* user_tran, item_tran, tran_cat, *generalization*	rent_vehicle, return_vehicle transactions
administrator	concrete, optional *attribute* identifier: IDENT *relationship* admin_policy, assist	librarian, vehicle rental agent
checkout trans	concrete, mandatory *attribute* return_date: DATE *relationship specialization*	borrow a book
checkin trans	concrete, mandatory *attribute* late_return: LATE charge: CHARGE *relationship specialization*	return a video
change trans	concrete, optional *relationship specialization*	change a vehicle reservation
reserve trans	concrete, optional *attribute* start_date: DATE end_date: DATE *relationship specialization*	reserve a book

The DCD would initially contain only the use cases of the domain. As the DA&D activity progresses operations are defined and these operations are added to the DCD.

Table 3.9: Checkin-Checkout domain dynamic concepts dictionary (DCD)

Term	*Property*	*Example*
checkin-checkout	abstract, optional	car rental, library rental systems
use system	abstract, optional	borrow a book, return a video
checkin item	concrete, mandatory *specification* An item that had been previously checked out of the system is checked back in.	return a rented vehicle return a borrowed book
checkout item	concrete, mandatory *specification* An item that is available for checkout is checked out by a user.	rent a vehicle borrow a library book
browse item	concrete, optional *specification* Items may be viewed in various order.	view all books on photography
change reserve	concrete, optional *specification* An existing reservation is updated by the user who made the reservation.	cancel a book reservation
reserve item	concrete, optional *pre − condition* An item that is available for the reservation period is reserved by a user	reserve library book reserve vehicle for rental
maintain category	abstract, optional	add a new category of books
add description	concrete, optional *specification* The description to be added must not exist in the system. Only a system administrator can execute this task.	add a new book to the library add a new car model to the rental
remove item	concrete, optional *specification* The item must exist in the system and only a system administrator can execute this task.	copy of a book is removed from the library a vehicle is removed from the rental fleet

3.6.3 *Checkin – Checkout* static concepts model (SCM)

The domain static concept model (SCM) for the *Checkin – Checkout* domain (Figure 3.25) captures the relationships between the static concepts of the domain. The attributes presented in this model are the user defined attributes for the domain. These attributes, like the static concepts, are the concepts that are known and are manipulated by the users of the system.

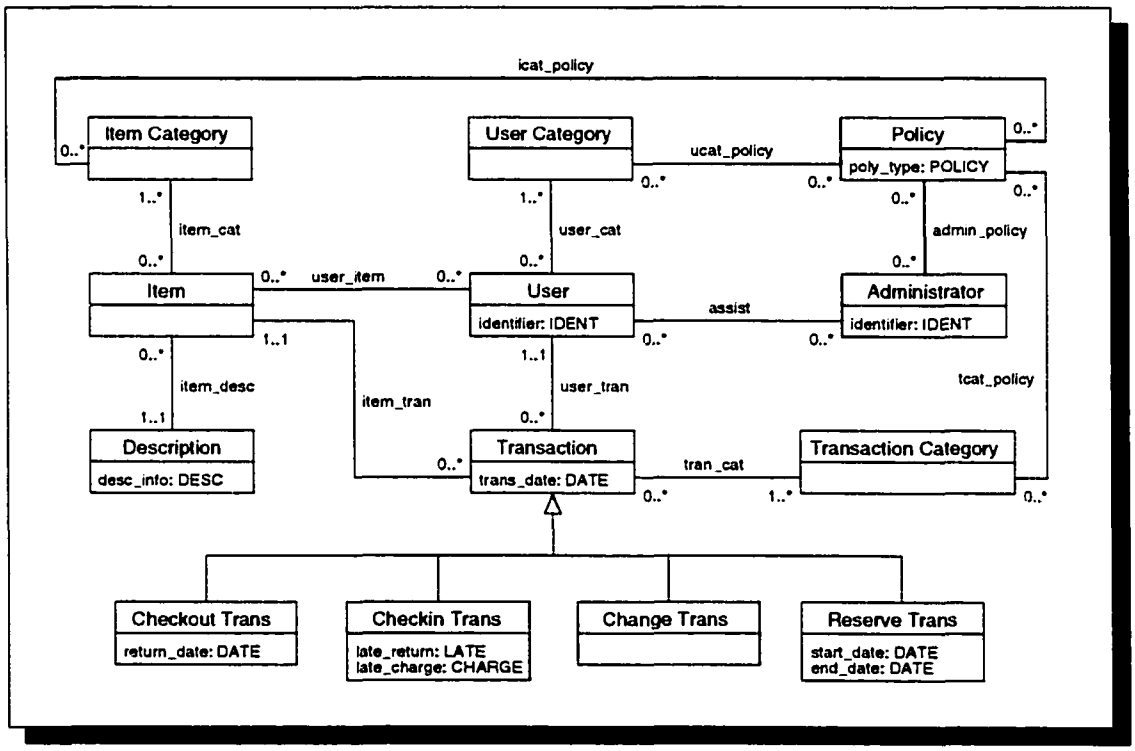

Figure 3.25: Checkin-Checkout domain static concept model (SCM)

3.6.4 *Checkin – Checkout* dynamic concepts model (DCM)

The domain dynamic concept model (DCM) for the *Checkin – Checkout* domain (Figure 3.26 captures the relationships between the dynamic concepts of the domain and the relationships between the users (actors) and the concepts. The dynamic concepts are the functions that are executed by the users of the system.

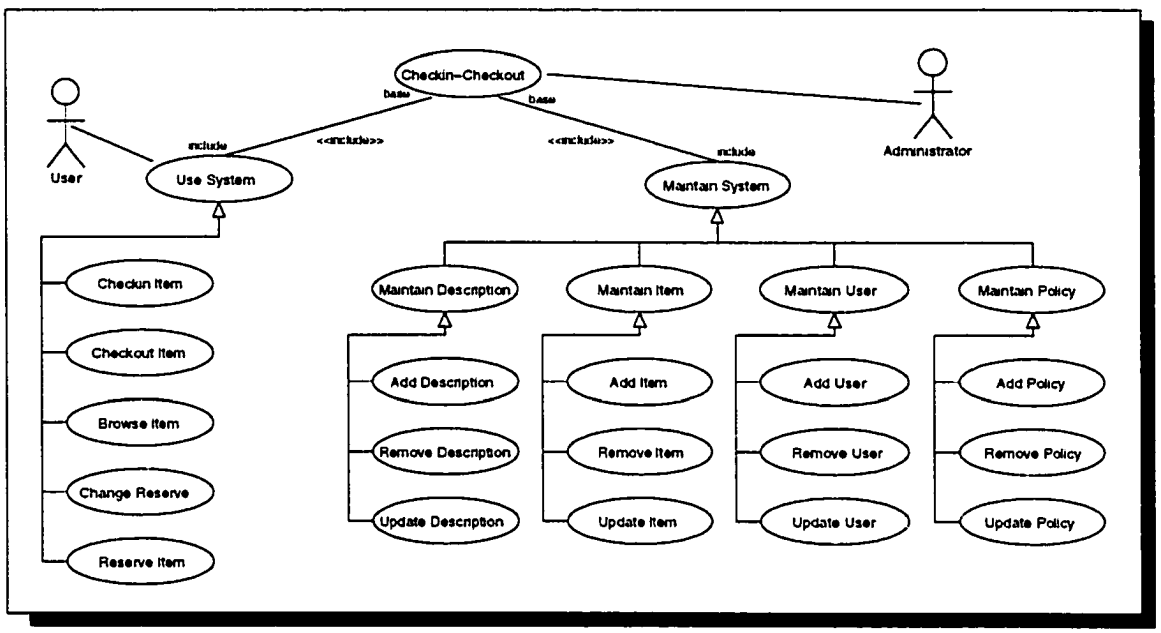

Figure 3.26: Checkin-Checkout domain dynamic concepts model (DCM)

3.6.5 *Checkin – Checkout* concepts context model (CCM)

The domain concept context model (CCM) for the *Checkin–Checkout* domain (Figure 3.27 captures the relationships between the static and dynamic concepts in the domain.

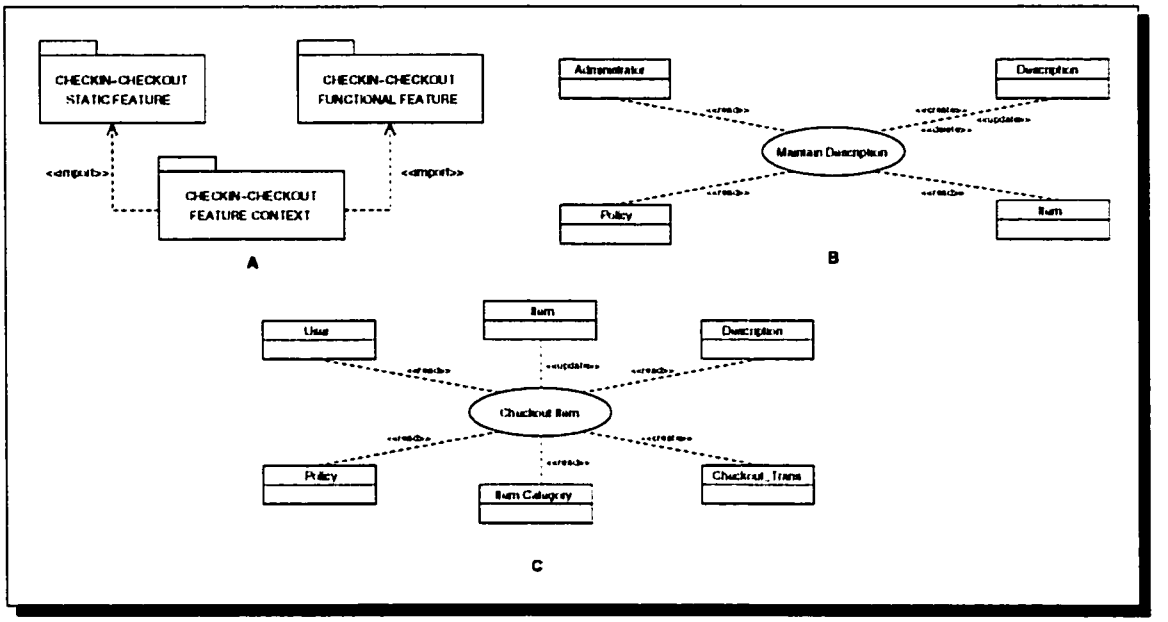

Figure 3.27: Checkin-Checkout concept context model (CCM)

3.6.6 *Checkin – Checkout* use cases

The following Tables 3.10 through 3.15 detail a set of use case specifications for the example *Checkin−Checkout* domain. The use case specifications are obtained from the SCM, DCM, and CCM of Figures 3.25, 3.26, and 3.27, respectively. The *Informal Descriptions* of each use case specification is derived from the *Formal Description*, which was initially obtained from the informally stated requirements for the domain applications.

Table 3.10: Use Case: *Remove Description*

Use Case	Remove Description
Generalization	Maintain Description
Specialization	
Read	Administrator, Policy
Update	
Create	
Delete	Description
Input	desc: Description, admin: Administrator
Output	result: Boolean
Formal description	**context** Remove_Description **inv:** (Description.oclIsKindOf(Class) **and** Administrator.oclIsKindOf(Class) **and** Policy.oclIsKindOf(Class) **and** desc.item→size() = 0 **and** Description→exists(d | d.desc_info@pre = desc.desc_info) **and** admin.policy→exists(p | p.poly_type = 'remove description') **and** Description→forAll(d | d.desc_info <> desc.desc_info) **and** result = TRUE) **xor** (Description.oclIsKindOf(Class) **and** Administrator.oclIsKindOf(Class) **and** desc.item→size() = 0 **and** Description→exists(d | d.desc_info@pre = desc.desc_info) **and** Description→forAll(d | d.desc_info <> desc.desc_info) **and** result = TRUE)
Informal description	Administrator, admin, must have permission to delete the Description **and** desc is the Description to be deleted **and** desc must exist in the system **and** the number of Item objects associated with desc must be zero **and** after execution Description, desc, does not exist in the system **and** result is true. **exclusive or** for a variation with no Policy concept: Administrator, admin can delete the Description **and** desc is the Description to be deleted **and** desc must exist in the system **and** the number of Item objects associated with desc must be zero **and** after execution Description, desc, does not exist in the system **and** result is true.

Table 3.11: Use Case *Add Item*

Use Case	Add Item
Generalization	Maintain Item
Specialization	
Read	Description, Item Category, Administrator, Policy
Update	Item Category
Create	Item
Delete	
Input	admin: Administrator, desc: Description, itm: Item, icat: Item Category
Output	result: Boolean
Formal description	**context** Add_Item **inv:** (Administrator.oclIsKindOf(Class) **and** Item.oclIsKindOf(Class) **and** Description.oclIsKindOf(Class) **and** Policy.oclIsKindOf(Class) **and** Item_Category.oclIsKindOf(Class) **and** Description→exists(d | d = desc) **and** Item_Category→exists(ic | ic = icat) **and** icat.item@pre→forAll(i | i <> itm) **and** admin.policy→exists(p: Policy | p.poly_type = 'add item') desc.item→exists(i | i = itm) **and** result' = TRUE) **xor** (Administrator.oclIsKindOf(Class) **and** Item.oclIsKindOf(Class) **and** Description.oclIsKindOf(Class) **and** Item_Category.oclIsKindOf(Class) **and** Description→exists(d | d = desc) **and** Item_Category→exists(ic | ic = icat) **and** icat.item@pre→forAll(i | i <> itm) **and** desc.item→exists(i | i = itm) **and** result' = TRUE)
Informal description	The Description (desc), of the Item (item) must exist in the system **and** the Item Category (icat) for itm must exist **and** itm must not already exist in the system **and** the Administrator (admin) must have permission to add itm **and** after execution itm must exist in the system **and** result is true. **exclusive or** a variation with no Policy: The Description (desc), of the Item (itm) must exist in the system **and** the Item Category (icat) for itm must exist **and** itm must not already exist in the system **and** the Administrator (admin) can add itm **and** after execution itm must exist in the system **and** result is true.

Table 3.12: Use Case *Add User*

Use Case	Add User
Generalization	Maintain User
Specialization	
Read	Administrator, Policy
Update	User Category
Create	User
Delete	
Input	ucat: User Category, usr: User, admin: Administrator
Output	result: Boolean
Formal description	(**context** Create_User **inv:** User.oclIsKindOf(Class) **and** Administrator.oclIsKindOf(Class) **and** Policy.oclIsKindOf(Class) **and** User_Category.oclIsKindOf(Class) **and** User_Category→exists(uc | uc = ucat) **and** ucat.user@pre→forAll(u | u.identifier <> usr.identifier) **and** admin.policy→exists(p: Policy | p.poly_type = 'add user') **and** ucat.user→exists(u: User | u.identifier = usr.identifier) **and** result = TRUE) **xor** (User.oclIsKindOf(Class) **and** Administrator.oclIsKindOf(Class) **and** User_Category.oclIsKindOf(Class) **and** User_Category→exists(uc | uc = ucat) **and** ucat.user@pre→forAll(u | u.identifier <> usr.identifier) **and** ucat.user→exists(u: User | u.identifier = usr.identifier) **and** result = TRUE)
Informal description	The User Category (ucat) must exist **and** the User (usr) to be added must not already exist in the system **and** the Administrator (admin) must have permission to add the User **and** at the end of execution usr must be in the system and **and** result is TRUE. **exclusive or** variation with no Policy The User Category (ucat) must exist **and** the User (usr) to be added must not already exist in the system **and** at the end of execution usr must be in the system and **and** result is TRUE.

Table 3.13: Use Case *Update Policy*

Use Case	Update Policy
Generalization	Maintain User
Specialization	
Read	Administrator, Policy
Update	Policy
Create	
Delete	
Input	opoly, npoly: Policy, admin: Administrator
Output	result: Boolean
Formal description	**context** Update_Policy **inv:** Administrator.oclIsKindOf(Class) **and** policy.oclIsKindOf(Class) **and** Policy→exists(p@pre | p@pre = opoly) **and** Policy→forAll(p | p <> npoly) **and** admin.policy→exists(p: Policy | p.poly_type = 'update policy') **and** Policy→forAll(p | p <> opoly) **and** Policy→exists(p: Policy | p = npoly) **and** result = TRUE
Informal description	The Policy (opoly) to be updated must exist **and** the new Policy (npoly) must not exist in the system **and** the Administrator (admin) must have permission to update opoly **and** at the end of execution the updated Policy (npoly) must now be in the system **and** result is TRUE.

Table 3.14: Use Case *Checkout Item*

Use Case	Checkout Item
Generalization	Use System
Specialization	
Read	Description, Policy, Item Category, User, Transaction, Item
Update	
Create	otrans: Checkout Trans
Delete	
Input	usr: User, icat: Item Category, desc: Description
Output	oitm: Item, otrans: Checkout Trans
Formal description	(**context** Checkout_Item **inv:** Checkout_Trans.oclIsKindOf(Class) **and** Policy.oclIsKindOf(Class) **and** Item_Category.oclIsKindOf(Class) **and** Description.oclIsKindOf(Class) **and** User.oclIsKindOf(Class) **and** Item.oclIsKindOf(Class) **and** Transaction.oclIsKindOf(Class) **and** Checkout_Trans.oclIsKindOf(Class) **and** usr.user_category.policy→exists(p: Policy | p.poly_type = 'check out') **and** desc.item@pre→intersection(icat.item@pre)→exists(i: Item | (i.checkout_trans@pre→size() = i.checkin_trans@pre→size()) **and** (i.reserve_trans→exists(r: Reserve_Trans | r.start_date > *current date*) **or** i.reserve_trans@pre→size() = 0)) **and** desc.item.item_category.policy→exists(p: Policy | p.poly_type = 'check out') **and** desc.item→exists(i: Item | (i = oitm) **and** (oitm.checkout_trans→exists(cout: Checkout_Trans | (otrans = cout) **and** (otrans.trans_date = *current date*) **and** (otrans.return_date ≥ *current date*))) **and** (oitm.checkout_trans→size() = oitm.checkin_trans→size() + 1))) **xor** (**context** Checkout_Item **inv:** Checkout_Trans.oclIsKindOf(Class) **and** Item_Category.oclIsKindOf(Class) **and** Description.oclIsKindOf(Class) **and** User.oclIsKindOf(Class) **and** Item.oclIsKindOf(Class) **and** Transaction.oclIsKindOf(Class) **and** Checkout_Trans.oclIsKindOf(Class) **and** desc.item@pre→intersection(icat.item@pre)→exists(i: Item | (i.checkout_trans@pre→size() = i.checkin_trans@pre→size()) **and** (i.reserve_trans→exists(r: Reserve_Trans | r.start_date > *current date*) **or** i.reserve_trans@pre→size() = 0)) **and** desc.item→exists(i: Item | (i = oitm) **and** (oitm.checkout_trans→exists(cout: Checkout_Trans | (otrans = cout) **and** (otrans.trans_date = *current date*) **and** (otrans.return_date ≥ *current date*))) **and** (oitm.checkout_trans→size() = oitm.checkin_trans→size() + 1)))
Informal description	The User (usr) must have permission to checkout an Item and an Item of Description (desc) must be available for checkout i.e. it must not be currently checked out nor reserved for the current date and the Item to be checked out must be approved for checkout and at the end of execution the Item (oitm) checked out must be associated with the newly created Checkout Trans (otrans) and the number of Checkout Trans associated with oitm must be greater that the number of Checkin Trans associated with oitm. exclusive or variation with no Policy An Item of Description description (desc) must be available for checkout i.e. it must not be currently checked out nor reserved for the current date and at the end of execution the Item (oitm) checked out must be associated with the newly created Checkout Trans (otrans) and the number of Checkout Trans associated with oitm must be greater that the number of Checkin Trans associated with oitm.

Table 3.15: Use Case *Checkin Item*

Use Case	Checkin Item
Generalization	Use System
Specialization	
Read	Item
Update	
Create	Checkin Trans
Delete	
Input	itm: Item
Output	itrans: Checkin Trans
Formal description	**context** Checkin_Item **inv:** Checkin_Trans.oclIsKindOf(Class) **and** Item.oclIsKindOf(Class) **and** Item→exists(i: Item | i = itm **and** itm.checkout_trans@pre→size() = itm.checkin_trans@pre→size() + 1) **and** itm.checkin_trans→exists(cin: Checkin_Trans | cin = itrans **and** itrans.trans_date = *current_date*) **and** itm.checkout_trans→size() = item.checkin_trans→size()
Informal	itm is the Item object to be checked in **and** it must exist in the system **and** the number of Checkout Trans objects associated with itm must be one more than the number of Checkin Trans **and** at the end of the execution there must be a Checkin Trans associated with itm for which the trans_date is equal to the *current date* **and** the number of Checkin Trans and Checkout Trans objects associated with itm are equal.

3.6.7 *Checkin – Checkout* domain class diagram

The analysis level CD for the example *Checkin – Checkout* domain (Figure 3.28) was obtained from the SCM (Figure 3.25) by mapping the static concepts to UML model elements and adding analysis level attributes.

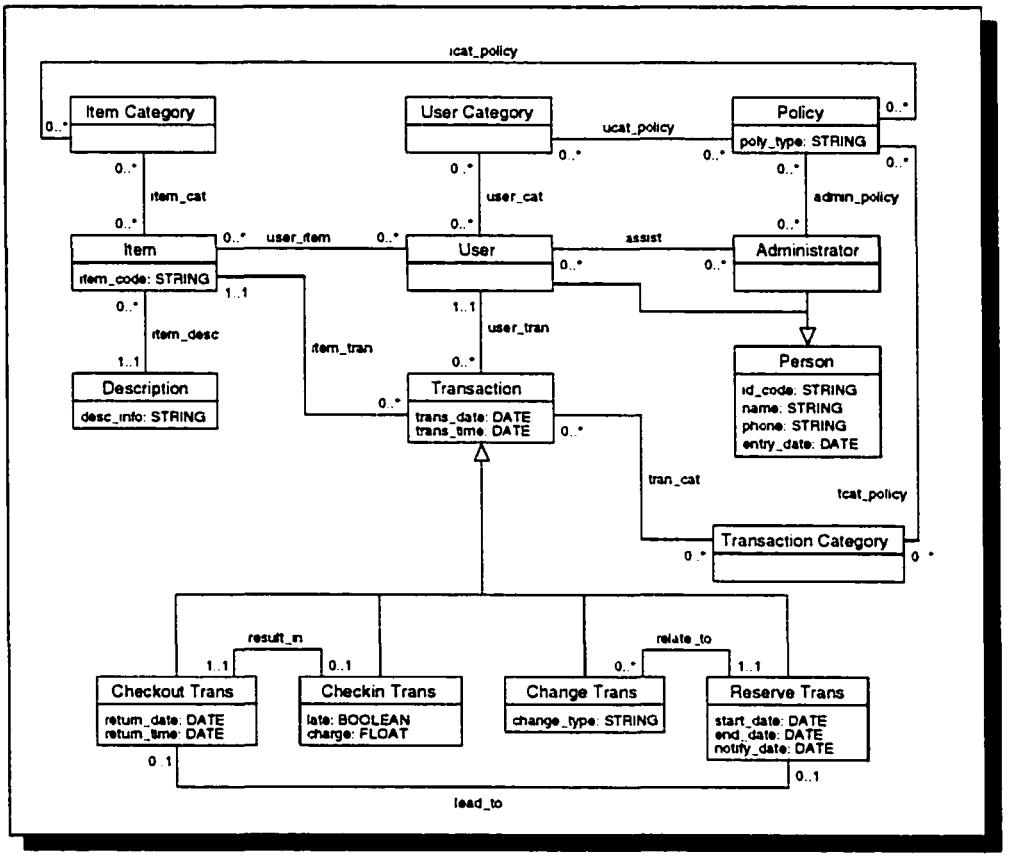

Figure 3.28: *Checkin – Checkout* domain CD model

The design level CD of Figure 3.28 is refined from the analysis level CD (Figure 3.28) by the addition of design level classes, attributes, and operations.

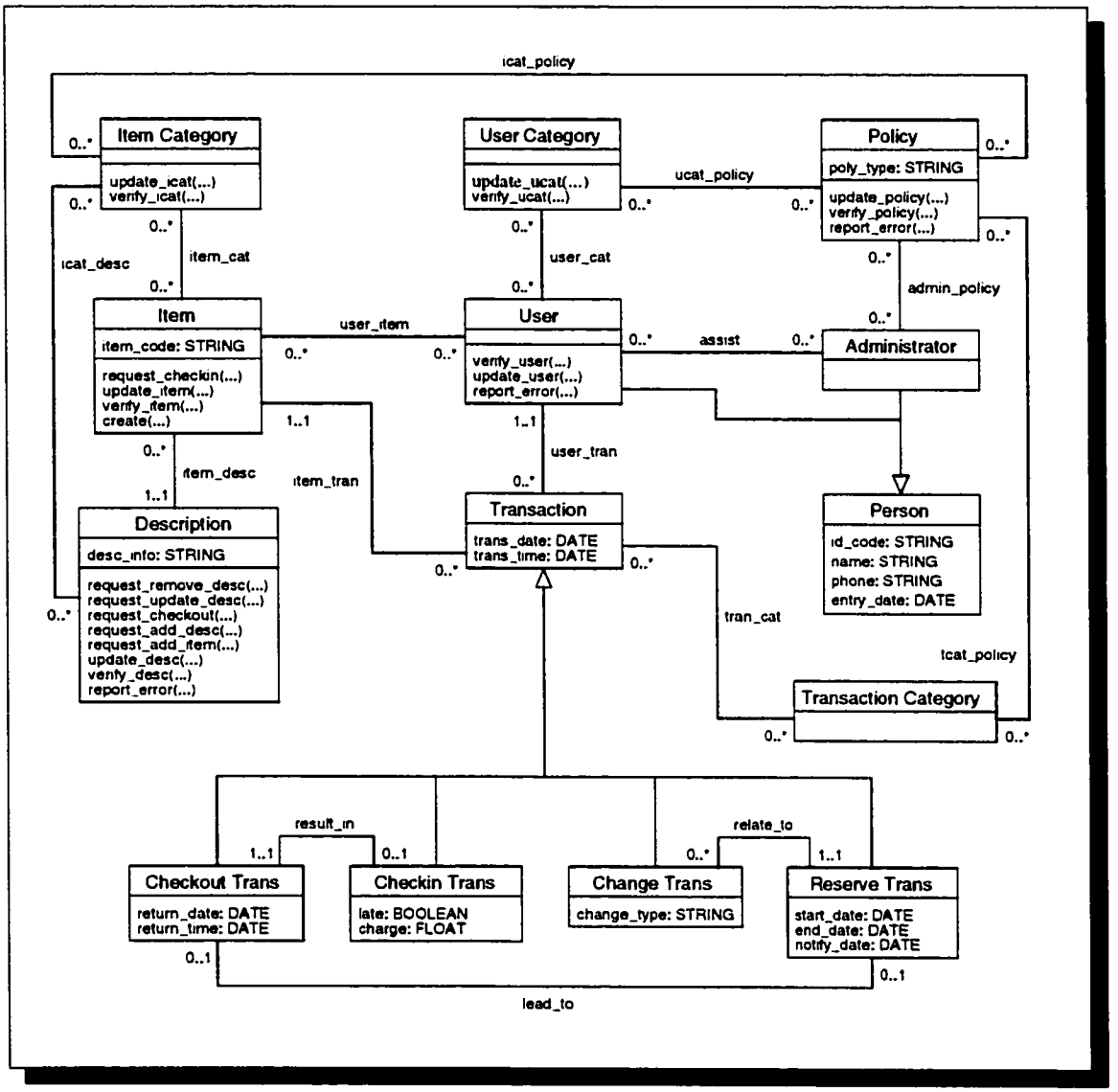

Figure 3.29: *Checkin – Checkout* domain design CD model

3.6.8 *Checkin – Checkout* domain activity diagrams

Figures 3.30**A** through 3.32**B** is a sub-set of the activity diagrams that may be created for the *Checkin – Checkout* domain.

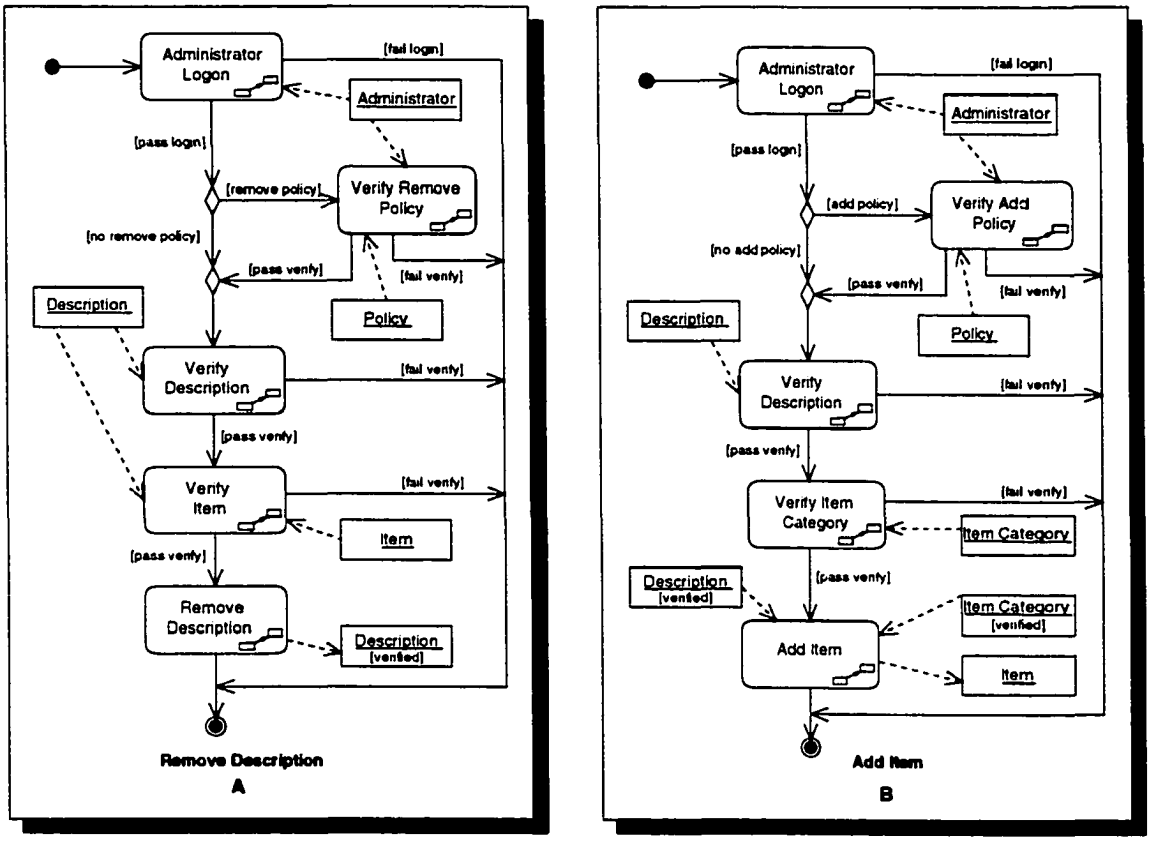

Figure 3.30: Activity diagrams - Remove Description & Add Item

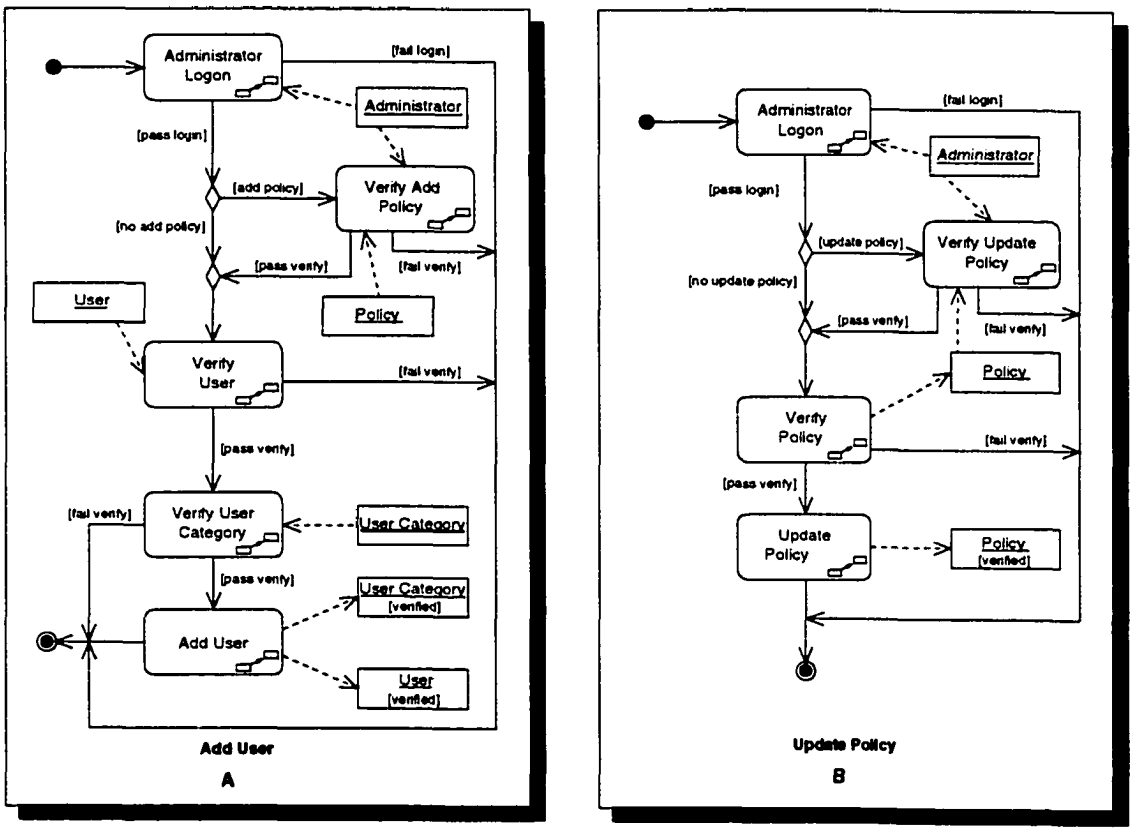

Figure 3.31: Activity diagrams - Add User & Update Policy

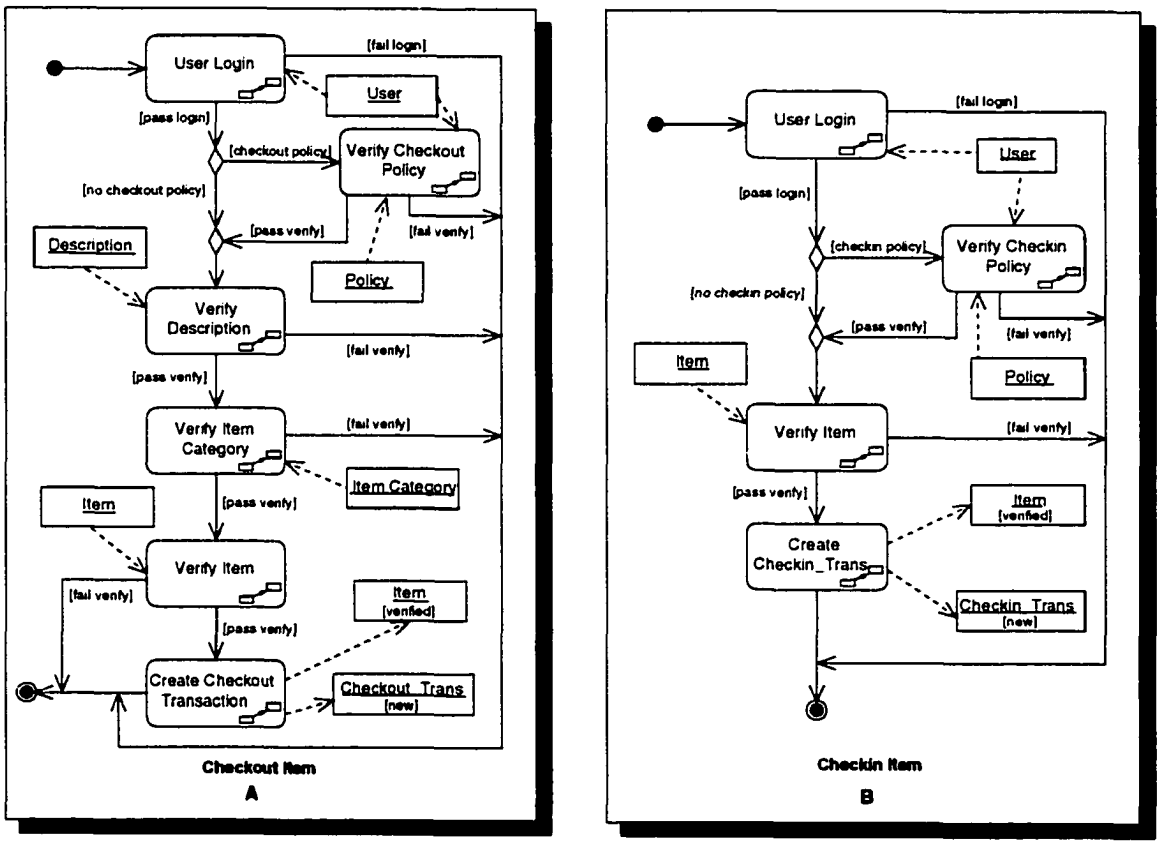

Figure 3.32: Activity diagrams - Checkout Item & Checkin Item

3.6.9 *Checkin – Checkout* domain collaboration diagrams

Figures 3.33 through 3.38 is a sub-set of the collaboration diagrams that may be created for the *Checkin – Checkout* domain.

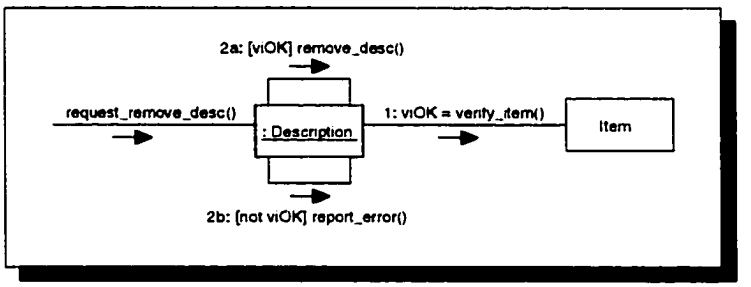

Figure 3.33: Collaboration diagram - Remove Description

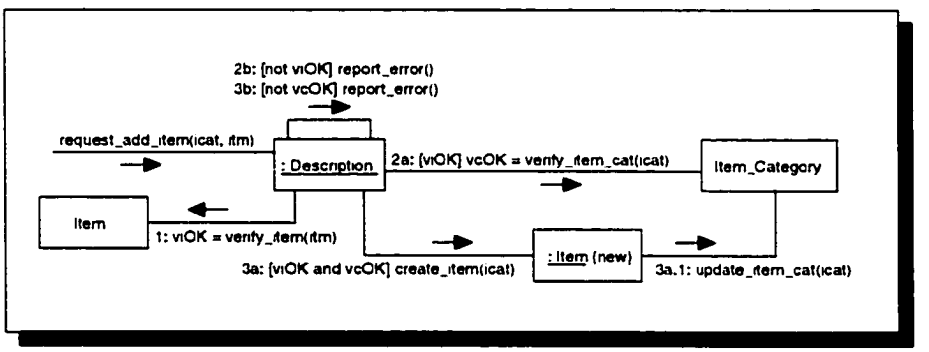

Figure 3.34: Collaboration diagram - Add Item

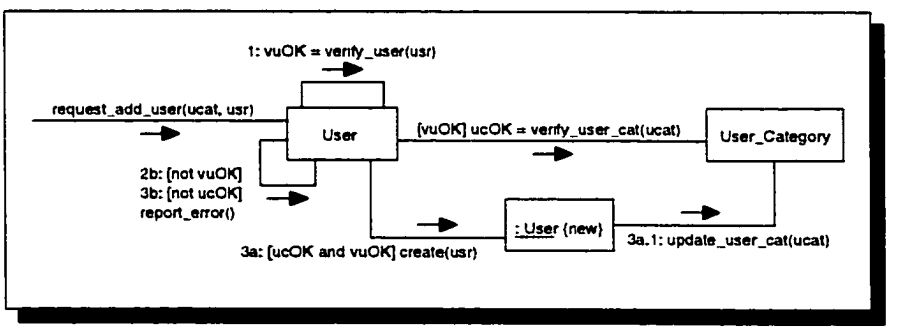

Figure 3.35: Collaboration diagram - Add User

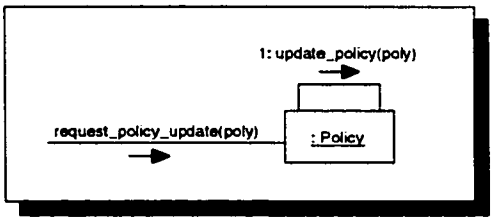

Figure 3.36: Collaboration diagram - Update Policy

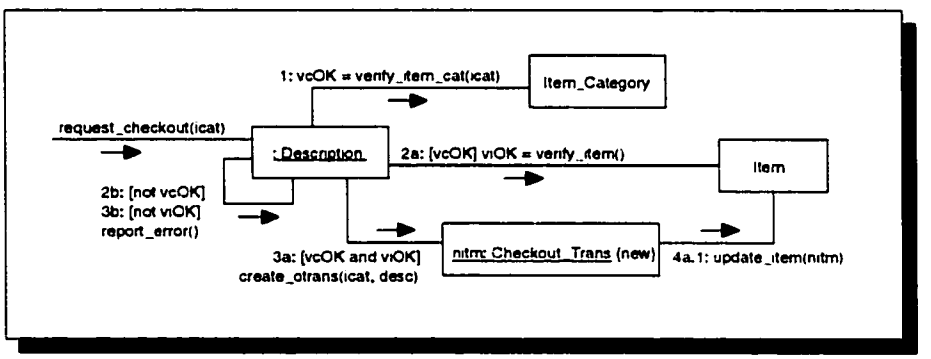

Figure 3.37: Collaboration diagram - Checkout Item

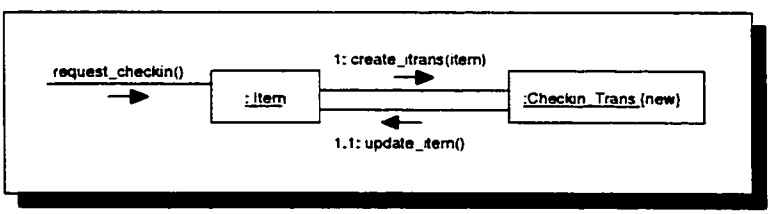

Figure 3.38: Collaboration diagram - Checkin Item

3.6.10 *Checkin – Checkout* domain sequence diagrams

Figures 3.28 through 3.44 is a sub-set of the sequence diagrams that may be created for the *Checkin – Checkout* domain.

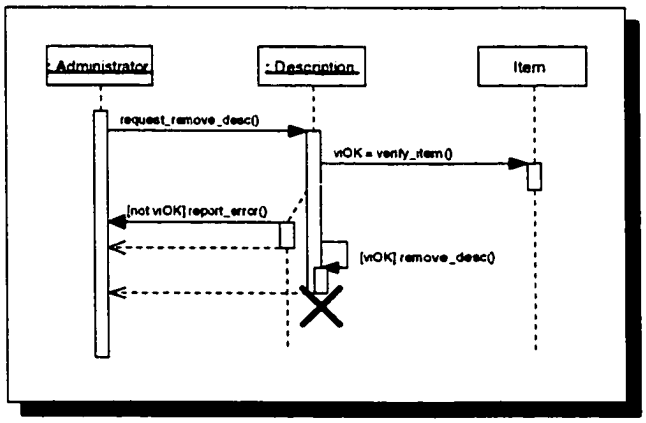

Figure 3.39: Sequence diagram - Remove Description

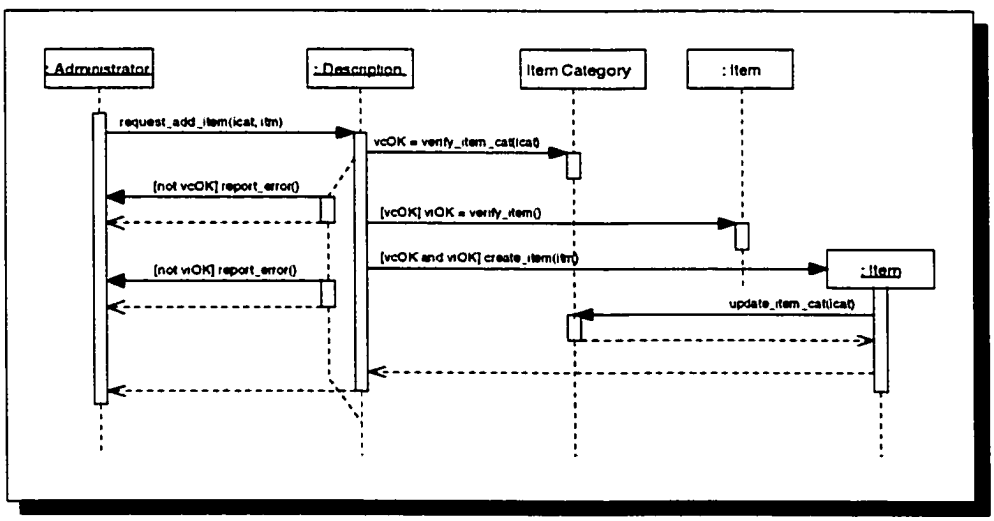

Figure 3.40: Sequence diagram - Add Item

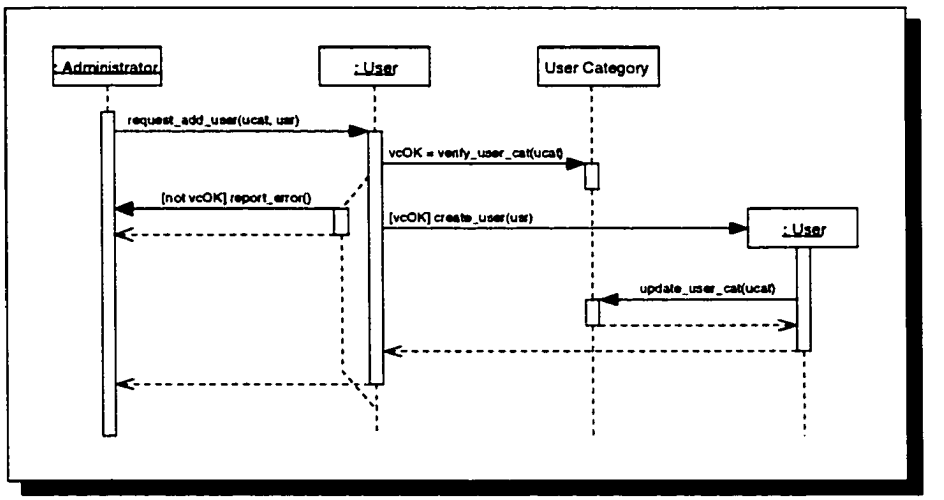

Figure 3.41: Sequence diagram - Add User

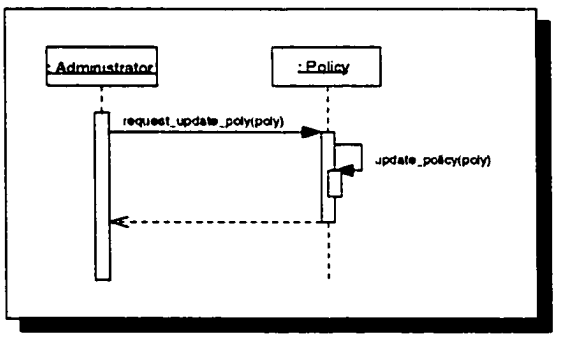

Figure 3.42: Sequence diagram - Update Policy

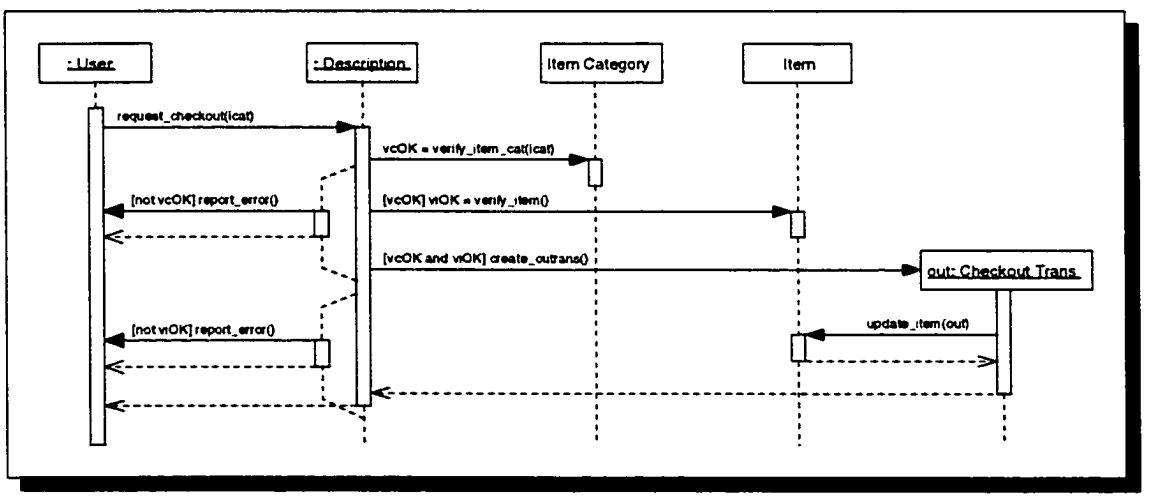

Figure 3.43: Sequence diagram - Checkout Item

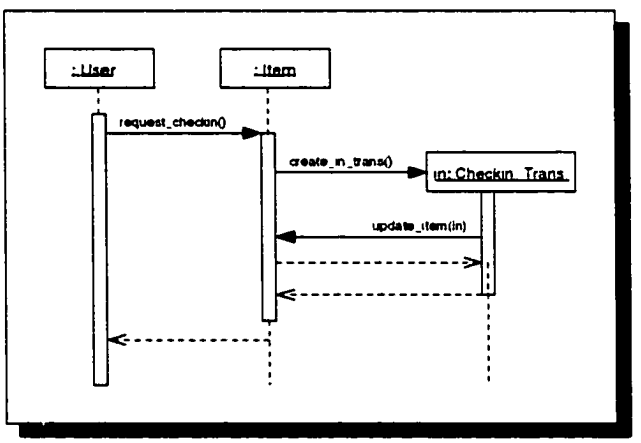

Figure 3.44: Sequence diagram - Checkin Item

3.6.11 *Checkin – Checkout* domain stereotypes

Tables 3.16 through 3.21 is a sub-set of the stereotypes that are created for the *Checkin – Checkout* domain.

Table 3.16: Operation *Verify Item* stereotype

Stereotype	verify item
Base Class	Operation
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** verify_item **inv:** self.ownerScope = classifier **and** [Operation's scope] self.visibility = public **and** [Operation's default value] self.isQuery = true **and** [Operation does not change state] self.isAbstract = false **and** [Operation is not abstract] self.isLeaf = false **and** [Implementation may be modified by descendant class] self.isRoot = false **and** [Operation can inherit declaration of the same name] self.mandatory = false **and** [Operation is optional in a domain application] ((self **implies** (exist()) **and** (exists() **implies** self)) **and** [Operation is equivalent to the OCL collection operation exists()] collaboration_diagram::remove_description→includes(self) **and** [A remove description collaboration includes an verify item operation] sequence_diagram::remove_description→includes(self) **and** [A remove description sequence diagram includes the verify item operation] collaboration_diagram::checkout_item→includes(self) **and** [A checkout item collaboration includes a verify item operation] sequence_diagram::checkout_item→includes(self) [A checkout item sequence diagram includes the verify item operation] collaboration_diagram::add_item→includes(self) **and** [An add item collaboration includes a verify item operation] sequence_diagram::add_item→includes(self) [A add item sequence diagram includes the verify item operation]
Description	The *constraint* specifies the meta-attributes, and context of the stereotyped operation.

Table 3.17: Operation *Create Item* stereotype

Stereotype	create item
Base Class	Operation
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** create_item **inv:** self.ownerScope = instance **and** [Operation's scope] self.visibility = private **and** [Operation's default value] self.isQuery = false **and** [Operation may change state] self.isAbstract = false **and** [Operation is not abstract] self.isLeaf = false **and** [Implementation may be modified by descendant class] self.isRoot = false **and** [Operation can inherit declaration of the same name] self.mandatory = true **and** [Operation is not optional in a domain application] self.parameter = 'icat: Item Category' **and** [Parameter of the stereotyped operation] item_category→exists(ic | ic = icat) **and** [The parameter must be known to the system] ((self **implies** (union()) **and** (union() **implies** self)) **and** [Operation is equivalent to the OCL collection operation union()] collaboration_diagram::add_item→includes(self) **and** [An add item collaboration includes an verify item operation] sequence_diagram::add_item→includes(self) [An add item sequence diagram includes the verify item operation]
Description	The *constraint* specifies the meta-attributes, and context of the stereotyped operation.

Table 3.18: Class *Description* stereotype

Stereotype	description
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** $\ll$ *description* $\gg$ **inv:** self.isAbstract = false **and** [Stereotyped classes may have instances] self.isLeaf = true **and** [Stereotyped classes may not have descendants] self.isRoot = true **and** [Stereotyped classes may not have ancestors] self.mandatory = true **and** [Stereotyped classes must be in domain applications] self.$\ll$ *desc_info* $\gg$→size() $\geq$ 1 **and** [Stereotype class has one or more desc info] self.$\ll$ *desc_info* $\gg$.visibility = private **and** [Stereotype class desc_info attributes is private] self.$\ll$ *item* $\gg$→size() = 1 **and** [Description class has one Item class] self.$\ll$ *item_category* $\gg$→size $\geq$ 0 **and** [Description class has zero or more Item Category class] self.operation→includesAll(request_checkout, request_update_desc, report_error, request_add_desc, request_add_item, update_desc, request_remove_desc, verify_desc) [Operations associated with the stereotyped class]
Description	The *constraint* specify the meta-attribute, meta-attributes and multiplicities of the classes that an Description stereotyped class is associated with.

Table 3.19: Attribute *Desc Info* stereotype

Stereotype	desc info
Base Class	Attribute
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** $\ll descinfo \gg$ **inv:** self.ownerScope = instance **and** [Operation's scope] self.visibility = private **and** [Operation's default value] self.targetScope = instance **and** [Operation's target scope] self.changeability = true **and** [Stereotype attribute value may be changed] self.multiplicity = 1..1 **and** [Stereotype stores one value] self.type = String **and** [Stereotype is of type String] self.mandatory = true **and** [Stereotype must be included in domain applications] self.persistence = transitory **and** [Stereotype is destroyed if instance is destroyed] (request_update_desc **or** request_add_desc **or** request_remove_desc **or** update_desc **or** verify_desc) **implies** self [Operations associated with the stereotype]
Description	The *constraint* specify the meta-attribute, meta-attributes and multiplicities of the classes that an Description stereotyped class is associated with.

Table 3.20: Association *Item Desc* stereotype

Stereotype	item_desc
Base Class	Association
Parent	N/A
Tags	none
Constraints	**context** $\ll item\_desc \gg$ **inv:** (self.AssociationEnd.participant.type = $\ll item \gg$ **or** self.AssociationEnd.participant.type = $\ll description \gg$) **and** [The stereotype may only associate Item and Description classes] self.AssociationEnd—forAll(ae: AssociationEnd | ae.participant.type=$\ll item \gg$ **implies** ae.multiplicity = 0..*) **and** [The multiplicities of the association end Item] self.AssociationEnd—forAll(ae: AssociationEnd | ae.participant.type = $\ll description \gg$ **implies** ae.multiplicity = 1..1) [The multiplicities of the association end Description]
Description	The *constraint* specifies the stereotyped classes that this association may have.

For the use case stereotype of Table 3.21 the constraint is taken almost verbatim from the *Checkout Item* use case specification, see Table 3.12. In order to reuse the use case formal description as the constraint for its stereotype definition the operations of the use case are identified, and the constraint implied from the operations' specification, i.e. the operations must preserve the static constraints of the stereotyped use case. The tag definition **actor** is introduced to identify the type of the actor that associates with use cases in the domain.

Table 3.21: Use Case *Checkout Item* stereotype

Stereotype	checkout item
Base Class	Use Case
Parent	none
Tag	initiator, mandatory
Representation	default
Constraint	(**context** checkout_item **inv:** self.initiator = user **and** self.mandatory = true **and** ((self **implies** excluding()) **and** (excluding() **implies** self)) **and** self.operation→includesAll(request_checkout, verify_item_cat, verify_item, create_otrans, update_item, report_error) **and** usr.user_category.policy→exists(p: policy | p.poly_type = 'check out') **and** desc.item@pre→intersection(icat.item@pre)→exists(i: Item | (i.checkout_trans@pre→size() = i.checkin_trans@pre→size()) **and** (i.reserve_trans→forAll(r: reserve_trans | r.start_date > *current date*) **or** i.reserve_trans@pre→size() = 0)) **and** desc.item.item_category.policy→exists(p: policy | p.poly_type = 'check out') **and** desc.item→exists(i: Item | (i = oitm) **and** (oitm.checkout_trans→exists(cout: checkout_trans | (otrans = cout) **and** (otrans.trans_date = *current date*) **and** (otrans.return_date ≥ *current date*))) **and** (oitm.checkout_trans→size() = oitm.checkin_trans→size() + 1))) **xor** (**context** checkout_item **inv:** self.initiator = user **and** self.mandatory = true **true** ((self **implies** excluding()) **and** (excluding() **implies** self)) **and** self.operation→includesAll(request_checkout, verify_item_cat, verify_item, create_otrans, update_item, report_error) **and** desc.item@pre→intersection(icat.item@pre)→exists(i: Item | (i.checkout_trans@pre→size() = i.checkin_trans@pre→size()) **and** (i.reserve_trans→forAll(r: reserve_trans | r.start_date > *current date*) **or** i.reserve_trans@pre→size() = 0)) **and** desc.item→exists(i: Item | (i = oitm) **and** (oitm.checkout_trans→exists(cout: Checkout_Trans | (otrans = cout) **and** (otrans.trans_date = *current date*) **and** (otrans.return_date ≥ *current date*))) **and** (oitm.checkout_trans→size() = oitm.checkin_trans→size() + 1)))
Description	Use case is equivalent to a OCL excluding operation **and** actor is of type user **and** The User (usr) must have permission to checkout an Item **and** an Item of Description (desc) must be available for checkout i.e. it must not be currently checked out nor reserved for the current date **and** the Item to be checked out must be approved for checkout **and** at the end of execution the Item (oitm) checked out must be associated with the newly created Checkout Trans (otrans) and the number of Checkout Trans associated with oitm must be greater that the number of Checkin Trans associated with oitm. **exclusive or** variation with no Policy Use case is equivalent to a OCL excluding operation **and** actor is of type user **and** An Item of Description (desc) must be available for checkout i.e. it must not be currently checked out nor reserved for the current date **and** at the end of execution the Item (oitm) checked out must be associated with the newly created Checkout Trans (otrans) and the number of Checkout Trans associated with oitm must be greater that the number of Checkin Trans associated with oitm.

Table 3.22: Tag definition *Initator*

Tag	initiator
Stereotype	checkout_item
Type	UML::Foundation::Class
Multiplicity	1
Description	Actor for use case

3.6.12 The *Checkin – Checkout* domain meta-models

The meta-models of Figures 3.6.12.1, 3.6.12.2, 3.6.12.3, 3.6.12.4, 3.6.12.5, and 3.6.12.6 are descriptions of the domain models of Figures 3.25, 3.26, 3.27, 3.29, 3.32, and 3.37 respectively.

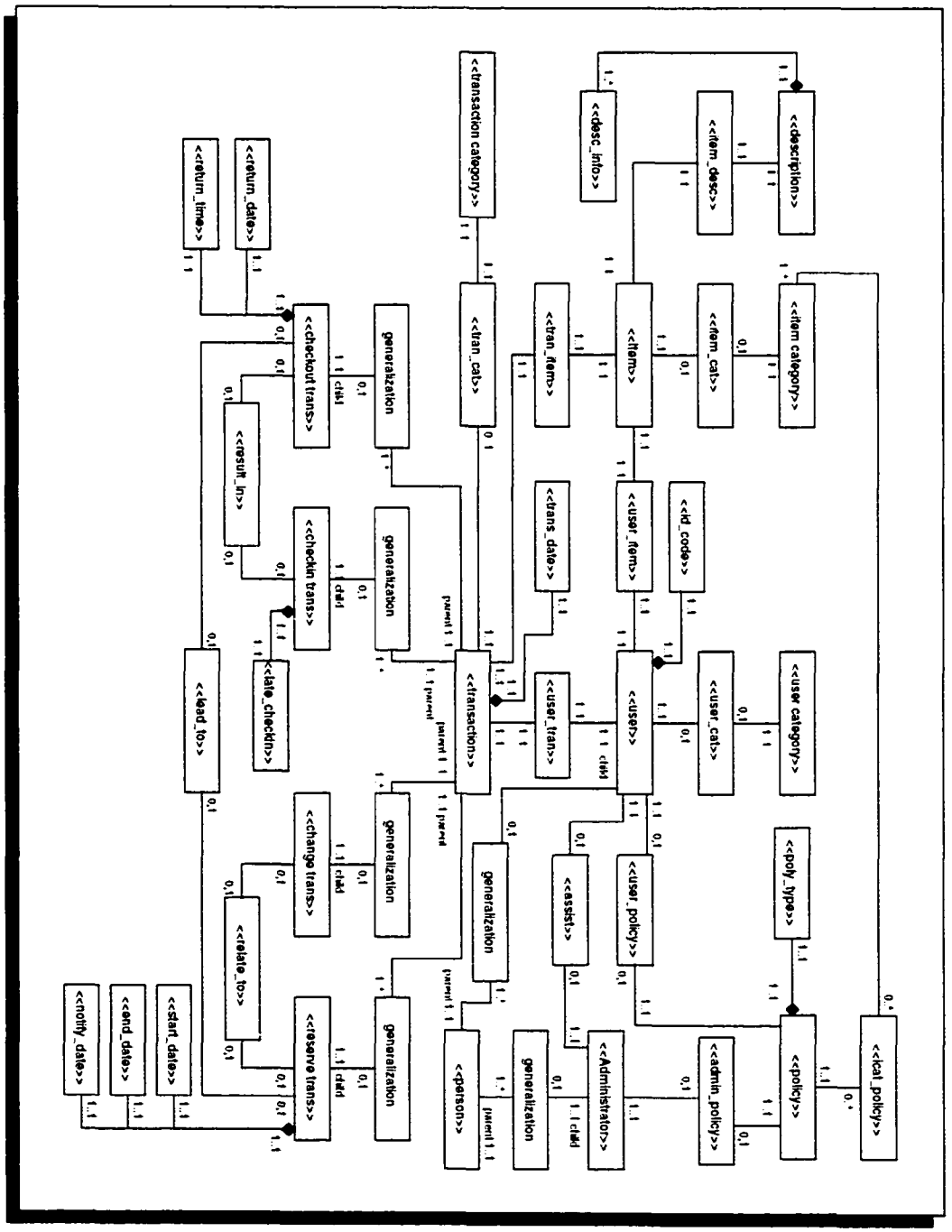

Figure 3.6.12.1: Checkin–Checkout SCM meta–model

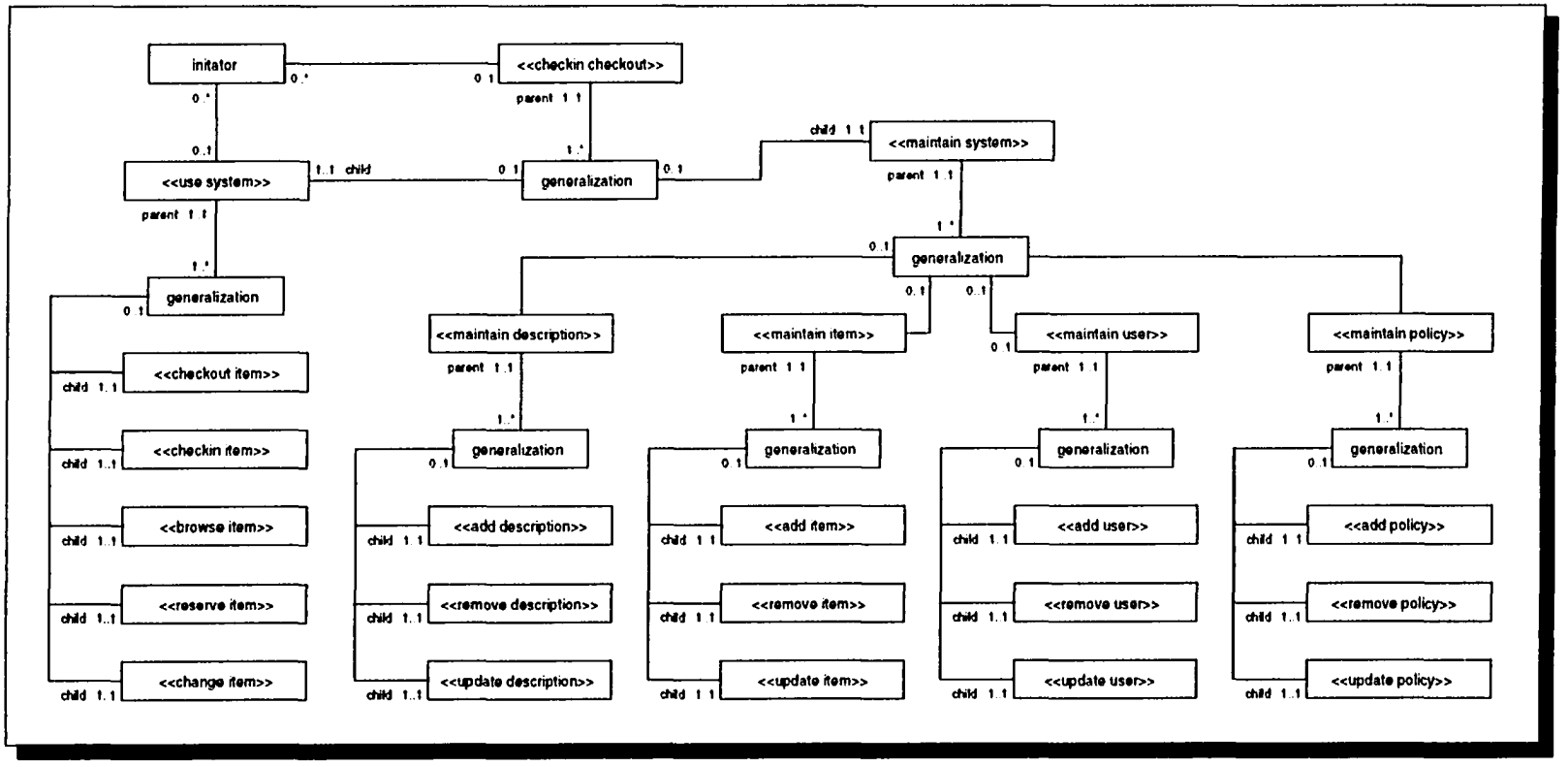

Figure 3.6.12.2: Checkin–Checkout DCM meta–model

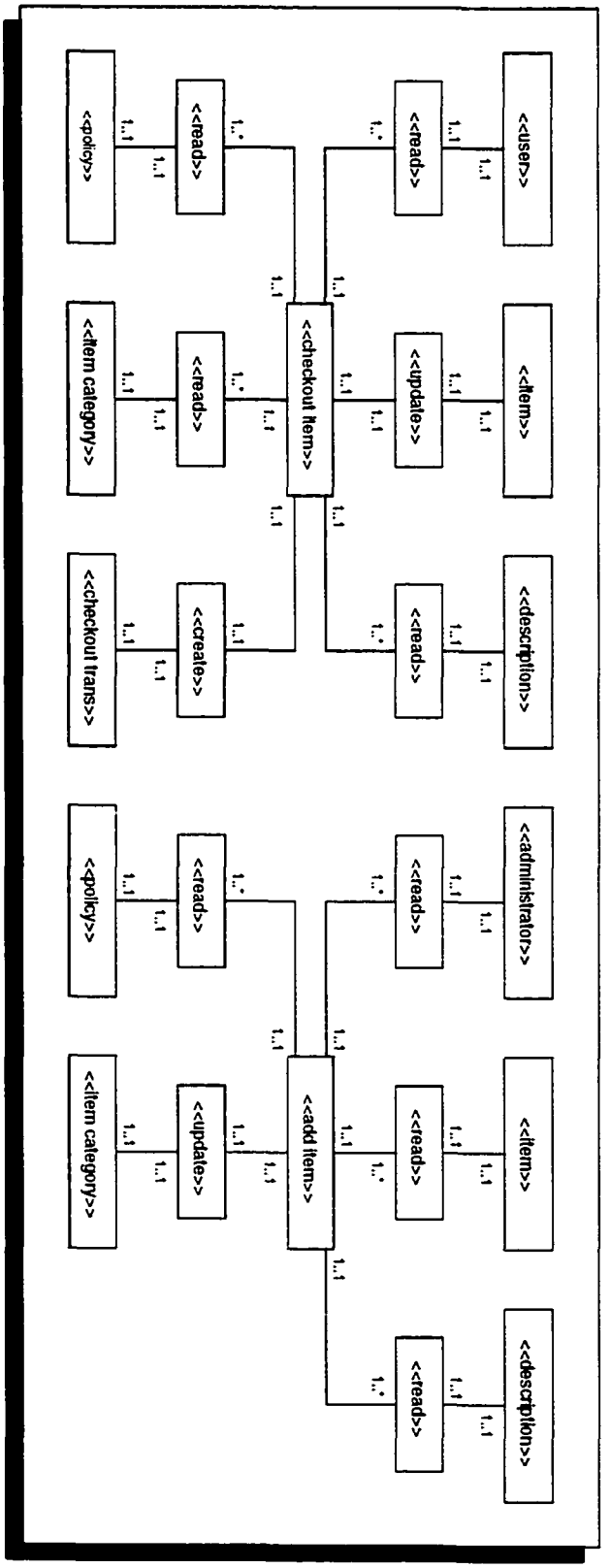

Figure 3.6.12.3: Checkin–Checkout CCM meta–model

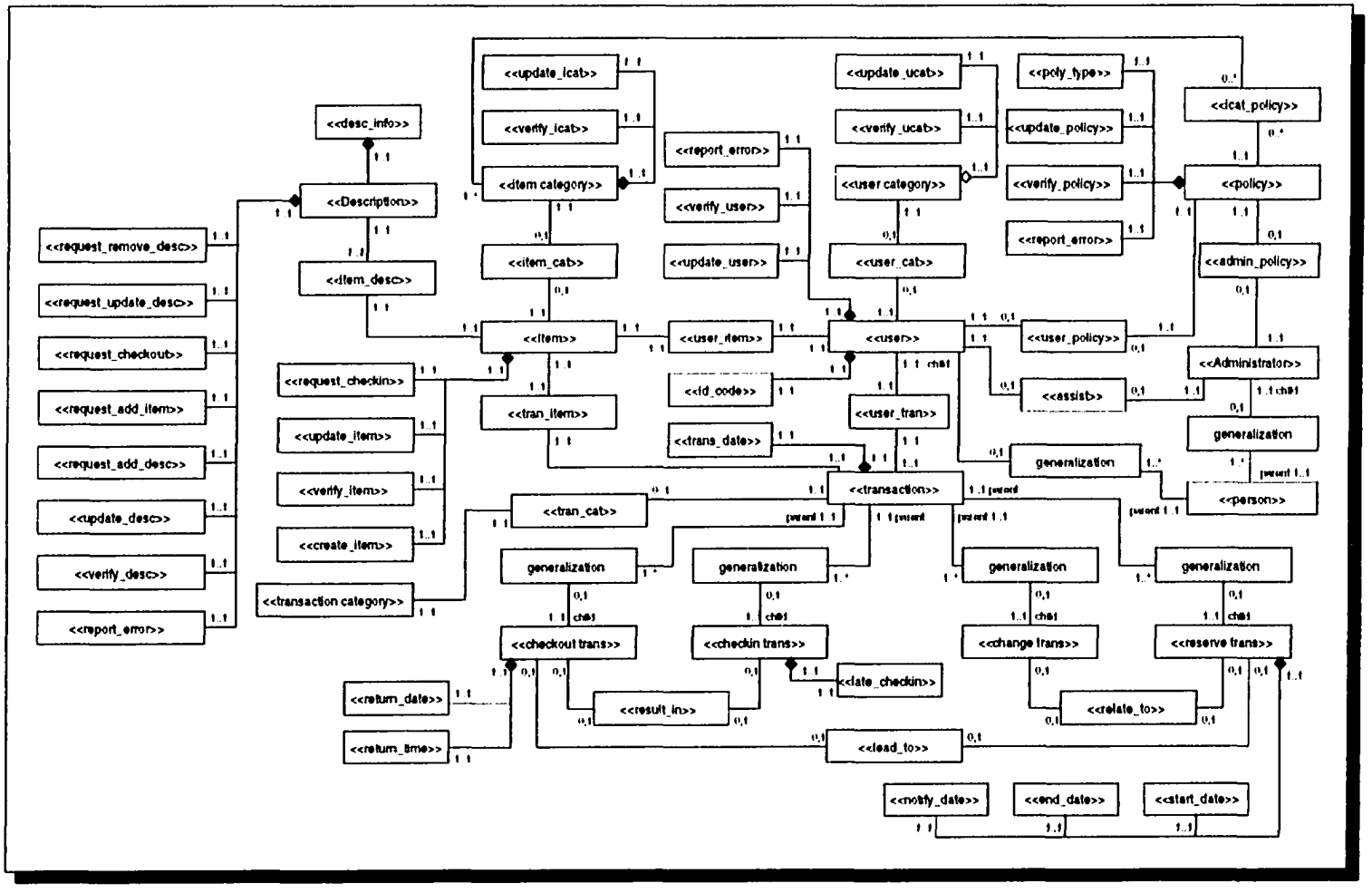

Figure 3.6.12.4: Checkin–Checkout CD meta–model

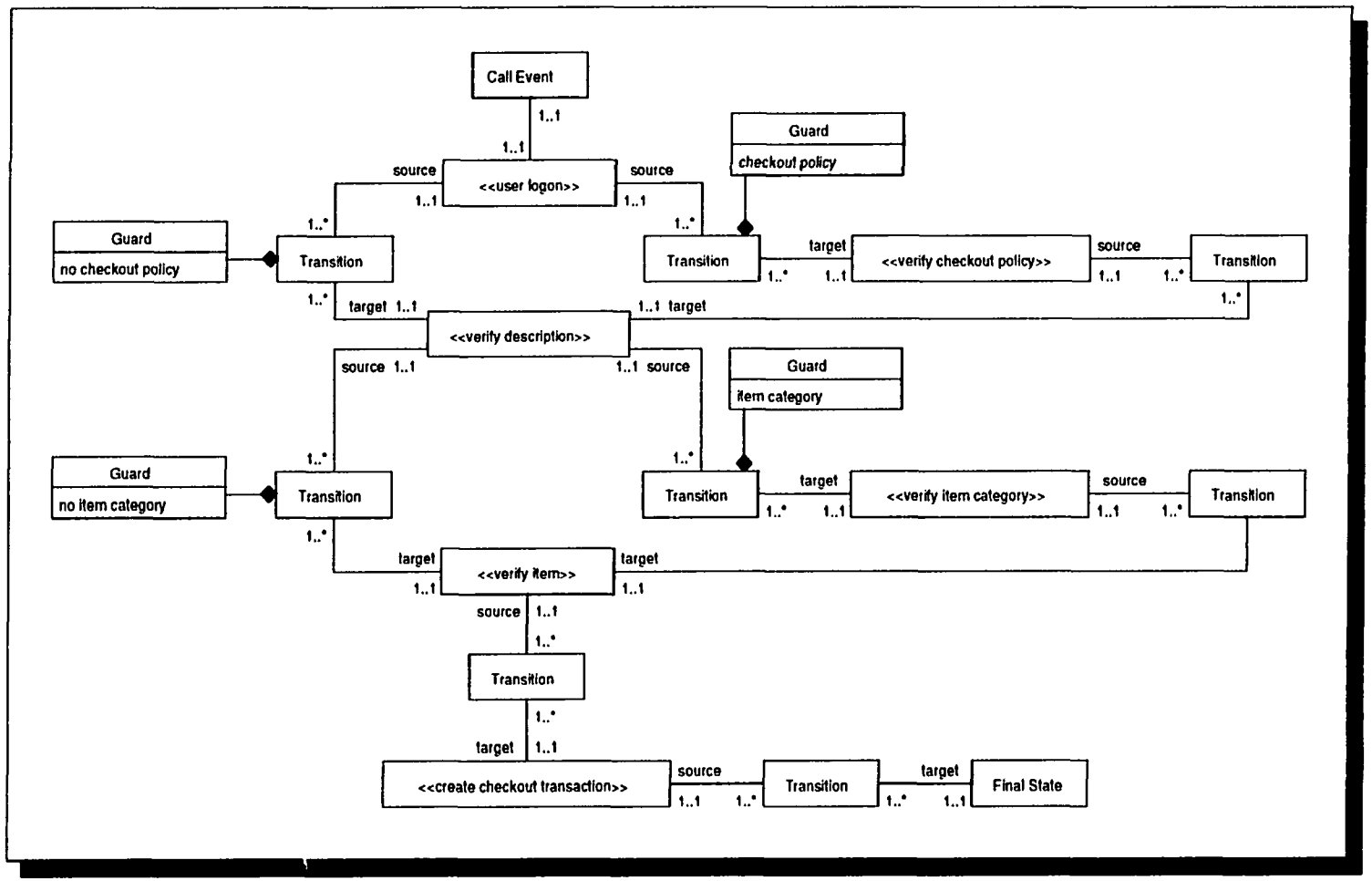

Figure 3.6.12.5: Check–Checkout Checkout Item activity diagram meta–model

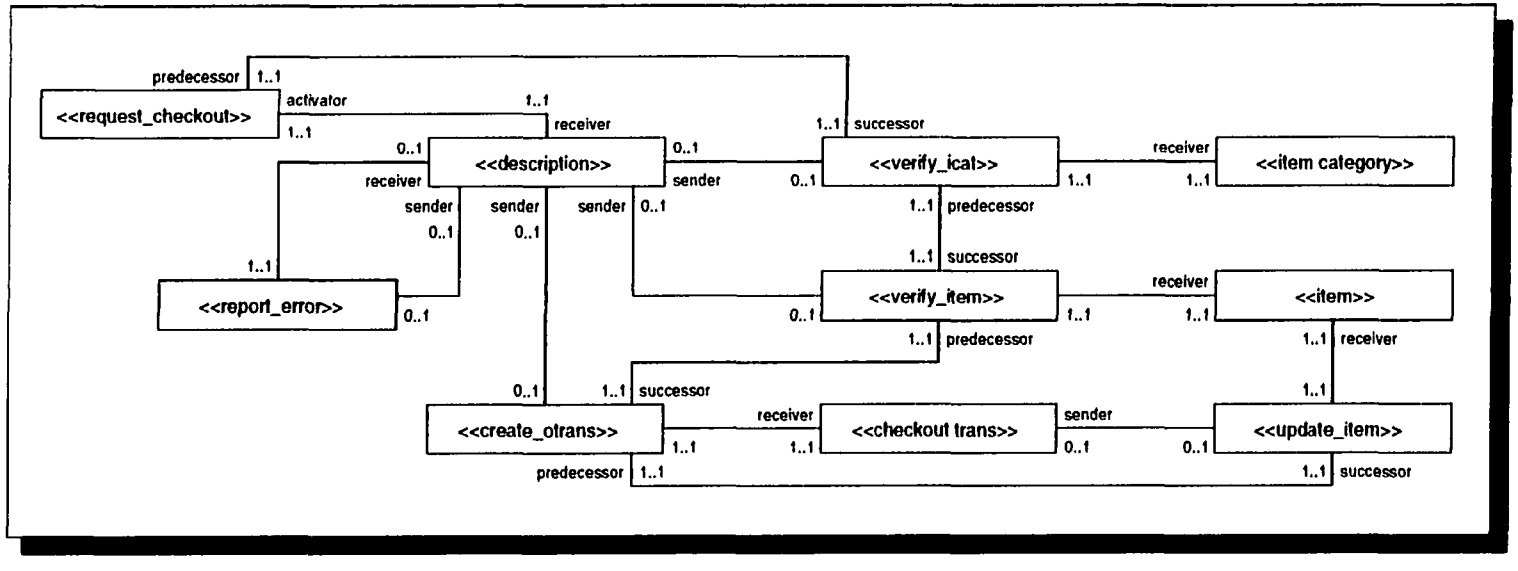

Figure 3.6.12.6: Checkin–Checkout Checkout Item collaboration diagram meta–model

Chapter 4

Engineering Domain Specific Modeling Language

4.1 Introduction

In order for the development and use of DSMLs to be successful there has to be a well defined process for creating DSMLs. In this chapter a process for creating DSMLs is presented, along with a description of how such DSMLs are intended to be used by application developers. Use of DSML by application developers is facilitated via a CASE tool environment, thus a description of how DSMLs may be incorporated into a CASE tool is also presented in this chapter. Figure 3.2 (DSSEE) identifies DSML engineering as an activity that uses the output of DA&D, namely the domain models and domain rules. This chapter will not address issues of how the domain models and rules are constructed - this was outlined in Section 3.2.1. It is assumed that these models and rules are available to the language development process.

The process for developing DSMLs will result in a profile packaging of the DSML components that were described in Chapter 3 and graphically illustrated in Figure 4.1, which has been reproduced for ease of reference.

Figure 4.1 shows that a DSML is a UML profile package that is composed of a set of UML profiles, which are name-spaces for the various meta-models, and semantic definitions of the language. This structuring of the DSML holds true to the principle, presented in Section 2.1, i.e. a language is composed of syntax (the meta-models) and semantics (the stereotype definitions).

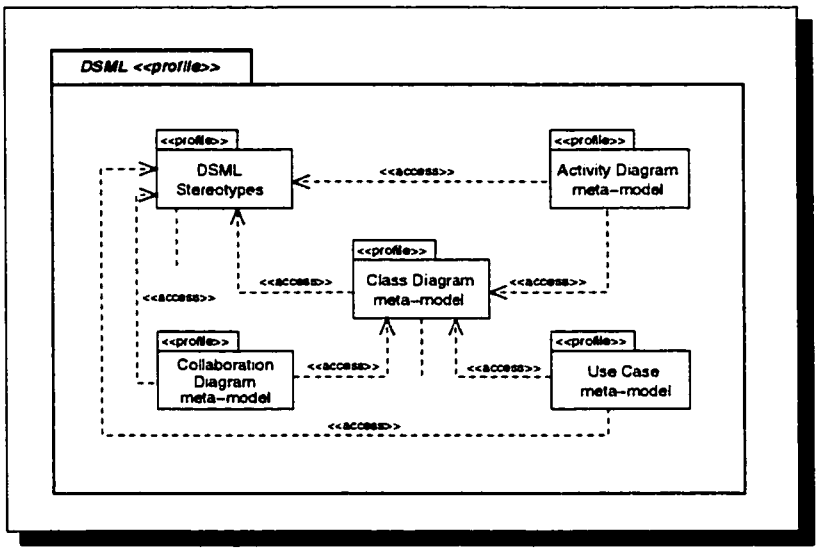

Figure 4.1: DSML architecture

4.2 The DSML Development Process

The process for developing a DSML in this work is made up of six key tasks. This process, of six tasks, is illustrated in Figure 4.2 in the format of a UML activity diagram. The six tasks associated with the development of DSML in the DSSEE are: (1) *Create SC* (static concept) *Stereotype*, (2) *Create DC* (dynamic concepts) *Stereotype*, (3) *Package Domain Stereotype*, (4) *Create Domain Meta-Models*, (5) *Package Domain Meta-Models*, and (6) *Package DSML* (domain-specific modeling language).

The first two task of DSML development (*Create SC Stereotype* and *Create DC Stereotype*) are actually started in the *Domain Analysis and Design* (DA&D) activity of DSSEE (see 3.2). These two tasks are presented in the *Language Engineering* activity of DSSEE because: (1) the importance of the stereotype definition activity to DSML development, and (2) only the initial aspects of the stereotype definition are carried out in the DA&D activity of DSSEE. The initial aspects of the stereotype definition involve: (1) the mapping of pertinent properties of the concepts to matching properties of UML model elements, and (2) specification of the semantics of the concepts in an informal textual format. In the description of the creation of the stereotypes all aspects of these tasks will be presented,

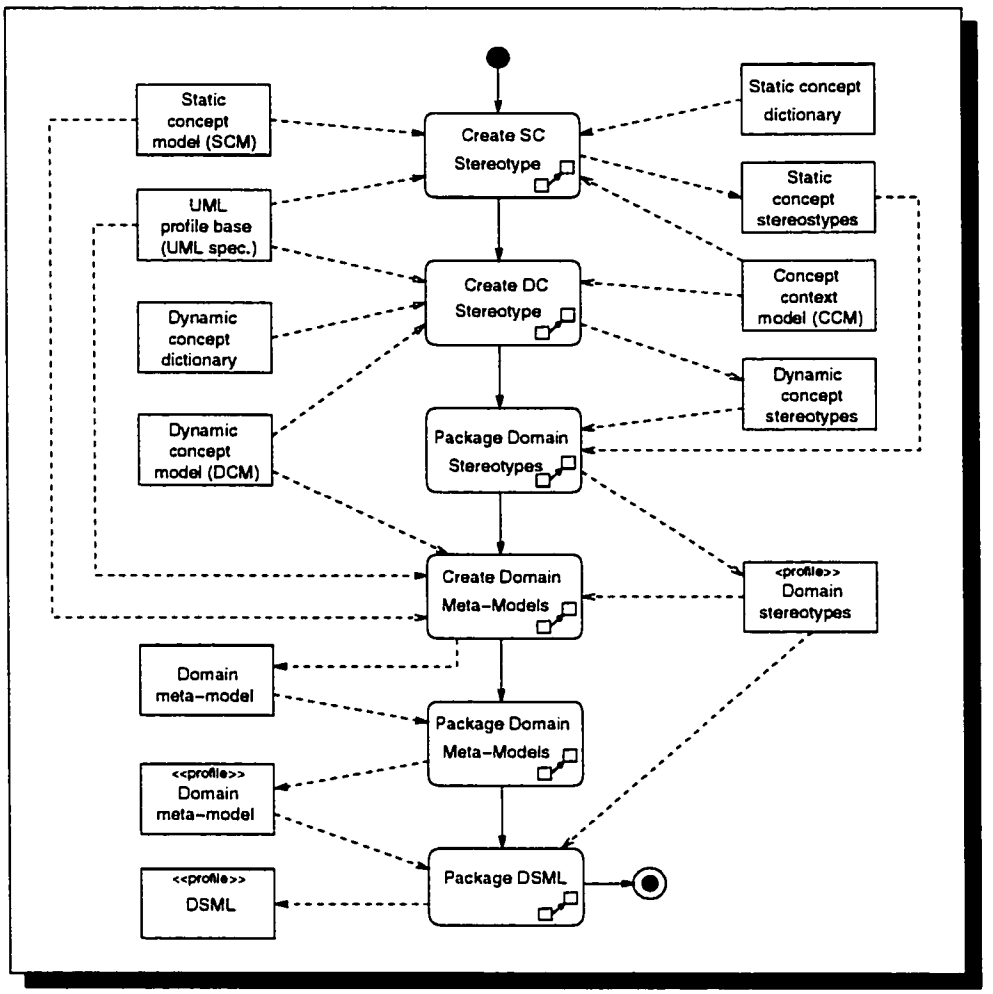

Figure 4.2: DSML development process

i.e. activities that occur during DA&D of Figure 3.2 with respect to stereotype creation are included in the definition of DSML. The following Sections 4.2.1 through 4.2.6 offer a detailed description of the DSML development tasks as illustrated in Figure 4.2.

4.2.1 Creating the SC stereotypes

Creation of the static concept stereotypes is carried out in an iterative manner with the static concept models (SCM, design class diagram and static concept domain dictionary) and the profile base (the UML specification) as inputs. The process for creating the static concept stereotypes is illustrated in the activity diagram of Figure 4.3.

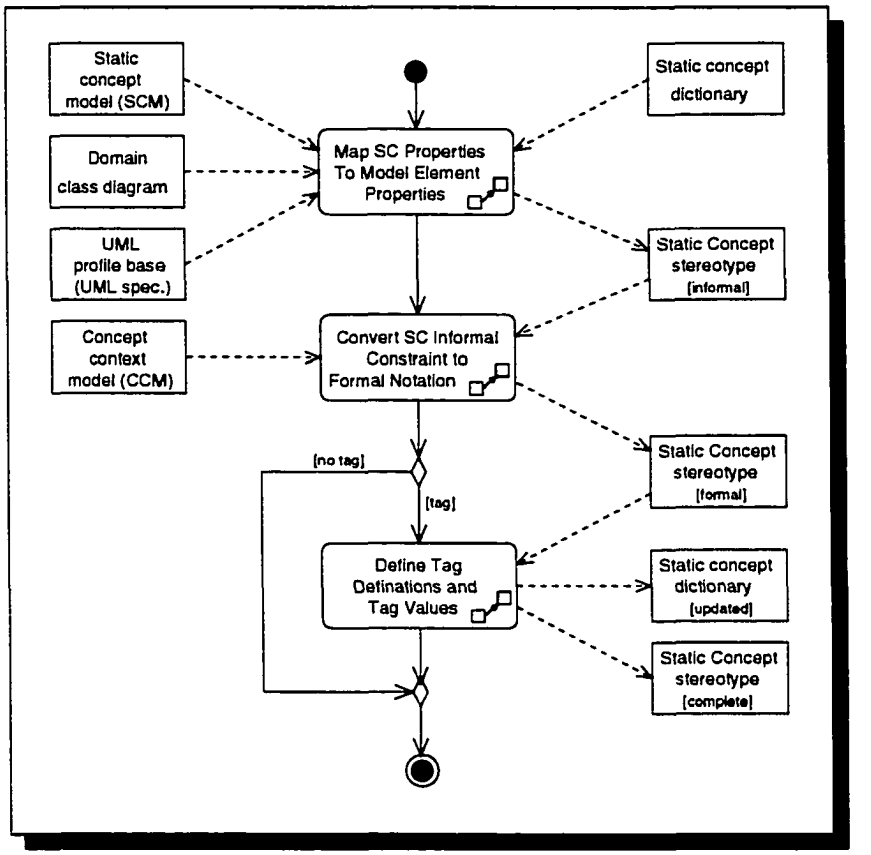

Figure 4.3: Create SC stereotype activity diagram

The detail description of the *Create SC Stereotype* activity diagram is as follows:

- *Map SC Properties to Model Element Properties* - For each static concept identified in the SCM and domain static concept dictionaries, their set of pertinent properties are matched to the set of properties of a UML model element that have similar properties. Thus each domain static concept in the problem (analysis) domain is associated with a UML model element in the solution (design) domain. Consequently, the problem concepts that have been *mapped* to UML model elements inherit the semantics of that UML model element. An example of this process from the $Checkin - Checkout$ domain is the decision to map the *Item* concept of the SCM to a UML *class* model element. This decision was driven by the properties of *Item* (uniqueness, attributes, concreteness, and relationships with other concepts) that correspond to the proper-

ties of the meta-model element *Class*. Inputs to this task are the SCM, domain class diagram, static concept dictionary, and the UML. The informal static stereotypes are created from this task. In addition to the prior mapping there may be additional mappings between the domain concepts and a formal domain for further precise semantic definition. This additional semantics that is derived from this second mapping must not conflict with the existing UML semantics - It may further constraint the UML semantics but it must not replace or invalidate any aspect of it.

The principle of *using a mapping from the properties of a concept in one domain to the properties of another concept in a different domain*, in order to *establish a semantic basis for the concept in one domain from that of a concept in another domain* is adapted from the work outlined in Section 2.1.2. This principle is applied throughout this work, and will be so noted when encountered. This *denotational* [84] *mapping* process is fundamental to the task of defining the semantics of a DSML, as it addresses an issue (*defining a precise semantics for the UML*) that has become a focus of research on the UML [44].

- *Convert SC Informal Constraint to Formal Notation* - The process to convert the informal constraints of the SC is carried out by first manually parsing the textual description of the concepts for phrases with a single subject and then **and**ing, **or**ing, or **xor**ing these phrases. The textual phrases are obtained from the requirements documentation and interviews with the users and domain experts. An example of this is the textual characterization of the semantics for the concept of electronic *memory* in a computer hardware configuration domain as:

  > A finite sequence of printable characters (*string*) **and** for which the cardinality of the sequence is fixed **and** whose characters may be updated, (where update is defined in terms of a set of OCL sequence operations.

  This may then be converted to OCL as:

  memory: Sequence(String) **and** *memory*→size() = *constant* **and**

```
context  memory  def:
         let self.update_memory(s: Integer, e:, n: String) =
           ((self→subSequence(1, e))→union(n))→union(
            self→subSequence(e + n→size(), self→size()))
```

This example also demonstrates the *mapping* principle that was described in the previous bulleted section on *Map SC Properties to Model Element Properties.* In this example the semantics of *memory* is obtained by a mapping of this concept to *Sequence(String)*, and restricting the known operations that may be applied to *Sequence(String)* to that of an *update* operation. Inputs to the task (*Convert SC Informal Constraint to Formal Notation*) are the CCM and the informal static stereotypes, with the output being the formal static concept stereotypes.

Because of the limited scope of the UML OCL (e.g. absence of an equivalence operator), which is the formal notation used in this work it may not be possible to explicitly convert all textual descriptions to OCL. There three options available as possible remedies for this situation:

1. Express the description as concise text.
2. Express the description with user-defined OCL constructs.
3. Express the description in some other formal notation.

The second option is the one selected in this work, an example of which was presented in the above discussion on the computer *memory* - where an operation to update an element of a sequence is defined in terms of two OCL sequence operations (subSequence() and union()).

- *Define Tag Definitions and Tag Values* - Tag definitions and tag values are defined in accordance with the UML specification, as outlined in Section 2.5.5.1. For the example domain the tag *mandatory* is defined for identifying which stereotyped model element must be used in an application model for the domain. This task is executed only if tag values and/or tag definitions are required. The inputs to this task are the formal stereotypes and the outputs are the completed (tag definitions and values added) stereotypes, along with the updating of the static concept dictionary to reflect the new meta-attributes.

Definition of the SC stereotype is difficult because the mapping of a SC to a UML model element is 1→**n**, where **n** may be > 2. The decision of which model element will represent a SC in the solution space is determined by the language engineer.

4.2.2 Creating the DC stereotypes

Figure 4.4 is an activity diagram for the *Create DC Stereotype* of Figure 4.2. This task is decomposed into three sub-tasks, namely: (1) *Define Use Case Stereotype*, (2) *Define Operation Stereotype*, and (3) *Convert Informal Operation Stereotype to Formal Notation.*

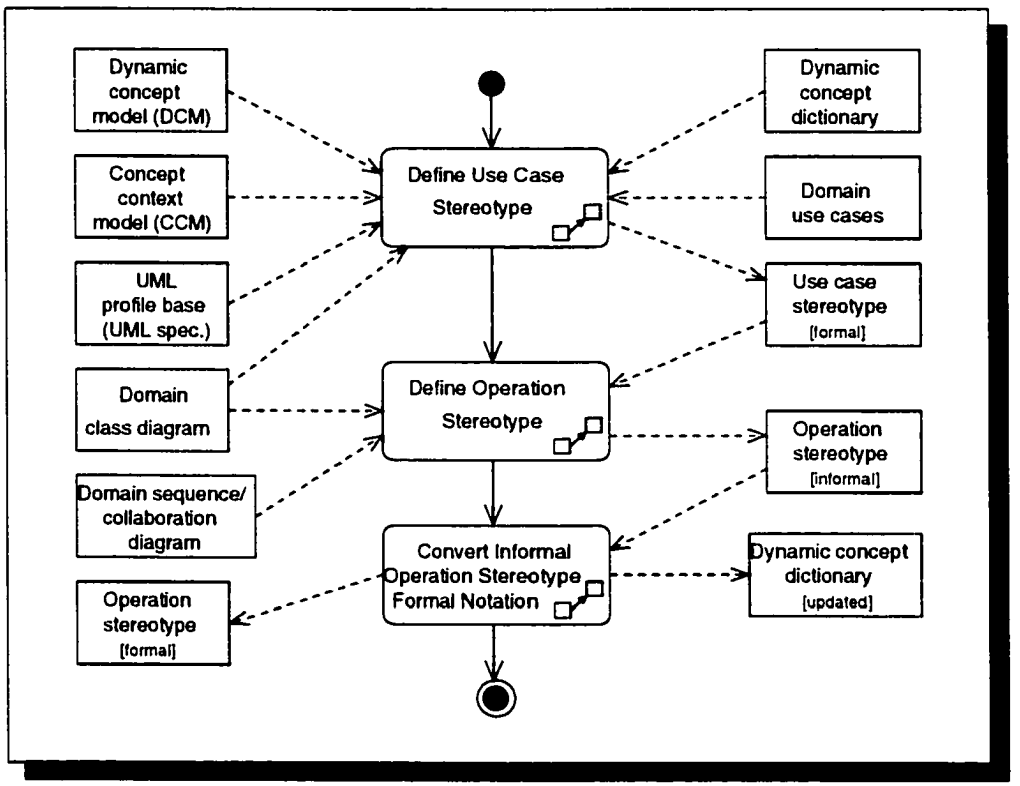

Figure 4.4: Create DC stereotype activity diagram

The detail descriptions of the *Create DC Stereotype* sub-tasks are:

- *Define Use Case Stereotypes* - Each use case identified in the domain dynamic concept dictionary, DCM, and the CCM is specified as a stereotyped use case for the purpose of reusability. This is a straight forward process of transforming the use case formal description to its corresponding use case stereotype *Constraint*, and copying

the use case informal description to its corresponding use case stereotype *Description*. The inputs to this task are the DCM, CCM, UML specification, dynamic concept dictionary, and use cases. The output is the formal use case stereotypes.

- *Define Operation Stereotype* - Definition of the operation stereotypes is carried out by first identifying the class that owns the operation. This information is obtained from the domain collaboration or sequence diagram. The description of the operation is first specified in concise informal text. The format of this description is similar to that of the informal DC (use case) constraints of Section 3.2.1.3. Inputs to this task are the domain class diagram, collaboration or sequence diagram, and informal use cases, with the output being the *informal Operation stereotype*.

- *Convert Informal Operation Stereotype to Formal Notation* - The informal operation stereotype descriptions are converted to UML OCL, and the informal specification is now presented as the description of the stereotype. The input to this task is the informal stereotype and the output is the formal stereotypes. The dynamic concept dictionary is also updated.

The mapping of a DC to a UML model element is 1→1..2 (i.e. DC is mapped to either a use case or an operation).

4.2.3 Packaging the stereotypes

The packaging of the stereotypes involves the definition of a name-spaces for the three sets of stereotypes (static concepts, use cases, and operations). The UML name-space concept ≪*profile*≫ is used to package the domain stereotypes, as illustrated in Figure 4.5.

The sets of stereotypes are individually packaged in profiles and these profiles are in turn packaged as a profile of profiles (*Domain Stereotypes*). In the profile of Figure 4.5 the relationship between the *Use Case Stereotype* and the *Static Concept Stereotype* profiles is that the *Use Case Stereotype* profile ≪*access* ≫ the *Static Concept Stereotype* profile. The *Operation Stereotype* profile has mutual ≪access≫ relationships with the *Static Concept Stereotype* and *Use Case Stereotype* profiles. The stereotype ≪*access*≫ indicates that model

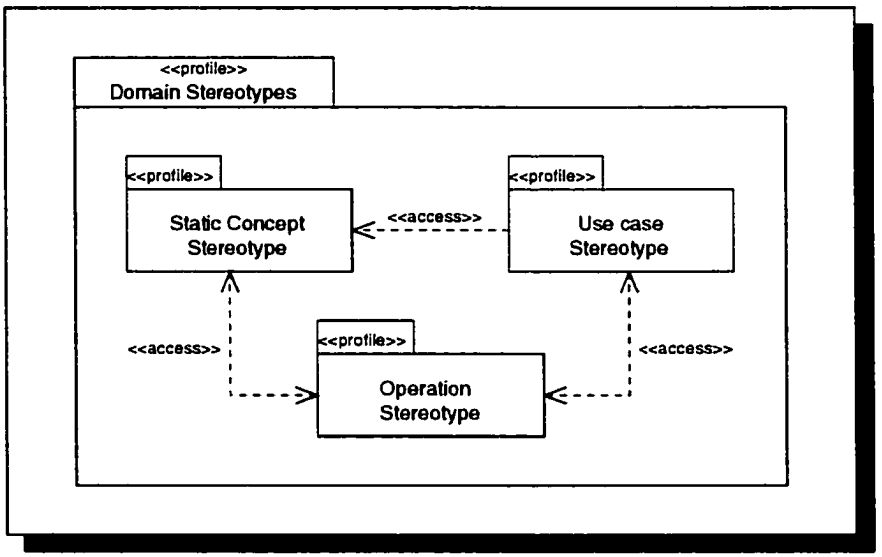

Figure 4.5: Domain stereotypes packaged as UML profile

elements from the accessed profile package may appear, unchanged, in the accessing profile package.

4.2.4 Creating the meta-models

Figure 4.6 is an activity diagram for the *Create Domain Meta-Models* of Figure 4.2. This task is decomposed into two sub-tasks, namely: (1) *Define Graphical Icon for Stereotype*, and (2) *Establish Association Between Icons*.

The detail descriptions of the *Create Domain Meta-Models* sub-tasks are:

- *Define Graphical Icons* - For each stereotype listed in the *Domain Stereotypes* profile package an iconic representation is defined. The icon may be the default UML representation or a domain-specific designed icon. These icons are the ones that will be used in developing domain application models. The inputs to this task are the *Domain stereotypes* and the UML specification, and the output is the icons.

- *Establish Associations Between the Icons* - Associations are established between the stereotyped icons of the domain to complete the meta-models. Like the graphical icon definitions, some of these associations are domain specific (i.e. specialized UML

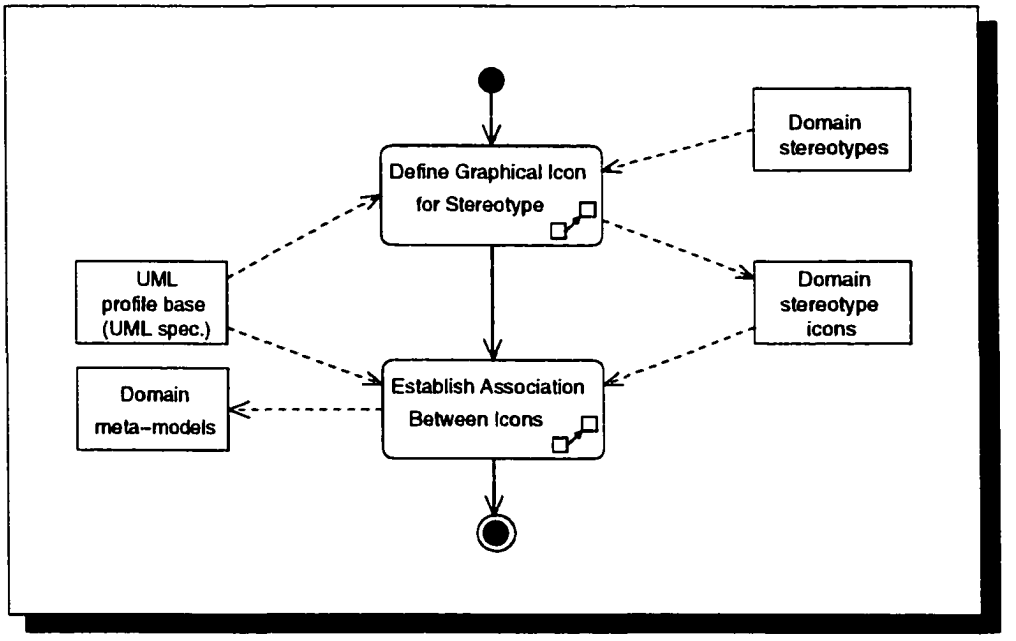

Figure 4.6: Create domain meta-model activity diagram

associations) or generic UML associations. The multiplicities on these associations are contained in the respective stereotype definitions. The input to this sub-task are the *Domain stereotype icons* and the UML specification, with the output being the *Domain meta-models*.

The domain meta-models are intended to provide a graphical representation of a sub-set of the information contained in the domain stereotypes.

4.2.5 Packaging the meta-models

The packaging of the meta-models involves the definition of a name-spaces for the meta-models of all the models defined for the domain. The UML name-space concept *≪profile≫* is used to package the domain meta-models, as illustrated in Figure 4.7.

The meta-models are individually packaged in profiles and these profiles are in turn packaged as a profile of meta-models (*Domain Meta-Models*). In the profile of Figure 4.7 the relationship between the *Class Diagram meta-model* and the other profile packages is that the other profile packages *≪access≫* the *Class Diagram meta-model*. The accessing

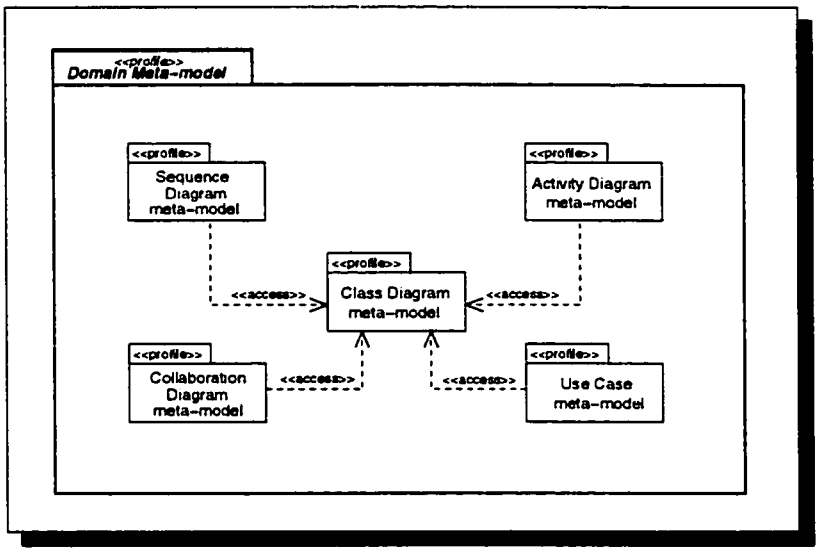

Figure 4.7: Domain meta-model packaged as UML **profile**

of the *Class Diagram meta-model* profile package by the other profile packages indicates that the domain class diagram model elements may appear, unchanged, in the other profile packages.

4.2.6 Packaging the DSML

The packaging of the DSML involves the definition of a name-spaces for all the domain profiles that have been previously defined, i.e. *Domain Stereotypes*, and *Domain Meta-Models*. The UML name-space concept *≪profile≫* is used to package these profiles, as illustrated in Figure 4.8. This is the same architecture as was identified in Section 3.1, and illustrated in Figure 3.1.

It should be noted that the packaging of the DSML is obtained by extending the meta-model profile package to include the DSML stereotype profile package. The unattached line from the *Class Diagram meta-model* profile indicates that there may be additional meta-model profile packages. This concludes the definition of the process for the engineering of a domain-specific modeling language (DSML).

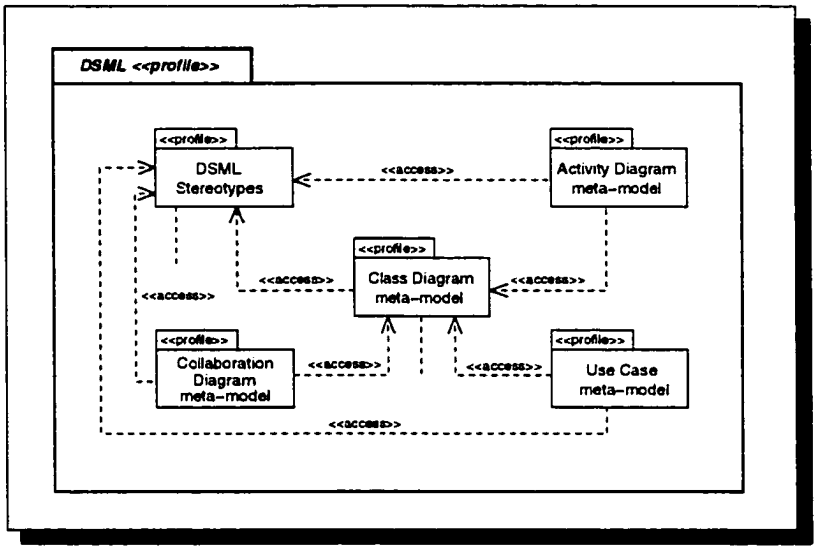

Figure 4.8: DSML packaged as UML profile

4.3 Using a DSML

DSML, as presented in this work, are intended to be used by application developers/engineers in a CASE tool environment. Tool developers would encode the DSML in the CASE tool in a manner that is identical to that which is currently done with the UML, with one major exception. This exception is with respect to the interpretation of the semantics of the DSML. The syntax of the DSML is presented to the tool developer in an identical format to that of the UML, i.e. a set of meta-models (abstract syntax).

The semantics of the UML as offered in the UML specification and used by tool developers is textual. This textual format is then interpreted by the particular developers and incorporated into their CASE tool. This results in semantics for the UML that may vary from one CASE to another. Whereas, the semantics of a DSML is offered to a tool developer in the format of a set of stereotypes that are constrained by OCL expressions and denotational mappings that constraint the model elements to the specific requirements of the domain. This is more precise and is directed towards a single interpretation for each DSML semantics. Consequently, the encoding of the semantics of a DSML in all CASE tools will result in all such tools expressing the semantics of the DSML in an identical manner.

An example of a CASE tool environment that an application developer/engineer will work in is illustrated in Figure 4.9.

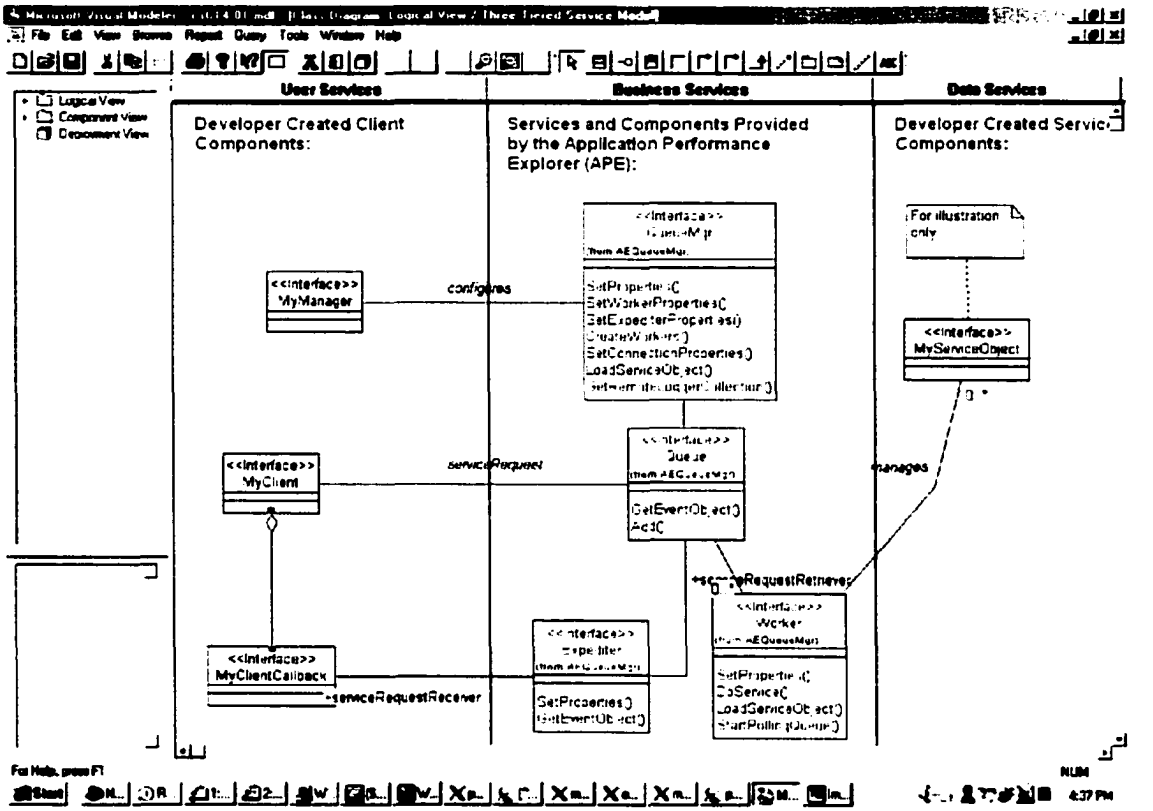

Figure 4.9: CASE tool for DSML

The stereotyped and UML generic icons for the domain would be listed on the drawing palette of a DSML tool . The mandatory concepts may be presented in a different color and/or text font from that of the optional ones. The application developer would select the desired concept icons from the palette and place them in the workspace area. The tool would then automatically include the mandatory concepts associated with the selected icon(s). This process would continue until the developer has created models that satisfy the requirements for the application.

Figure 4.10 illustrates an activity diagram for using a DSML in a tool environment. It is assumed that the developer has documented the set of user services required for the application prior to commencing work with the DSML.

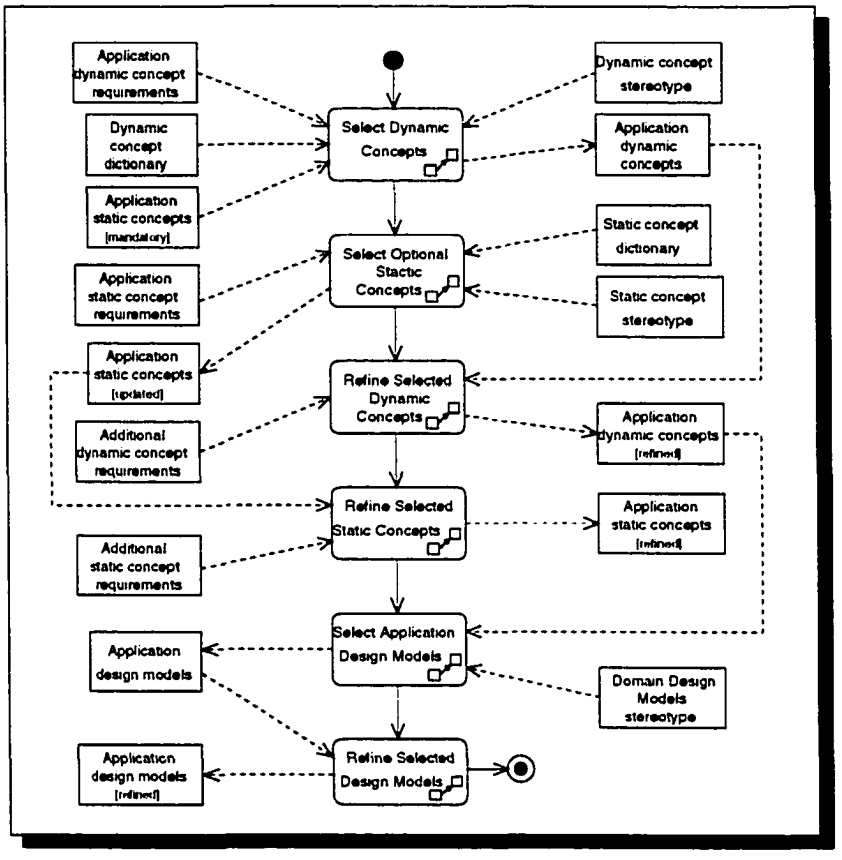

Figure 4.10: Process for using a DSML

The detail description of the process of Figure 4.10 is as follows:

- *Select Dynamic Concepts* - By examining the domain *Dynamic concept dictionary* and *Dynamic concept stereotype* (which are accessible from within the CASE tool) the application developer is able to identify and then select the set of dynamic concepts that satisfy the dynamic requirements (user services) as set out in the *Application dynamic concept requirements.* In so doing a mandatory set of static concepts will be automatically determined by the CASE tool. The output of this task is the set of *Application dynamic concepts* and *mandatory Application dynamic concepts.*

- *Select Optional Static Concepts* - The developer will next select the set of optional static concepts that are needed to satisfy the *Application static concept requirements.* This is accomplished by examining the *Static concept dictionary* and *Static concept*

stereotype (which are accessible from within the CASE tool) in order to identify those that have matching properties of the concepts listed in the *Application static concept requirements*. The output of this task is the *updated* set of *Application static concepts*.

- *Refine Selected Dynamic Concepts* - The dynamic concepts selected (*Application dynamic concepts*) by the application developer are not complete with respect to the specific requirements of the application. Hence, the developer will have to *refine* the dynamic concepts specification in order to completely satisfy the application requirements. This task is accomplished by adding constraints for the specifications of the application that are not contained in the unrefined dynamic concept specification. These added constraints must not violate any of the already existing constraints. The output of this task is the *refined Application dynamic concepts*.

- *Refine Selected Static Concept* - Similar to the task of refining the dynamic concepts, the developer will be required to refine the *Application static concepts*. This refinement may involve: (1) including additional attributes and operations in the classes, and (2) including additional design classes and associations. The output of this task is the *refined Application static concepts*.

- *Select Application Design Models* - The developer will next have to *Select Application Design Models* to satisfy the application requirements. Selection of the *Application dynamic concepts*, i.e. application use cases, will automatically identify a complete set of design models (activity, collaboration, sequence diagrams) associated with the use cases. The developer will then select those that will be used to express the design solution for th application. The output of this task is a set of design models.

- *Refine Selected Design Models* - The selected design models, like that of the initially selected dynamic and static concepts have to be refined to meet the specific requirements of the application. The initially selected concepts (dynamic and static) and design models are *domain−specific* and the refinement process transform them into *application–specific* concepts and models. The task of refining the *Application design models* involves: (1) selection of alternate paths in activity diagrams, (2) population

of the design models with the refined design classes and associations, (3) determining the application-specific operations for the design models and (4) including additional design level operations.

By using a DSML the application developer/engineer is able to leverage the analysis and design decisions that have been incorporated in the language. There is also the benefit of using analysis and design models that have an implicit level of high reliability that was derived from the work conducted during the DA&D and language engineering activities. This will result in a greater level of efficiency in the analysis and design phase of application development, and the production of models that are: (1) completed in a shorter time-frame than during the pre-DSML development cycle, and (3) applications that are more easily maintained.

Chapter 5

Case Study (Service Provisioning System)

5.1 Introduction

In this chapter a DSML for a sub-domain within the telecommunication industry will be developed, using actual data from a one such corporation. The data presented herein will be sanitized so as to protect the identity and proprietary rights of the company. The sanitization process will involved: (1) the renaming of some concepts presented, so as not to reveal the identity of the company, (2) and abstracting out some details of concepts presented, so as to protect any proprietary information.

The process outlined in Chapter 4 will be applied to the data obtained to produce a set of models as presented in Chapter 3. It will not be possible to develop some of the models listed in Chapter 3 for this domain, as the data obtained is incomplete. Aspects of this incompleteness will be handled by recreating some of the missing data through an intuitive and logical analysis of the available data.

The data that will be used in this chapter was obtained during a graduate internship with a telecommunication company. Throughout the remainder of this chapter the company will be referred to as **Alpha, Inc.** This company was formed from the merger of two existing telecommunication companies (**Alpha, Inc.** and **Beta, Inc**), which resulted in the need to merge the existing two information technology (IT) departments into a single operational unit. The company decided to go about this operational merger of the two IT departments by analyzing the duel systems for making a determination as to: (1) replace **Alpha's** system

with **Beta's** system, (2) replace **Beta's** system with **Alpha's** system, (3) use parts of both systems to build a new one, or (4) replace both systems with a new one.

The internship work involved the creation of a set of UML models for the two systems (one from **Alpha** and the from **Beta**) that essentially provided the same set of services. These models would then be used my **Alpha's** IT management to aid in their decision making process, as to which of the above four options will be selected for these two systems. Thus the internship team was face with conducting a *domain analysis* of the service domain, with the documentation of the two systems as the legacy input to the DA&D process, along with the input from domain experts (the engineers, users, and managers) involved with the two systems.

The purpose of the DA&D exercise conducted departed from the traditional purpose of DA&D in DSSEE, as illustrated in Figure 1.2, and presented in Section 2.4.4. The traditional purpose of DA&D activity in general and in DSSEE is for "*... identifying, collecting, organization, analyzing, and representing ... knowledge ... in support of the description and solutions* [of] *problems*". Whereas in the **Alpha** project the purpose was not for the derivation of a solution (solutions exist in the form of the two systems analyzed) but to determine whether either of the existing solution(s) is (are) suitable. At the end of the internship exercise the DA&D outputs would prove to be an added benefit to **Alpha**, as they concluded that a new system would be developed to replace the existing two systems, and the models of the DA&D activity would be used in defining the new system.

5.2 Description of the Domain

The domain of **Alpha, Inc** business that was investigated is that of the provisioning of electronic voice and data communication services to a major portion of its consumer base. These services include: (1) digital subscriber line (DSL), (2) wireless service, (3) repair verification, (4) call forwarding, (5) call blocking, (6) integrated service digital network (ISDN), (7) local number portability, (8) plain old telephone service (POTS), (9) ring cycle control, (10) call waiting, (11) toll free call, (12) calling card, (13) credit card billing, and variants of these services. There are two systems, owned by **Alpha** that automatically

provision these services to the customers. One of these two systems was inherited by **Alpha, Inc.** when it acquired **Beta, Inc.**, and the other was its own service provision system (SPS). Some of the provisioning services are duplicated in both systems and some service provisioning are offered by only one of the systems. The number of duplicated services is greater that the number of un-duplicated services.

The original **Alpha** SPS is a real-time web-based (Internet) and client server graphical user interface system. In this system the customers will request, from the list of available telecommunication services, a set of desired phone services over the Internet by supplying the required data. The SPS inherited by **Alpha** from **Beta** is designed to receive and process transaction request from multiple interfaces by **Beta** customer service representatives. In the original **Beta** SPS all required data had to be submitted before the automatic processing would proceed. The **Beta** SPS would attempt to automatically provision the service even in the situation where the required data is incomplete.

Both systems interfaces with other sub-systems to establish or update: (1) client account data, (2) service requisition orders, and (3) telephone network switch instructions. Both SPSs run on multiple types of hardware and software platforms, and interface with multiple types of telephone network switches. Both systems incorporate fault tolerance via hardware and software redundancy. In the original **Alpha** SPS fault tolerance is enhanced by requiring that complete service requisition data be submitted before any attempt is made to provision the service. While, the original **Beta** system hands off failed service request to a sub-system where a **Beta** representative interfaces to make amendment to the fail service request. In both system the initial requisition data is converted to parameterized code that is then loaded onto hardware telecommunication network element processors (NEP). The code then execute on these processors to set the network switches that completes the service request.

The scope of the system analyzed commences with the input of the service requisition data and terminates at the point at which the data is converted to NEP code and sent to the NEP. The scope is also bounded at points where processing is handed off to sub-systems that handle other aspect of the service request, e.g. customer accounting, and billing. The

hand off to error handling sub-systems is included in the scope of the SPS that is analyzed.

5.3 Specifying the Domain Models

The activities for defining the domain models in DSSEE as presented in Section 2.4.4 is assumed to have been conducted within a preferred domain analysis and design methodology, and these models are presented below in Sections 5.3.1 through 5.3.6. It is to be noted that the definition of a DSML is independent of any domain analysis methodology that is used to define the domain models for DSML. In Section 2.4.4 a set of activities from FODA [56] and CA [96] were identified for creating the analysis models (SCM, DCM, and CCM). The design models (design class, activity, collaboration, and sequence diagrams) are derived from the analysis models.

5.3.1 The static conceptual model

The static concept model (SCM) for **Alpha's** service activation system is presented in Figure 5.1.

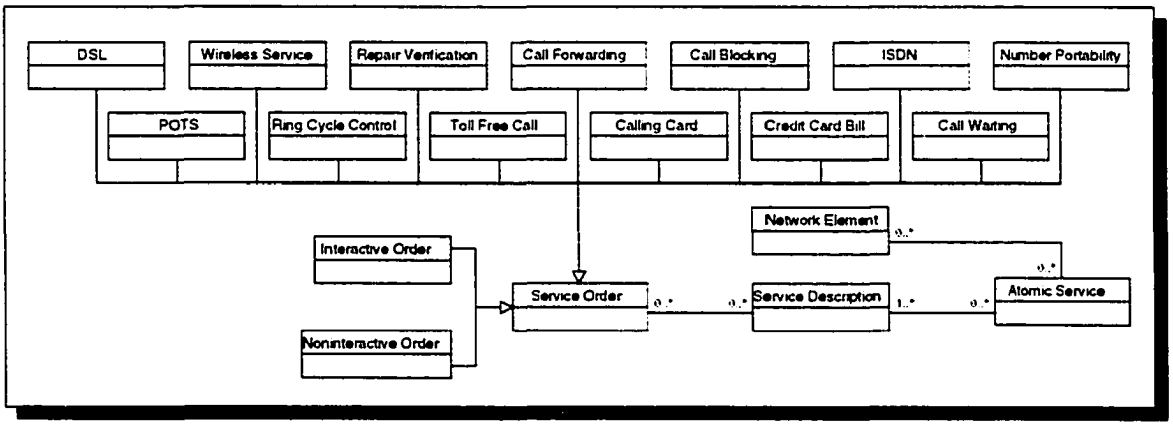

Figure 5.1: SCM for **Alpha, Inc.** service activation system

Figure 5.1 illustrates the following static requirements of the domain:

- There is a variety of *Service Orders*, which may be provisioned by the system, and these *Service Orders* are processed in an interactive (*Interactive Order*) or non-interactive (*Non-Interactive Order*) manner.
- A *Service Order* may be associated with multiple *Service Description*, and a *Service Description* may be associated with multiple *Service Orders*.

- A *Service Description* may be associated with multiple *Atomic Services*, and an *Atomic Service* may be associated with one or more *Service Descriptions*.
- An *Atomic Service* may be associated with multiple *Network elements*, and a *Network Element* may be associated with multiple *Atomic Services*.

The static concept dictionary (SCD) is developed concurrently with the SCM. The initial SCD for **Alpha**'s service provisioning system is presented in Table 5.1.

Table 5.1: **Alpha, Inc.** domain dictionary (static concepts)

Term	*Property*	*Example*
service order	abstract, mandatory *association* service description	wireless service
DSL	concrete, optional *specialization* service order	data transfer
wireless service	concrete, optional *specialization* service order	pager
repair verification	concrete, optional *specialization* service order	check service is on
call forwarding	concrete, optional *specialization* service order	forward POTS to pager
call blocking	concrete, optional *specialization* service order	block 1 900 numbers
ISDN	concrete, optional *specialization* service order	Internet and POTS
number portability	concrete, optional *specialization* service order	change address
POTS	concrete, optional *specialization* service order	POTS
ring cycle control	concrete, optional *specialization* service order	forward on 3^{rd} ring
toll free call	concrete, optional *specialization* service order	1 800 number
calling card	concrete, optional *specialization* service order	phone card
credit card bill	concrete, optional *specialization* service order	bill to VISA card
call waiting	concrete, optional *specialization* service order	interrupt call
interactive order	concrete, optional *specialization* service order	failed order
non-interactive order	concrete, optional *specialization* service order	non-failed order
network element	concrete, mandatory *association* atomic service	network switch
service description	concrete, mandatory *association* service order, atomic service	load next command
atomic service	concrete, mandatory *association* service description, network element	set switch on

Attributes have not been included in the SCM of Figure 5.1 because of the specificity of the known attribute names to **Alpha**'s SPS, and including such attribute names do not enhance the problem definition activity at this stage of the language development process. At the point when the design CD is introduced then some known generic attribute names will be included in the model - The intent is to keep the models as simple as possible, by adding information when it is needed and not before.

5.3.2 The dynamic conceptual model

The dynamic concept model (DCM) for **Alpha's** service activation system is presented in Figure 5.2.

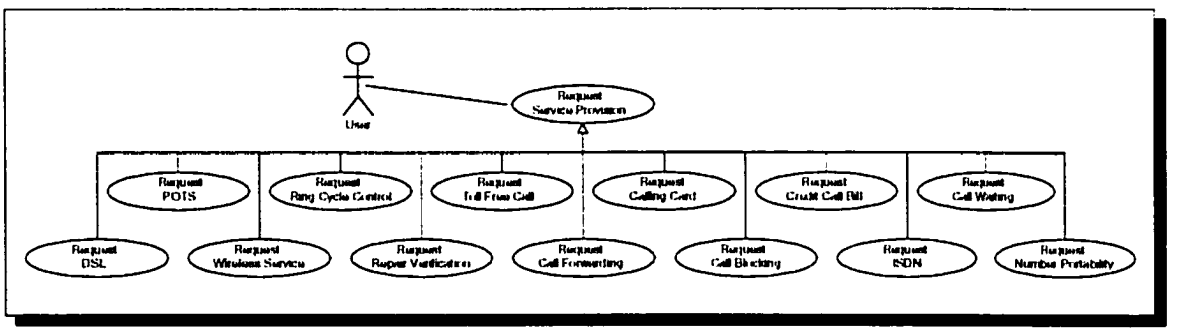

Figure 5.2: DCM for **Alpha, Inc.** service activation system

Figure 5.2 illustrates that the user service requirements of the domain are in a one-to-one relationship with the static service concepts, as presented in Figure 5.1. The dynamic concept dictionary (DCD) is developed concurrently with the DCM. The initial DCD for **Alpha**'s service provisioning system is presented in Table 5.2.

The DCD and SCD are updated at the end of the DA&D activity to include all information that has been elicited during the DA&D process about the concepts.

For each use case listed in the use case diagram of Figure 5.2 a use case specification is created. Tables 5.3 and 5.4 are specifications for the use cases of Figure 5.2 (*Request Service Provision* and *Request POTS*). Complete specification for all the use cases of the DCM

Table 5.2: **Alpha**'s domain dictionary (dynamic concepts)

Term	*Property*	*Example*
request service provision	abstract, optional	
request POTS	concrete, optional	request POTS
request ring cycle control	concrete, optional *specification* Customer must exist and Customer is associated with ring cycle type service.	forward on 3^{rd} ring
request toll free call	concrete, optional *specification* Customer must exist and Customer is associated with toll free type service.	request 1 800 number
request calling card	concrete, optional *specification* Customer must exist and Customer is associated with calling card type service.	request phone card
request credit card bill	concrete, optional *specification* Customer must exist and Customer is associated with credit card.	request bill VISA card
request call waiting	concrete, optional *specification* Customer must exist and Customer is associated with call waiting type service.	request call interrupt
request DSL	concrete, optional *specification* Customer must exist and Customer is associated with DSL type service.	data transfer
request wireless service	concrete, optional	data transfer
request repair verification	concrete, optional *specification* Customer must exist and repair service was previously requested.	check service is on
request call forwarding	concrete, optional *specification* Customer must exist and Customer is associated with call forwarding type service.	request POTS forward to pager
request call blocking	concrete, optional *specification* Customer must exist and Customer is associated with call blocking type service.	request block 1 900 numbers
request ISDN	concrete, optional *specification* Customer must exist and Customer is associated with ISDN type service.	request Internet and POTS
request number portability	concrete, optional *specification* Customer must exist and Customer is associated with number portability type service.	request address change

(Figure 5.2 *Request Service Provision. Request POTS. Ring Cycle Control. Request Toll Free Call. Request Calling Card. Request Credit Call Bill. Request Call Waiting. Request DSL. Request Wireless Service. Request Repair Verification. Request Call Forwarding. Request Call Blocking. Request ISDN*, and *Request Number Portability*) are given in Section 5.5 at the end of this chapter.

Table 5.3: Use Case: *Request Service Provision*

Use Case	Request Service Provision
Generalization	
Specialization	Request Service Provision, Request POTS, Request Ring Cycle Control, Request Toll Free Call, Request Calling Card, Request Credit Call Bill, Request Call Waiting, Request DSL, Request Wireless Service, Request Repair Verification, Request Call Forwarding, Request Call Blocking, Request ISDN, Request Number Portability
Read	Customer, Service Order, Interactive Order, Non-interactive Order
Update	Network Element
Create	Atomic Service, Service Description
Delete	
Input	serv: Service Order, cust: Customer
Output	stab: State Table
Formal description	**context** Request_Service_Provision **inv:** (Network_Element.oclIsKindOf(Class) **and** Interactive_Order.oclIsKindOf(Class) **and** Non-interactive_Order.oclIsKindOf(Class) **and** Atomic_Service.oclIsKindOf(Class) **and** Service_Description.oclIsKindOf(Class) **and** Service_Order.oclIsKindOf(Class) **and** Customer.oclIsKindOf(Class) **and** Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)))
Informal description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order.

The following use case specifications (*Request POTS* is a specialization of the *Request Service Provision* of Table 5.3.

Table 5.4: Use Case: *Request POTS*

Use Case	Request POTS
Generalization	Request Service Provision
Specialization	
Read	Customer, Service Order, Interactive Order, Non-interactive Order
Update	Network Element
Create	Atomic Service, Service Description
Delete	
Input	pno: POTS, cust: Customer
Output	stab: State Table
Formal description	**context** Request_POTS **inv:** (Network_Element.oclIsKindOf(Class) **and** Interactive_Order.oclIsKindOf(Class) **and** Non-interactive_Order.oclIsKindOf(Class) **and** Atomic_Service.oclIsKindOf(Class) **and** Service_Description.oclIsKindOf(Class) **and** Service_Order.oclIsKindOf(Class) **and** Customer.oclIsKindOf(Class) **and** Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)) **and** POTS→forAll(ps | ps.pots_no@pre <> pno.pots_no))
Informal description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order **and** the number to be provisioned (pno.pots_no) must not exist at the start.

It will be noted that the use case specifications of Table 5.11 through 5.23 contain information, such as class attributes, which would not have been available at the point when these use case specifications were initially created. The contents of the specification, as presented in Tables 5.16 to 5.23 was obtained by updating the initial use case specifications after the CD of Figure 5.4 was completed. Hence, Tables 5.16 to 5.23 represents the final iteration of the use case specification process.

5.3.3 The concept context model

The concept context model (CCM) for **Alpha's** service activation system is presented in Figure 5.3.

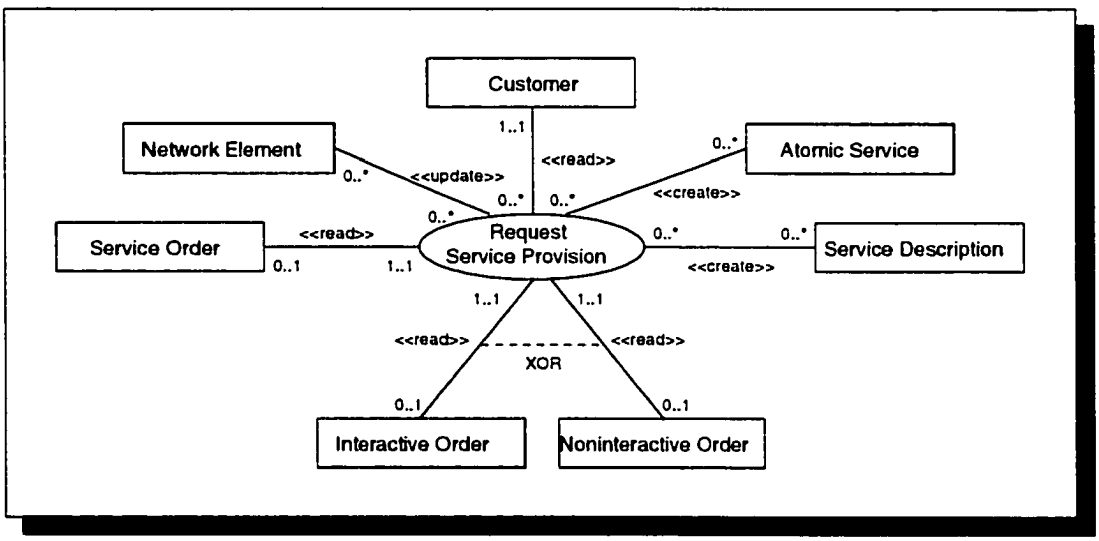

Figure 5.3: CCM for **Alpha, Inc.** service activation system

Figure 5.3 illustrates that all service requests of the domain interacts with the same set of static concepts. Itemized below are the relationships between the static concepts and the dynamic concepts:

- A *Request Service Provision* may be associated with multiple *Atomic Services*, and an *Atomic Service* may be associated with multiple *Request Service Provisions*.
- A *Request Service Provision* may be associated with multiple *Description Services*, and a *Description Service* may be associated with multiple *Request Service Provisions*.
- A *Request Service Provision* may be associated with multiple *Non-interactive Orders*, and a *Non-interactive Service* is associated with one *Request Service Provision*.

- A *Request Service Provision* may be associated with multiple *Interactive Orders*, and an *Interactive Service* is associated with one *Request Service Provision.*
- A *Request Service Provision* may be associated with multiple *Service Orders*, and a *Service Order* is associated with one *Request Service Provision.*
- A *Request Service Provision* may be associated with one *Network Element*, and a *Network Element* may be associated with multiple *Request Service Provisions.*
- A *Request Service Provision* is associated with one *Customer*, and a *Customer* may be associated with multiple *Request Service Provisions.*

The domain experts introduced a set of dynamic concepts (internal system services) that are fundamental to the operation of the system. Though these dynamic concepts are not visible to the system users they are introduced here because of their importance. These dynamic concepts will be elaborated on in the domain design models. Itemized below are the relationships between the static concepts and these internal dynamic concepts:

- A *Service Order* may be associated with a single service request processor (*SRP*), and a *SRP* may be associated with multiple *Service Orders.*
- A *Service Order* may be associated with a single activation manager (*AM*), and an *AM* may be associated with multiple *Service Orders.*
- A *SRP* may be associated with multiple *Service Descriptions*, and a *Service Description* is associated with one or more *SRP.*
- A *Service Description* may be associated with multiple *AMs*, and an *AM* may be associated with multiple *Service Descriptions.*
- A network element processor *NEP* may be associated with multiple *AMs*, and an *AM* may be associated with multiple *NEPs.*
- A *NEP* may be associated with multiple *Network Elements*, and a *Network Element* may be associated with multiple *NEPs.*

The *Network Element* is outside of the stated scope of the domain, but have been mentioned here for completeness of the description of the concepts.

5.3.4 The design class diagram model

The design level CD model for **Alpha's** service activation system is presented in Figure 5.4.

From the design CD the SCD is updated to produce the following final version as presented in Table 5.3:

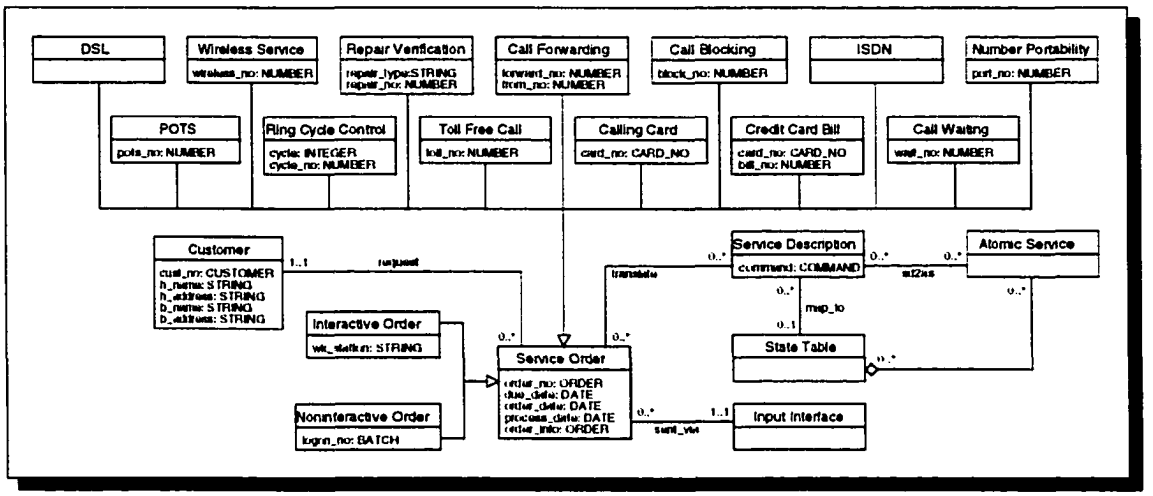

Figure 5.4: Design CD for **Alpha, Inc.** service activation system

The existing application models were not developed as object-oriented (OO) models, thus adding difficulty to the process of deriving OO domain models from these application models. This difficulty in deriving OO models from non-OO models is exemplified in the addition of a *Customer* class in the design CD, which was not present in the SCM. The arose from analysis of the *Service Order* which also stores the *Customer* data. This data was then factored out to constitute a separate class (*Customer*).

5.3.5 The activity chart model

The design level activity diagram model for **Alpha's** service activation system is presented in Figure 5.5. Unlike the analysis level diagrams (SCM of Figure 5.1, DCM of Figure 5.2, and CCM of Figure 5.3) and the design CD of Figure 5.4 for *Alpha*'s SPS, the domain activity diagram presented in Figure 5.5 was developed with immense input from the domain experts.

The sub-tasks *Receive Service Order*, *Validate Service Order*, *Translate Service Order*, *Create Service Description*, *Store and Schedule Order and Description*, *Generate Atomic Service*, *Determine Route and Send Atomic Service to NEP* and *Execute Atomic Service* are all too specific to **Alpha**'s operations to refine a domain representation for each of

Table 5.5: **Alpha, Inc.** domain dictionary (static concepts)

Term	*Property*	*Example*
service order	abstract, mandatory *association* customer, service description, service description *attribute* due_date process_date, order_date, order_info	wireless service
DSL	concrete, optional *specialization* service order	data transfer
wireless service	concrete, optional *specialization* service order *attribute* wireless_no	pager
repair verification	concrete, optional *specialization* service order *attribute* repair_type, repair_no	check service is on
call forwarding	concrete, optional *specialization* service order *attribute* forward_no, from_no	forward POTS to pager
call blocking	concrete, optional *specialization* service order *attribute* block_no	block 1 900 numbers
ISDN	concrete, optional *specialization* service order	Internet and POTS
number portability	concrete, optional *specialization* service order *attribute* port_no	change address
POTS	concrete, optional *specialization* service order *attribute* pots_no	POTS
ring cycle control	concrete, optional *specialization* service order *attribute* cycle, cycle_no	forward on 3^{rd} ring
toll free call	concrete, optional *specialization* service order *attribute* toll_no	1 800 number
calling card	concrete, optional *specialization* service order *attribute* card_no	phone card
credit card bill	concrete, optional *specialization* service order *attribute* card_no, bill_no	bill to VISA card
call waiting	concrete, optional *specialization* service order *attribute* wait_no	interrupt call
interactive order	concrete, optional *specialization* service order *attribute* wk_station	failed order
non-interactive order	concrete, optional *specialization* service order *attribute* logon_no	non-failed order
network element	concrete, mandatory *association* atomic service	network switch
service description	concrete, mandatory *association* service order, atomic service *attribute* command	load next command
atomic service	concrete, mandatory *association* service description, state table	set switch on
state table	concrete, optional *association* atomic service	database
customer	concrete, mandatory *association* service order *attribute* h_name, h_address, b_name, b_address	new customer

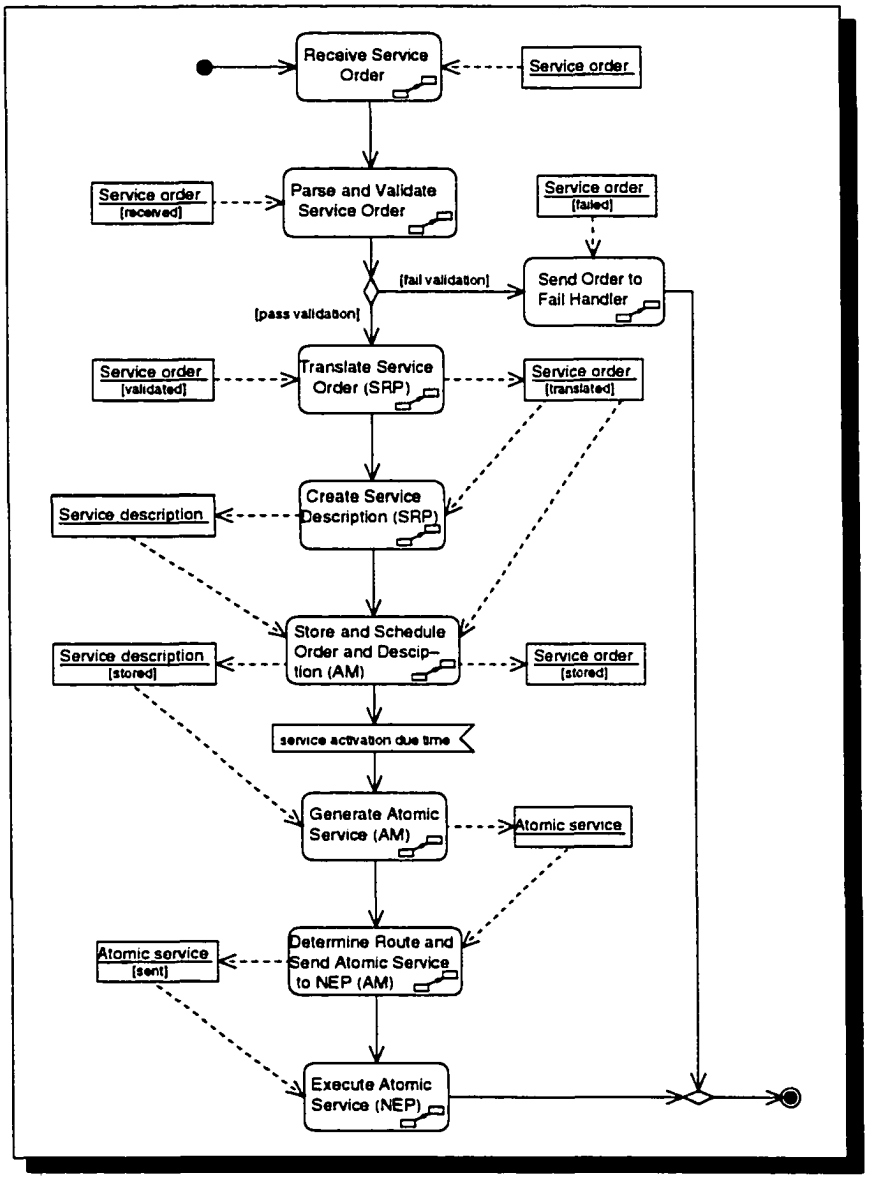

Figure 5.5: Activity diagram for **Alpha, Inc.** service activation system

them. Consequently, as no other telecommunication enterprise service provisioning system is available for this exercise Figure 5.5 is the single domain activity diagram developed for the domain.

5.3.6 The sequence diagram model

The domain sequence diagram of Figure 5.3.6 was developed from the domain activity diagram of Figure 5.5, and the domain design CD of Figure 5.4. The method calls of the domain sequence diagrams are based on an OO design, and though the sequence are actual representation of **Alpha**'s service activation process flow the association of methods to objects is an imposition of the OO approach.

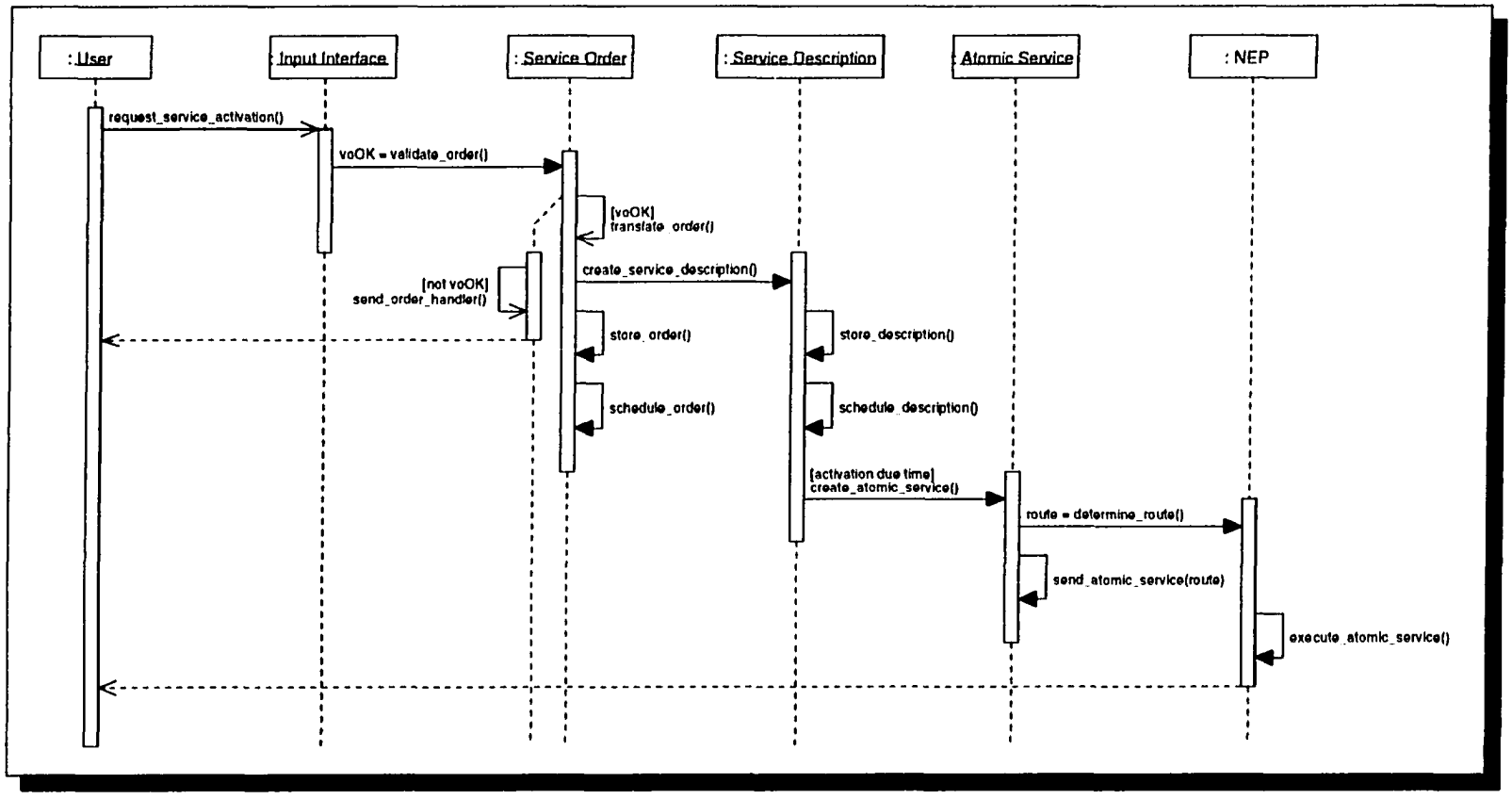

Figure 5.3.6: Sequence diagram for **Alpha, Inc.** service provisioning system

Figure 5.3.6 is but one possible distribution of methods between the possible classes that may be defined within a telecommunication service provisioning system. The definition of OO classes and assignment of methods to these classes for **Alpha**'s SPS was conducted without direct input from the domain engineers. The definition of classes and assignment of methods were based on the information contained in the system documentation, and interviews conducted with the domain engineers during the DA&D activity. The primary purpose for the DA&D activity was the creation of a set of activity diagrams (including that of Figure 5.5). The DA&D activity was then extended, at a later time, to produce the analysis models of Figure 5.1, 5.2 and 5.3, and the design models of Figure 5.4 and 5.3.6.

5.4 Developing the Domain Specification Modeling Language

The process for defining a DSML as presented in Chapter 4 and illustrated in Figure 4.2 will be applied in Section 5.4.1 through 5.4.5.

The stereotypes for the static concepts will be defined first in Section 5.4.1, followed by those for the dynamic concepts in Section 5.4.2, and the meta-models in Section 5.4.3.

5.4.1 Creating the SC stereotypes

Following is the static concept (see Figure 5.4) stereotype for **Alpha**'s service provisioning system *Service Order*. Creation of the static concept stereotypes is carried out in an iterative manner with the static concept models (SCM, design class diagram and static concept domain dictionary) and the profile base (the UML specification) as inputs. The process for creating the static concept stereotypes is illustrated in the activity diagram of Figure 4.3. Additional SPS stereotype definitions are given in Section 5.5.3 at the end of this chapter.

Tables of the stereotypes for the attributes of the static concepts (classes) are not illustrated as this is straight forward and can be easily from the owning static concept stereotype specification and the SC dictionary. This is with exception of the of the *order_info* attribute of the *Service Order* class. The stereotype definition of this model element is presented in

Table 5.6: Class *Service Order* stereotype

Stereotype	service order
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪service_order≫ **inv:** self.isAbstract = true **and** self.isLeaf = false **and** self.isRoot = true **and** self.mandatory = true **and** self→isUnique(self.order_no) **and** self.≪customer≫→size() = 1 **and** self.≪service_description≫→size() ≥ 0 **and** self.≪input_interface≫→size() = 1 **and** self.≪order_no≫.multiplicity = 1 **and** self.≪order_no≫.changeability = frozen **and** self.≪order_no≫.visibility = public **and** self.≪due_date≫.multiplicity = 1 **and** self.≪due_date≫.changeability = changeable **and** self.≪due_date≫.visibility = private **and** self.≪order_date≫.multiplicity = 1 **and** self.≪order_date≫.changeability = changeable **and** self.≪order_date≫.visibility = private **and** self.≪process_date≫.multiplicity = 1 **and** self.≪process_date≫.changeability = changeable **and** self.≪process_date≫.visibility = private **and** self.≪order_info≫.multiplicity = 1 **and** self.≪order_info≫.changeability = changeable **and** self.≪order_info≫.visibility = private **and** [All specialized class attributes are specified as follows] ((self.≪wk_station≫.multiplicity = 1 **and** self.≪wk_station≫.changeability = frozen **and** self.≪wk_station≫.visibility = public) **xor** (self.≪logon_no≫.multiplicity = 1 **and** self.≪logon_no≫.changeability = frozen **and** self.≪logon_no≫.visibility = public)) **and** ((self.≪wireless_no≫.multiplicity = 1 **and** self.≪wireless_no≫.changeability = frozen **and** self.≪wireless_no≫.visibility = public) **xor** (self.≪pots_no≫.multiplicity = 1 **and** self.≪pots_no≫.changeability = frozen **and** self.≪pots_no≫.visibility = public) **xor** ... **and** self→forAll(so | so.order_info: Sequence(String) **and** so.order_info→size() > 1 **and** (so.due_date ≥ so.process_date) **and** (so.process_date ≥ so.order_date)) **and** self.operations→includesAll(validate_order, translate_order, send_order_handler, store_order, schedule_order)
Description	The *constraint* specifies the meta-attributes, multiplicities and operations of the stereotype.

Table 5.36 as its specification adds to the meaning (semantics) of the owning class (*Service Order*), and it is not as intuitively discerned as the other attributes presented in Figure 5.4.

5.4.2 Creating the DC stereotypes

Each use case identified in the domain dynamic concept dictionary, DCM (see Figure 5.2, and the CCM (see Figure 5.3 is specified as a stereotyped use case for the purpose of reusability. Definition of the operation stereotypes is carried out by first identifying the class that owns the operation. This information is obtained from the domain sequence diagram (see Figure 5.5. The description of the operation is first specified in concise informal text. The format of this description is similar to that of the informal DC (use case) constraints. Inputs to this task are the domain class diagram, sequence diagram, and use cases. A couple of the DC stereotypes are illustrated *Request Service Provision* - Table 5.7 and *Request Service Activation* - Table 5.8. An extensive listing of DC stereotypes are given in Section 5.5.1 at the end of this chapter.

Table 5.7: Use Case: *Request Service Provision* stereotype

Stereotype	request service provision
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** Request_Service_Provision **inv:** self.initiator = user **and** self.mandatory = false **and** ((self **implies** including()) **and** (including() **implies** self) **and** self.operations→includesAll(service_activation, validate_order, send_order_handler, translate_order, create_service_description, store_order, schedule_order, store_description, schedule_description, create_atomic_service, determine_route, send_atomic_service, execute_atomic_service) **and** self.Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)))
Description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order.

The operations identified in the domain sequence diagram of Figure 5.3.6 were derived from analysis of the available **Alpha** SPS documentation. These operations were not explicitly identified in the documentation, but were inferred from the containing information. Consequently, the stereotype definitions of these operations are limited to the specification of some of their meta-attribute values, and to the identification of their formal semantic equivalent, i.e. the mapping of these operations into a formal semantic domain (the UML OCL

operations).

Table 5.8: Operation *Request Service Activation* stereotype

Stereotype	request_service_activation
Base Class	Operation
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** request_service_activation **inv:** self.ownerScope = instance **and** self.visibility = private **and** self.isQuery = false **and** self.isAbstract = false **and** self.isLeaf = false **and** self.isRoot = false **and** self.mandatory = true **and** ((self **implies** (union()) **and** (union() **implies** self)) **and** sequence_diagram::service_provisioning_system→includes(self)
Description	The *constraint* specifies the meta-attributes, and context of the stereotyped operation.

See Section 5.5.4 for additional SPS operation stereotype specifications.

5.4.2.1 Packaging the stereotypes

The packaging of the stereotypes involves the definition of a name-spaces for the three sets of stereotypes (static concepts, use cases, and operations). The UML name-space concept ≪*profile*≫ is used to package the domain stereotypes, as illustrated in Figure 5.6. The sets of stereotypes are individually packaged in profiles and these profiles are then packaged as a profile of profiles (*Domain Stereotypes*).

5.4.3 Creating the meta-models

The process for creating the meta-models of the domain models calls for: (1) the definition of domain-specific graphical icons, and (2) the establishment of the relationships between the icons. In this domain (service provisioning systems) domain-specific graphical icons are not created as such information was not obtained from the domain experts. Consequently, the default UML icons are used. The association are then established between the default graphical icons that represents the stereotypes of Table 5.24 through 5.36. These meta-models are illustrated in Figure 5.7 (the static concepts), Figure 5.8 (the dynamic concepts),

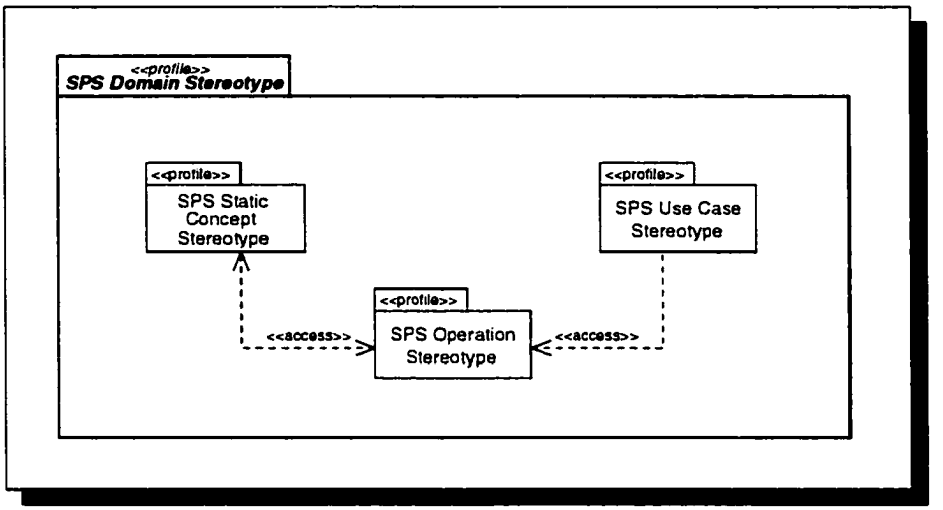

Figure 5.6: SPS DSML stereotype profile packaging

Figure 5.9 (the concepts context), Figure 5.10 (the design class diagram), Figure 5.11 (the activity diagram), and Figure 5.4.3 (the sequence diagram).

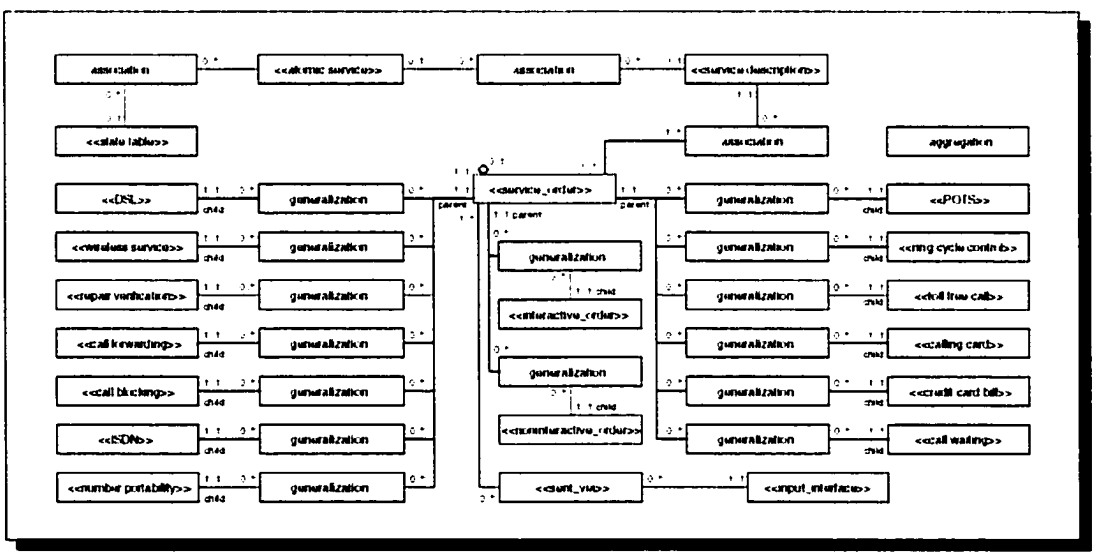

Figure 5.7: SPS DSML static concept meta-model

These meta models were derived from the domain models of Figure 5.1 (the static concepts), Figure 5.2 (the dynamic concepts), Figure 5.3 (the concept context), Figure 5.4 (the design class diagram), Figure 5.5 (the activity diagram), and Figure 5.3.6 (the sequence

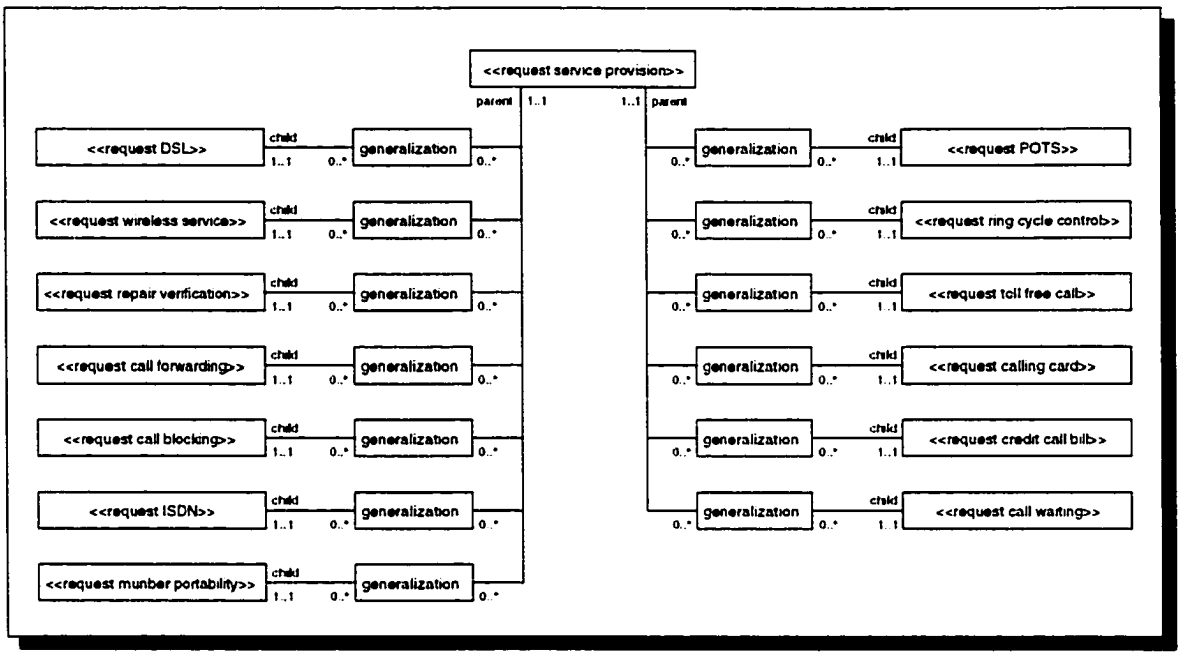

Figure 5.8: SPS DSML dynamic concept meta-model

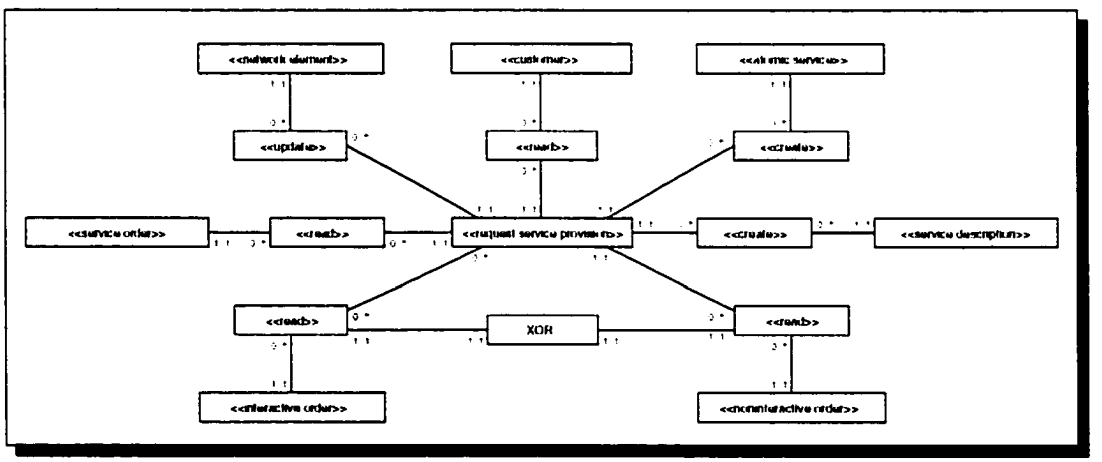

Figure 5.9: SPS DSML concept context meta-model

diagram), respectively.

5.4.4 Packaging the meta-models

The packaging of the meta-models involves the definition of a name-spaces for the six meta-models (static concepts, dynamic concepts, concept context, design CD, activity diagram

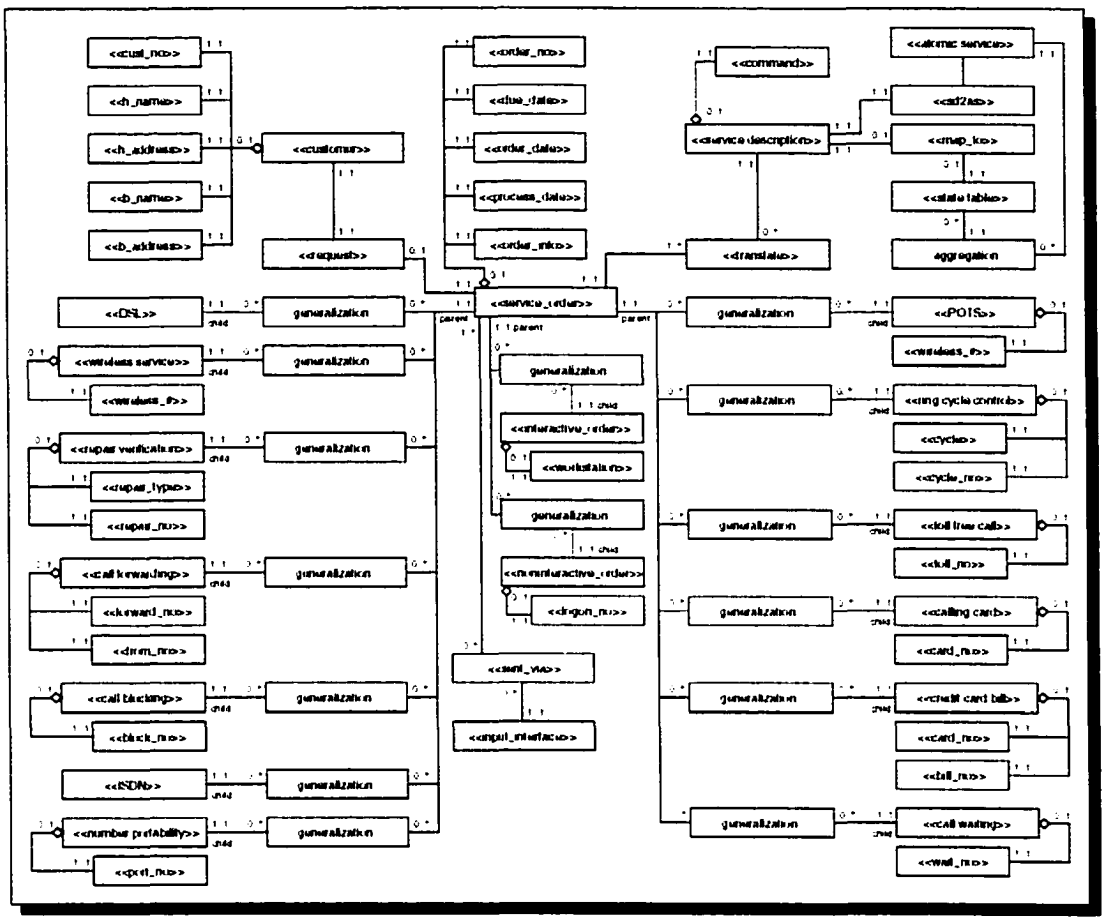

Figure 5.10: SPS DSML design class diagram meta-model

and sequence diagram). The UML name-space concept ≪*profile*≫ is used to package the domain meta-models, as illustrated in Figure 5.12. The meta-models are individually packaged in profiles and these profiles are then packaged as a profile of profiles SPS (*Domain Meta-Models*).

5.4.5 Packaging the DSML

The packaging of the DSML involves the definition of a name-spaces for the stereotype profile (*Domain Stereotype*, and the meta-model profile (*Domain Meta-Model*). The UML name-space concept ≪*profile*≫ is used to package the domain stereotypes, and meta-models, as illustrated in Figures 5.6, and 5.12. These individually packaged profiles are then packaged as a profile of profiles (*DSML*) as illustrated in Figure 5.13.

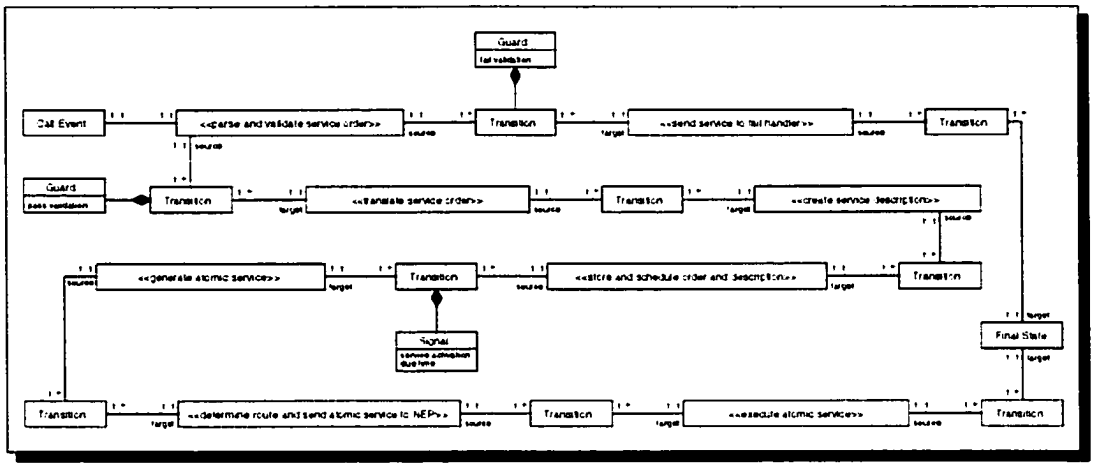

Figure 5.11: DSML activity diagram meta-model

This concludes the creation of the domain-specific modeling language (DSML) for **Alpha**'s service provisioning system (SPS).

Figure 5.4.3: SPS DSML sequence diagram meta-model

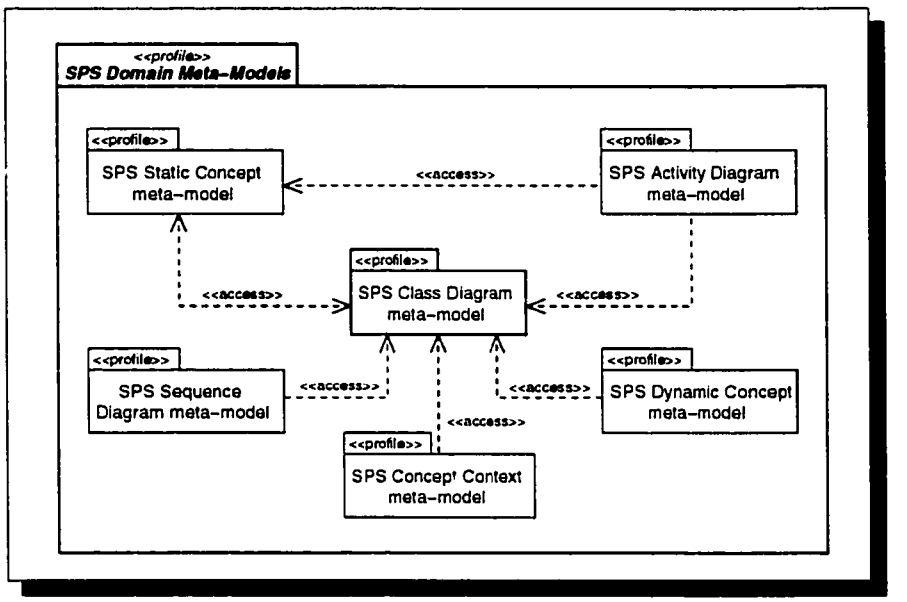

Figure 5.12: SPS DSML meta-model profile packaging

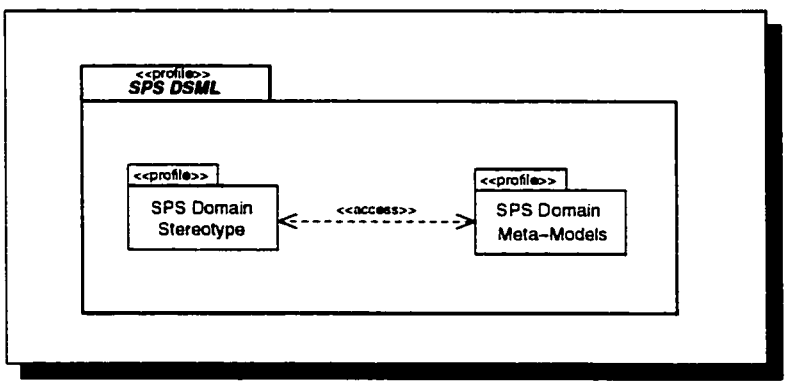

Figure 5.13: DSML profile packaging

5.5 Service Provisioning System DSML Specification

The following sub-sections contain the complete specification for the **Alpha, Inc.** service provisioning system (SPS).

5.5.1 The dynamic conceptual model

The dynamic concept model (DCM) for **Alpha's** service provisioning system is presented in Figure 5.14.

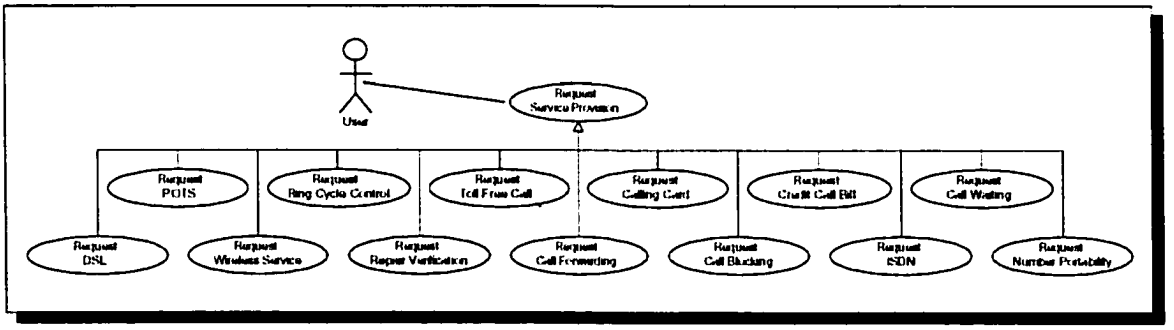

Figure 5.14: DCM for **Alpha, Inc.** service activation system

The initial dynamic concept dictionary (DCD) for **Alpha**'s service provisioning system is presented in Table 5.9.

The DCD and SCD are updated at the end of the DA activity to include all information that has been elicited during the DA process about the concepts.

5.5.2 The use cases

The following use case specifications (*Request POTS, Request Ring Cycle Control, Request Toll Free Call, Request Calling Card, Request Credit Call Bill, Request Call Waiting, Request DSL, Request Wireless Service, Request Repair Verification, Request Call Forwarding, Request Call Blocking, Request ISDN,* and *Request Number Portability*) in Tables 5.11 to 5.23 are specializations of the *Request Service Provision* of Table 5.10.

Table 5.9: **Alpha**'s domain dictionary (dynamic concepts)

Term	*Property*	*Example*
request service provision	abstract, optional	
request POTS	concrete, optional	request POTS
request ring cycle control	concrete, optional *specification* Customer must exist and Customer is associated with ring cycle type service.	forward on 3^{rd} ring
request toll free call	concrete, optional *specification* Customer must exist and Customer is associated with toll free type service.	request 1 800 number
request calling card	concrete, optional *specification* Customer must exist and Customer is associated with calling card type service.	request phone card
request credit card bill	concrete, optional *specification* Customer must exist and Customer is associated with credit card.	request bill VISA card
request call waiting	concrete, optional *specification* Customer must exist and Customer is associated with call waiting type service.	request call interrupt
request DSL	concrete, optional *specification* Customer must exist and Customer is associated with DSL type service.	data transfer
request wireless service	concrete, optional	data transfer
request repair verification	concrete, optional *specification* Customer must exist and repair service was previously requested.	check service is on
request call forwarding	concrete, optional *specification* Customer must exist and Customer is associated with call forwarding type service.	request POTS forward to pager
request call blocking	concrete, optional *specification* Customer must exist and Customer is associated with call blocking type service.	request block 1 900 numbers
request ISDN	concrete, optional *specification* Customer must exist and Customer is associated with ISDN type service.	request Internet and POTS
request number portability	concrete, optional *specification* Customer must exist and Customer is associated with number portability type service.	request address change

Table 5.10: Use Case: *Request Service Provision*

Use Case	Request Service Provision
Generalization	
Specialization	Request Service Provision, Request POTS, Request Ring Cycle Control, Request Toll Free Call, Request Calling Card, Request Credit Call Bill, Request Call Waiting, Request DSL, Request Wireless Service, Request Repair Verification, Request Call Forwarding, Request Call Blocking, Request ISDN, Request Number Portability
Read	Customer, Service Order, Interactive Order, Non-interactive Order
Update	Network Element
Create	Atomic Service, Service Description
Delete	
Input	serv: Service Order, cust: Customer
Output	stab: State Table
Formal description	**context** Request_Service_Provision **inv:** (Network_Element.oclIsKindOf(Class) **and** Interactive_Order.oclIsKindOf(Class) **and** Non-interactive_Order.oclIsKindOf(Class) **and** Atomic_Service.oclIsKindOf(Class) **and** Service_Description.oclIsKindOf(Class) **and** Service_Order.oclIsKindOf(Class) **and** Customer.oclIsKindOf(Class) **and** Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)))
Informal description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order.

Table 5.11: Use Case: *Request POTS*

Use Case	Request POTS
Generalization	Request Service Provision
Specialization	
Read	Customer, Service Order, Interactive Order, Non-interactive Order
Update	Network Element
Create	Atomic Service, Service Description
Delete	
Input	pno: POTS, cust: Customer
Output	stab: State Table
Formal description	**context** Request_POTS **inv:** (Network_Element.oclIsKindOf(Class) **and** Interactive_Order.oclIsKindOf(Class) **and** Non-interactive_Order.oclIsKindOf(Class) **and** Atomic_Service.oclIsKindOf(Class) **and** Service_Description.oclIsKindOf(Class) **and** Service_Order.oclIsKindOf(Class) **and** Customer.oclIsKindOf(Class) **and** Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)) **and** POTS→forAll(ps | ps.pots_no@pre <> pno.pots_no))
Informal description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order **and** the number to be provisioned (pno.pots_no) must not exist at the start.

Table 5.12: Use Case: *Request Ring Cycle Control*

Use Case	Request Ring Cycle Control
Generalization	Request Service Provision
Specialization	
Read	Customer, Service Order, Interactive Order, Non-interactive Order
Update	Network Element
Create	Atomic Service, Service Description
Delete	
Input	rcc: Ring Cycle Control, cust: Customer
Output	stab: State Table
Formal description	**context** Request_Ring_Cycle_Control **inv:** (Network_Element.oclIsKindOf(Class) **and** Interactive_Order.oclIsKindOf(Class) **and** Non-interactive_Order.oclIsKindOf(Class) **and** Atomic_Service.oclIsKindOf(Class) **and** Service_Description.oclIsKindOf(Class) **and** Service_Order.oclIsKindOf(Class) **and** Customer.oclIsKindOf(Class) **and** Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)) **and** pno.cycle $\geq$ 0 **and** (POTS→exist(ps | ps.pots_no@pre = rcc.cycle_no) **xor** Wireless_Service→exists(ws | ws.wireless_no@pre = rcc.cycle_no) **xor** Toll_Free_Call→exists(tfc | tfc.toll_no@pre = rcc.cycle_no)))
Informal description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order **and** the number to be provisioned (rcc.cycle_no) must exist at the start.

Table 5.13: Use Case: *Request Toll Free Call*

Use Case	Request Toll Free Call
Generalization	Request Service Provision
Specialization	
Read	Customer, Service Order, Interactive Order, Non-interactive Order
Update	Network Element
Create	Atomic Service, Service Description
Delete	
Input	tfc: Toll Free Call, cust: Customer
Output	stab: State Table
Formal description	**context** Request_Toll_Free_Call **inv:** (Network_Element.oclIsKindOf(Class) **and** Interactive_Order.oclIsKindOf(Class) **and** Non-interactive_Order.oclIsKindOf(Class) **and** Atomic_Service.oclIsKindOf(Class) **and** Service_Description.oclIsKindOf(Class) **and** Service_Order.oclIsKindOf(Class) **and** Customer.oclIsKindOf(Class) **and** Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)) **and** Toll_Free_Call→forAll(tfc | tfc.toll_no@pre <> tfc.toll_no))
Informal description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order **and** the number to be provisioned (tfc.toll_no) must not exist at the start.

Table 5.14: Use Case: *Request Calling Card*

Use Case	Request Calling Card
Generalization	Request Service Provision
Specialization	
Read	Customer, Service Order, Interactive Order, Non-interactive Order
Update	Network Element
Create	Atomic Service, Service Description
Delete	
Input	cc: Calling Card, cust: Customer
Output	stab: State Table
Formal description	**context** Request_Calling_Card **inv:** (Network_Element.oclIsKindOf(Class) **and** Interactive_Order.oclIsKindOf(Class) **and** Non-interactive_Order.oclIsKindOf(Class) **and** Atomic_Service.oclIsKindOf(Class) **and** Service_Description.oclIsKindOf(Class) **and** Service_Order.oclIsKindOf(Class) **and** Customer.oclIsKindOf(Class) **and** Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)) **and** Calling_Card→forAll(cd | cd.card_no@pre <> cc.card_no))
Informal description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order **and** the number to be provisioned (cc.card_no) must not exist at the start.

Table 5.15: Use Case: *Request Credit Card Bill*

Use Case	Request Credit Card Bill
Generalization	Request Service Provision
Specialization	
Read	Customer, Service Order, Interactive Order, Non-interactive Order
Update	Network Element
Create	Atomic Service, Service Description
Delete	
Input	ccb: Credit Card Bill, cust: Customer
Output	stab: State Table
Formal description	**context** Request_Credit_Card_Bill **inv:** (Network_Element.oclIsKindOf(Class) **and** Interactive_Order.oclIsKindOf(Class) **and** Non-interactive_Order.oclIsKindOf(Class) **and** Atomic_Service.oclIsKindOf(Class) **and** Service_Description.oclIsKindOf(Class) **and** Service_Order.oclIsKindOf(Class) **and** Customer.oclIsKindOf(Class) **and** Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)) **and** ccb.card_no.size() = 16 **and** (POTS→exist(ps | ps.pots_no@pre = ccb.bill_no) **xor** Wireless_Service→exists(ws | ws.wireless_no@pre = ccb.bill_no) **xor** Toll_Free_Call→exists(tfc | tfc.toll_no@pre = ccb.bill_no)))
Informal description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order **and** the number to be provisioned (rcc.cycle_no) must exist at the start.

Table 5.16: Use Case: *Request Call Waiting*

Use Case	Request Call Waiting
Generalization	Request Service Provision
Specialization	
Read	Customer, Service Order, Interactive Order, Non-interactive Order
Update	Network Element
Create	Atomic Service, Service Description
Delete	
Input	cw: Call Waiting, cust: Customer
Output	stab: State Table
Formal description	**context** Request_Call_Waiting **inv:** (Network_Element.oclIsKindOf(Class) **and** Interactive_Order.oclIsKindOf(Class) **and** Non-interactive_Order.oclIsKindOf(Class) **and** Atomic_Service.oclIsKindOf(Class) **and** Service_Description.oclIsKindOf(Class) **and** Service_Order.oclIsKindOf(Class) **and** Customer.oclIsKindOf(Class) **and** Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)) **and** (POTS→exist(ps | ps.pots_no@pre = cw.wait_no) **xor** Wireless_Service→exists(ws | ws.wireless_no@pre = cw.wait_no) **xor** Toll_Free_Call→exists(tfc | tfc.toll_no@pre = cw.wait_no)) **and** stab.service_Description.call_Waiting→includes(cw)))
Informal description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order **and** the number to be provisioned (wc.wait_no) must exist at the start.

Table 5.17: Use Case: *Request DSL*

Use Case	Request DSL
Generalization	Request Service Provision
Specialization	
Read	Customer, Service Order, Interactive Order, Non-interactive Order
Update	Network Element
Create	Atomic Service, Service Description
Delete	
Input	dsl: DSL, cust: Customer
Output	stab: State Table
Formal description	**context** Request_DSL **inv:** (Network_Element.oclIsKindOf(Class) **and** Interactive_Order.oclIsKindOf(Class) **and** Non-interactive_Order.oclIsKindOf(Class) **and** Atomic_Service.oclIsKindOf(Class) **and** Service_Description.oclIsKindOf(Class) **and** Service_Order.oclIsKindOf(Class) **and** Customer.oclIsKindOf(Class) **and** Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)))
Informal description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order

It will be noted that the use case specifications of Table 5.11 through 5.23 contain informa-

Table 5.18: Use Case: *Request Wireless Service*

Use Case	Request Wireless Service
Generalization	Request Service Provision
Specialization	
Read	Customer, Service Order, Interactive Order, Non-interactive Order
Update	Network Element
Create	Atomic Service, Service Description
Delete	
Input	ws: Wireless service, cust: Customer
Output	stab: State Table
Formal description	**context** Request_Wireless_Service **inv:** (Network_Element.oclIsKindOf(Class) **and** Interactive_Order.oclIsKindOf(Class) **and** Non-interactive_Order.oclIsKindOf(Class) **and** Atomic_Service.oclIsKindOf(Class) **and** Service_Description.oclIsKindOf(Class) **and** Service_Order.oclIsKindOf(Class) **and** Customer.oclIsKindOf(Class) **and** Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)) **and** (Wireless_Service→forAll(wl | wl.wireless_no@pre <> ws.wireless_no)
Informal description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order **and** the number to be provisioned (ws.wireless_no) must not exist at the start.

Table 5.19: Use Case: *Request Repair Verification*

Use Case	Request Repair Verification
Generalization	Request Service Provision
Specialization	
Read	Customer, Service Order, Interactive Order, Non-interactive Order
Update	Network Element
Create	Atomic Service, Service Description
Delete	
Input	rv: Repair Verification, cust: Customer
Output	stab: State Table
Formal description	**context** Request_Repair_Verification **inv:** (Network_Element.oclIsKindOf(Class) **and** Interactive_Order.oclIsKindOf(Class) **and** Non-interactive_Order.oclIsKindOf(Class) **and** Atomic_Service.oclIsKindOf(Class) **and** Service_Description.oclIsKindOf(Class) **and** Service_Order.oclIsKindOf(Class) **and** Customer.oclIsKindOf(Class) **and** Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)) **and** (POTS→exist(ps | ps.pots_no@pre = rv.repair_no)))
Informal description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order **and** the number to be repaired (rv.repair_no) must exist at the start.

tion, such as class attributes, which would not have been available at the point when these use case specifications were initially created. The contents of the specification, as presented in Tables 5.16 to 5.23 was obtained by updating the initial use case specifications after the CD of Figure 5.4 was completed. Hence, Tables 5.16 to 5.23 represents the final iteration

Table 5.20: Use Case: *Request Call Forwarding*

Use Case	Request Call Forwarding
Generalization	Request Service Provision
Specialization	
Read	Customer, Service Order, Interactive Order, Non-interactive Order
Update	Network Element
Create	Atomic Service, Service Description
Delete	
Input	cf: Call Forwarding, cust: Customer
Output	stab: State Table
Formal description	**context** Request_Call_Forwarding **inv:** (Network_Element.oclIsKindOf(Class) **and** Interactive_Order.oclIsKindOf(Class) **and** Non-interactive_Order.oclIsKindOf(Class) **and** Atomic_Service.oclIsKindOf(Class) **and** Service_Description.oclIsKindOf(Class) **and** Service_Order.oclIsKindOf(Class) **and** Customer.oclIsKindOf(Class) **and** Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)) **and** (POTS→exist(ps | ps.pots_no@pre = cf.forward_no) **xor** Wireless_Service→exists(ws | ws.wireless_no@pre = cf.forward_no) **xor** Toll_Free_Call→exists(tfc | tfc.toll_no@pre = cf.forward_no)) **and** (POTS→exist(ps | ps.pots_no@pre = cf.from_no) **xor** Wireless_Service→exists(ws | ws.wireless_no@pre = cf.from_no) **xor** Toll_Free_Call→exists(tfc | tfc.toll_no@pre = cf.from_no)))
Informal description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order **and** the numbers to be provisioned (wf.forward_no and wf.from_no) must exist at the start.

Table 5.21: Use Case: *Request Call Blocking*

Use Case	Request Call Blocking
Generalization	Request Service Provision
Specialization	
Read	Customer, Service Order, Interactive Order, Non-interactive Order
Update	Network Element
Create	Atomic Service, Service Description
Delete	
Input	cb: Call Blocking, cust: Customer
Output	stab: State Table
Formal description	**context** Request_Call_Blocking **inv:** (Network_Element.oclIsKindOf(Class) **and** Interactive_Order.oclIsKindOf(Class) **and** Non-interactive_Order.oclIsKindOf(Class) **and** Atomic_Service.oclIsKindOf(Class) **and** Service_Description.oclIsKindOf(Class) **and** Service_Order.oclIsKindOf(Class) **and** Customer.oclIsKindOf(Class) **and** Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)))
Informal description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order.

of the use case specification process.

Table 5.22: Use Case: *Request ISDN*

Use Case	Request ISDN
Generalization	Request Service Provision
Specialization	
Read	Customer, Service Order, Interactive Order, Non-interactive Order
Update	Network Element
Create	Atomic Service, Service Description
Delete	
Input	isd: ISDN, cust: Customer
Output	stab: State Table
Formal description	**context** Request_ISDN **inv:** (Network_Element.oclIsKindOf(Class) **and** Interactive_Order.oclIsKindOf(Class) **and** Non-interactive_Order.oclIsKindOf(Class) **and** Atomic_Service.oclIsKindOf(Class) **and** Service_Description.oclIsKindOf(Class) **and** Service_Order.oclIsKindOf(Class) **and** Customer.oclIsKindOf(Class) **and** Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)))
Informal description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order

Table 5.23: Use Case: *Request Number Portability*

Use Case	Request Number Portability
Generalization	Request Service Provision
Specialization	
Read	Customer, Service Order, Interactive Order, Non-interactive Order
Update	Network Element
Create	Atomic Service, Service Description
Delete	
Input	np: Number Portability, cust: Customer
Output	stab: State Table
Formal description	**context** Request_Number_Portability **inv:** (Network_Element.oclIsKindOf(Class) **and** Interactive_Order.oclIsKindOf(Class) **and** Non-interactive_Order.oclIsKindOf(Class) **and** Atomic_Service.oclIsKindOf(Class) **and** Service_Description.oclIsKindOf(Class) **and** Service_Order.oclIsKindOf(Class) **and** Customer.oclIsKindOf(Class) **and** Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)) **and** (POTS→exist(ps | ps.pots_no@pre = np.port_no) **xor** Wireless_Service→exists(ws | ws.wireless_no@pre = np.port_no) **xor** Toll_Free_Call→exists(tfc | tfc.toll_no@pre = pn.port_no)) **and** stab.service_Description.call_Waiting→includes(cw)))
Informal description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order **and** the number to be provisioned (pn.port_no) must exist at the start.

5.5.3 SPS SC stereotypes

Following are the static concept (see Figure 5.4) stereotypes for **Alpha**'s service provisioning system **Service Order**. Creation of the static concept stereotypes is carried out in an iterative manner with the static concept models (SCM, design class diagram and static concept domain dictionary) and the profile base (the UML specification) as inputs. The process for creating the static concept stereotypes is illustrated in the activity diagram of Figure 4.3.

Tables of the stereotypes for the attributes of the static concepts (classes) are not illustrated as this is straight forward and can be easily from the owning static concept stereotype specification and the SC dictionary. This is with exception of the of the *order_info* attribute of the *Service Order* class. The stereotype definition of this model element is presented in Table 5.36 as its specification adds to the meaning (semantics) of the owning class (*Service Order*), and it is not as intuitively discerned as the other attributes presented in Figure 5.4.

Table 5.24: Class *Service Order* stereotype

Stereotype	service order
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪service_order≫ **inv:** self.isAbstract = true **and** self.isLeaf = false **and** self.isRoot = true **and** self.mandatory = true **and** self→isUnique(self.order_no) **and** self.≪customer≫→size() = 1 **and** self.≪service_description≫→size() ≥ 0 **and** self.≪input_interface≫→size() = 1 **and** self.≪order_no≫.multiplicity = 1 **and** self.≪order_no≫.changeability = frozen **and** self.≪order_no≫.visibility = public **and** self.≪due_date≫.multiplicity = 1 **and** self.≪due_date≫.changeability = changeable **and** self.≪due_date≫.visibility = private **and** self.≪order_date≫.multiplicity = 1 **and** self.≪order_date≫.changeability = changeable **and** self.≪order_date≫.visibility = private **and** self.≪process_date≫.multiplicity = 1 **and** self.≪process_date≫.changeability = changeable **and** self.≪process_date≫.visibility = private **and** self.≪order_info≫.multiplicity = 1 **and** self.≪order_info≫.changeability = changeable **and** self.≪order_info≫.visibility = private **and** [All specialized class attributes are specified as follows] ((self.≪wk_station≫.multiplicity = 1 **and** self.≪wk_station≫.changeability = frozen **and** self.≪wk_station≫.visibility = public) **xor** (self.≪logon_no≫.multiplicity = 1 **and** self.≪logon_no≫.changeability = frozen **and** self.≪logon_no≫.visibility = public)) **and** ((self.≪wireless_no≫.multiplicity = 1 **and** self.≪wireless_no≫.changeability = frozen **and** self.≪wireless_no≫.visibility = public) **xor** (self.≪pots_no≫.multiplicity = 1 **and** self.≪pots_no≫.changeability = frozen **and** self.≪pots_no≫.visibility = public) **xor** ... **and** self→forAll(so | so.order_info: Sequence(String) **and** so.order_info→size() > 1 **and** (so.due_date ≥ so.process_date) **and** (so.process_date ≥ so.order_date)) **and** self.operations→includesAll(validate_order, translate_order, send_order_handler, store_order, schedule_order)
Description	The *constraint* specifies the meta-attributes, multiplicities and operations of the stereotype.

Table 5.25: Tag definition *Mandatory*

Tag	mandatory
Stereotype	item
Type	UML::Datatype::Boolean
Multiplicity	1
Description	Indicates whether a stereotyped class with this tag must be included in a domain application model.

Table 5.26: Class *Service Description* stereotype

Stereotype	service description
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪service_description≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self→isUnique(self.command) **and** self.≪service_order≫→size() ≥ 0 **and** self.≪atomic_service≫→size() ≥ 0 **and** ((self.≪state_table≫→size() = 0) **xor** (self.≪state_table≫→size() = 1)) **and** self.≪command≫.multiplicity = 1 **and** self.≪command≫.changeability = frozen **and** self.≪command≫.visibility = public **and** self→size() = *constant* **and** self.operations→includesAll(create_service_description, store_description schedule_description)
Description	The *constraint* specifies the meta-attributes, multiplicities and operations of the stereotype.

Table 5.27: Class *Atomic Service* stereotype

Stereotype	atomic service
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪atomic_service≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪service_description≫→size() ≥ 0 **and** self.≪state_table≫→size() ≥ 0 **and** self→size() = *constant* **and** self.operations→includesAll(create_atomic_service, send_atomic_service)
Description	The *constraint* specifies the meta-attributes, multiplicities and operations of the stereotype.

Table 5.28: Class *State Table* stereotype

Stereotype	state table
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪state_table≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = false **and** self.≪service_description≫→size() ≥ 0 **and** self.≪atomic_service≫→size() ≥ 0
Description	The *constraint* specifies the meta-attributes, and multiplicities of the stereotype.

Table 5.29: Class *Input Interface* stereotype

Stereotype	input interface
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪input_interface≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = false **and** self.≪service_order≫→size() ≥ 0 **and** self.operations→includesAll(request_service_activation)
Description	The *constraint* specifies the meta-attributes, and multiplicities of the stereotype.

Table 5.30: Class *Customer* stereotype

Stereotype	customer
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪customer≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self→isUnique(self.cust_no) **and** self.≪service_order≫→size() ≥ 0 **and** self.≪cust_no≫.multiplicity = 1 **and** self.≪cust_no≫.changeability = frozen **and** self.≪cust_no≫.visibility = public **and** self.≪h_name≫.multiplicity = 1 **and** self.≪h_name≫.changeability = frozen **and** self.≪h_name≫.visibility = public **and** self.≪h_address≫.multiplicity ≥ 1 **and** self.≪h_address≫.changeability = changeable **and** self.≪h_address≫.visibility = private **and** self.≪b_name≫.multiplicity = 1 **and** self.≪b_name≫.changeability = changeable **and** self.≪b_name≫.visibility = private **and** self.≪b_address≫.multiplicity ≥ 1 **and** self.≪b_address≫.changeability = changeable **and** self.≪b_address≫.visibility = private **and**
Description	The *constraint* specifies the meta-attributes, multiplicities and operations of the stereotype.

Table 5.31: Association *Request* stereotype

Stereotype	request
Base Class	Association
Parent	N/A
Tags	none
Constraints	**context** ≪request≫ **inv:** (self.AssociationEnd.participant.type = ≪customer≫ **or** self.AssociationEnd.participant.type = ≪service_order≫) **and** self.AssociationEnd→forAll(ae: AssociationEnd | ae.participant.type=≪customer≫ **implies** ae.multiplicity = 1..1) **and** self.AssociationEnd→forAll(ae: AssociationEnd | ae.participant.type = ≪service_order≫ **implies** ae.multiplicity = 0..*)
Description	The *constraint* specifies the stereotyped classes that this association may have.

Table 5.32: Association *Translate* stereotype

Stereotype	translate
Base Class	Association
Parent	N/A
Tags	none
Constraints	**context** ≪translate≫ **inv:** (self.AssociationEnd.participant.type = ≪service_description≫ **or** self.AssociationEnd.participant.type = ≪service_order≫) **and** self.AssociationEnd→forAll(ae: AssociationEnd | ae.participant.type = ≪service_description≫ **implies** ae.multiplicity = 0..*) **and** self.AssociationEnd→forAll(ae: AssociationEnd | ae.participant.type = ≪service_order≫ **implies** ae.multiplicity = 0..*)
Description	The *constraint* specifies the stereotyped classes that this association may have.

Table 5.33: Association *SD2AS* stereotype

Stereotype	sd2as
Base Class	Association
Parent	N/A
Tags	none
Constraints	**context** ≪sd2as≫ **inv:** (self.AssociationEnd.participant.type = ≪service_description≫ **or** self.AssociationEnd.participant.type = ≪atomic_service≫) **and** self.AssociationEnd→forAll(ae: AssociationEnd | ae.participant.type = ≪service_description≫ **implies** ae.multiplicity = 0..*) **and** self.AssociationEnd→forAll(ae: AssociationEnd | ae.participant.type = ≪atomic_service≫ **implies** ae.multiplicity = 0..*)
Description	The *constraint* specifies the stereotyped classes that this association may have.

Table 5.34: Association *Map_To* stereotype

Stereotype	store
Base Class	Association
Parent	N/A
Tags	none
Constraints	**context** ≪map_to≫ **inv:** (self.AssociationEnd.participant.type = ≪service_description≫ **or** self.AssociationEnd.participant.type = ≪state_table≫) **and** self.AssociationEnd→forAll(ae: AssociationEnd | ae.participant.type = ≪service_description≫ **implies** ae.multiplicity = 0..1) **and** self.AssociationEnd→forAll(ae: AssociationEnd | ae.participant.type = ≪state_table≫ **implies** ae.multiplicity = 0..*)
Description	The *constraint* specifies the stereotyped classes that this association may have.

Table 5.35: Association *Sent_Via* stereotype

Stereotype	sent_via
Base Class	Association
Parent	N/A
Tags	none
Constraints	**context** ≪sent_via≫ **inv:** (self.AssociationEnd.participant.type = ≪service_order≫ **or** self.AssociationEnd.participant.type = ≪input_interface≫) **and** self.AssociationEnd→forAll(ae: AssociationEnd | ae.participant.type = ≪service_order≫ **implies** ae.multiplicity = 0..*) **and** self.AssociationEnd→forAll(ae: AssociationEnd | ae.participant.type = ≪input_interface≫ **implies** ae.multiplicity = 0..1)
Description	The *constraint* specifies the stereotyped classes that this association may have.

Table 5.36: Association *Order_Info* stereotype

Stereotype	order_info
Base Class	Attribute
Parent	N/A
Tags	none
Constraints	**context** ≪order_info≫ **inv:** self.multiplicity = 1 **and** self.changeability = changeable **and** self.type = string self.operations→includesAll(read_info, write_info)
Description	The *constraint* specifies the type of the stereotype attribute and the operations that may be applied to it.

5.5.4 SPS DC stereotypes

Only three of the use case stereotypes (*Request Service Provision* - Tables 5.37, *Request POTS* - Table 5.38, and *Request Ring Cycle Control* - Table 5.39) are listed, as the definition of the stereotypes is directly derived from the use case specifications of Tables 5.10 through 5.12.

Instances of the following use case stereotype specifications (≪*request POTS*≫, ≪*request ring cycle control*≫, ≪*request toll free call*≫, ≪*request calling card*≫, ≪*request credit call bill*≫, ≪*request call waiting*≫, ≪*request DSL*≫, ≪*request wireless service*≫, ≪*request repair verification*≫, ≪*request call forwarding*≫, ≪*request call blocking*≫, ≪*request ISDN*≫, and ≪*request number portability*≫) are specializations of instances of the ≪*request service provision*≫ stereotype of Table 5.7. Thus they inherit the constraint of the parent (generalization) stereotype ≪*request service provision*≫.

The operations identified in the domain sequence diagram of Figure 5.3.6 were derived from analysis of the available **Alpha** SPS documentation. These operations were not explicitly identified in the documentation, but were inferred from the containing information. Consequently, the stereotype definitions of these operations are limited to the specification of some of their meta-attribute values, and to the identification of their formal semantic equivalent, i.e. the mapping of these operations into a formal semantic domain (the UML OCL operations).

Table 5.37: Use Case: *Request Service Provision* stereotype

Stereotype	request service provision
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** Request_Service_Provision **inv:** self.initiator = user **and** self.mandatory = false **and** ((self **implies** including()) **and** (including() **implies** self) **and** self.operations→includesAll(service_activation, validate_order, send_order_handler, translate_order, create_service_description, store_order, schedule_order, store_description, schedule_description, create_atomic_service, determine_route, send_atomic_service, execute_atomic_service) **and** self.Customer→exists(c | c.cust_no@pre = cust.cust_no) **and** (Interactive_Order→exists(i | serv.order_no = i.order_no@pre) **xor** Non-interactive_Order→exists(ni | serv.order_no = ni.order_no@pre)))
Description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order.

Table 5.38: Use Case: *Request POTS* stereotype

Stereotype	request POTS
Base class	Use Case
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** Request_POTS **inv:** POTS→forAll(ps | ps.pots_no@pre <> pno.pots_no))
Description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order **and** the number to be provisioned (pno.pots_no) must not exist at the start.

Table 5.39: Use Case: *Request Ring Cycle Control* stereotype

Stereotype	request ring cycle control
Base class	Use Case
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** Request_Ring_Cycle_Control **inv:** pno.cycle $\geq$ 0 **and** (POTS→exist(ps | ps.pots_no@pre = rcc.cycle_no) **xor** Wireless_Service→exists(ws | ws.wireless_no@pre = rcc.cycle_no) **xor** Toll_Free_Call→exists(tfc | tfc.toll_no@pre = rcc.cycle_no))
Description	A Customer (cust) must be associated with the Service Order (serv) **and** the order must be either an Interactive Order or a Non-interactive Order **and** the number to be provisioned (rcc.cycle_no) must exist at the start.

Table 5.40: Operation *Request Service Activation* stereotype

Stereotype	request_service_activation
Base Class	Operation
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** request_service_activation **inv:** self.ownerScope = instance **and** self.visibility = private **and** self.isQuery = false **and** self.isAbstract = false **and** self.isLeaf = false **and** self.isRoot = false **and** self.mandatory = true **and** ((self **implies** (union()) **and** (union() **implies** self)) **and** sequence_diagram::service_provisioning_system→includes(self)
Description	The *constraint* specifies the meta-attributes, and context of the stereotyped operation.

Table 5.41: Operation *Validate Order* stereotype

Stereotype	validate_order
Base Class	Operation
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** validate_order **inv:** self.ownerScope = instance **and** self.visibility = private **and** self.isQuery = false **and** self.isAbstract = false **and** self.isLeaf = false **and** self.isRoot = false **and** self.mandatory = true **and** ((self **implies** (isQuery()) **and** (isQuery() **implies** self)) **and** sequence_diagram::service_provisioning_system→includes(self)
Description	The *constraint* specifies the meta-attributes, and context of the stereotyped operation.

Table 5.42: Operation *Send Order Handler* stereotype

Stereotype	send_order_handler
Base Class	Operation
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** send_order_handler **inv:** self.ownerScope = instance **and** self.visibility = private **and** self.isQuery = false **and** self.isAbstract = false **and** self.isLeaf = false **and** self.isRoot = false **and** self.mandatory = true **and** ((self **implies** (union()) **and** (union() **implies** self)) **and** sequence_diagram::service_provisioning_system→includes(self)
Description	The *constraint* specifies the meta-attributes, and context of the stereotyped operation.

Table 5.43: Operation *Translate Order* stereotype

Stereotype	translate_order
Base Class	Operation
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** translate_order **inv:** self.ownerScope = instance **and** self.visibility = private **and** self.isQuery = false **and** self.isAbstract = false **and** self.isLeaf = false **and** self.isRoot = false **and** self.mandatory = true **and** ((self **implies** (collect()) **and** (collect() **implies** self)) **and** sequence_diagram::service_provisioning_system→includes(self)
Description	The *constraint* specifies the meta-attributes, and context of the stereotyped operation.

Table 5.44: Operation *Create Service Description* stereotype

Stereotype	create_service_description
Base Class	Operation
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** create_service_descriptionr **inv:** self.ownerScope = instance **and** self.visibility = private **and** self.isQuery = false **and** self.isAbstract = false **and** self.isLeaf = false **and** self.isRoot = false **and** self.mandatory = true **and** ((self **implies** (create()) **and** (create() **implies** self)) **and** sequence_diagram::service_provisioning_system→includes(self)
Description	The *constraint* specifies the meta-attributes, and context of the stereotyped operation.

Table 5.45: Operation *Store Order* stereotype

Stereotype	store_order
Base Class	Operation
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** store_order **inv:** self.ownerScope = instance **and** self.visibility = private **and** self.isQuery = false **and** self.isAbstract = false **and** self.isLeaf = false **and** self.isRoot = false **and** self.mandatory = true **and** ((self **implies** (collect()) **and** (collect() **implies** self)) **and** sequence_diagram::service_provisioning_system→includes(self)
Description	The *constraint* specifies the meta-attributes, and context of the stereotyped operation.

Table 5.46: Operation *Store Description* stereotype

Stereotype	store_description
Base Class	Operation
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** store_description **inv:** self.ownerScope = instance **and** self.visibility = private **and** self.isQuery = false **and** self.isAbstract = false **and** self.isLeaf = false **and** self.isRoot = false **and** self.mandatory = true **and** ((self **implies** (collect()) **and** (collect() **implies** self)) **and** sequence_diagram::service_provisioning_system→includes(self)
Description	The *constraint* specifies the meta-attributes, and context of the stereotyped operation.

Table 5.47: Operation *Schedule Service Desc* stereotype

Stereotype	schedule_service_desc
Base Class	Operation
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** schedule_scervice_desc **inv:** self.ownerScope = instance **and** self.visibility = private **and** self.isQuery = false **and** self.isAbstract = false **and** self.isLeaf = false **and** self.isRoot = false **and** self.mandatory = true **and** ((self **implies** (**asSequence**()) **and** (asSequence() **implies** self)) **and** sequence_diagram::service_provisioning_system→includes(self)
Description	The *constraint* specifies the meta-attributes, and context of the stereotyped operation.

Table 5.48: Operation *Create Atomic Service* stereotype

Stereotype	create_atomic_service
Base Class	Operation
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** create_atomic_service **inv:** self.ownerScope = instance **and**self.visibility = private **and** self.isQuery = false **and** self.isAbstract = false **and** self.isLeaf = false **and** self.isRoot = false **and** self.mandatory = true **and** ((self **implies** (create()) **and** (create() **implies** self)) **and** sequence_diagram::service_provisioning_system→includes(self)
Description	The *constraint* specifies the meta-attributes, and context of the stereotyped operation.

Table 5.49: Operation *Determine Route* stereotype

Stereotype	determine_route
Base Class	Operation
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** determine_route **inv:** self.ownerScope = instance **and** self.visibility = private **and** self.isQuery = false **and** self.isAbstract = false **and** self.isLeaf = false **and** self.isRoot = false **and** self.mandatory = true **and** ((self **implies** (isQuery()) **and** (isQuery() **implies** self)) **and** sequence_diagram::service_provisioning_system→includes(self)
Description	The *constraint* specifies the meta-attributes, and context of the stereotyped operation.

Table 5.50: Operation *Send Atomic Service* stereotype

Stereotype	send_atomic_service
Base Class	Operation
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** send_atomic_service **inv:** self.ownerScope = instance **and** self.visibility = private **and** self.isQuery = false **and** self.isAbstract = false **and** self.isLeaf = false **and** self.isRoot = false **and** self.mandatory = true **and** ((self **implies** (collect()) **and** (collect() **implies** self)) **and** sequence_diagram::service_provisioning_system→includes(self)
Description	The *constraint* specifies the meta-attributes, and context of the stereotyped operation.

Table 5.51: Operation *Execute Atomic Service* stereotype

Stereotype	execute_atomic_service
Base Class	Operation
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** execute_atomic_service **inv:** self.ownerScope = instance **and** self.visibility = private **and** self.isQuery = false **and** self.isAbstract = false **and** self.isLeaf = false **and** self.isRoot = false **and** self.mandatory = true **and** ((self **implies** (iterate()) **and** (iterate() **implies** self)) **and** sequence_diagram::service_provisioning_system→includes(self)
Description	The *constraint* specifies the meta-attributes, and context of the stereotyped operation.

Chapter 6

Case Study (Code Generator System)

6.1 Introduction

In this chapter a DSML for a program synthesizer that is used within the satellite navigation system domain will be developed, using actual data from a one such organization. The data presented herein will not be sanitized, as was done with the data used in Chapter 5. This is because of the presence of publications of this work that are available in the public domain. Hence there is no need for the protection of any proprietary information.

The process outlined in Chapter 4 will be applied to the data obtained to produce a set of models as presented in Chapter 3. Some of these models have no applicability in this domain, and thus will not be developed.

The data that will be used in this chapter was obtained during a ten week internship at the *National Aeronautics and Space Administration* (NASA). The internship took place at the NASA Ames Research Center in California, USA under the NASA/Research Institute for Advanced Computer Science (RIACS) 2002 Summer Student Research Program. This work was supervised by Dr. Jonathan Whittle Ph.D. of QSS Group, Inc. at **NASA Ames**.

The internship work involved the creation of a set of UML models along with OCL constraint statements of an existing systems for which documentation of the system exist in the form of legacy code, and publications from conference proceedings [80]. The UML models and OCL constraint statements are intended to be used in validating the output from the system against the system input.

The purpose of the DA exercise conducted departed from the traditional purpose of DA in DSSEE, as illustrated in Figure 1.2, and presented in Section 2.4.4. The traditional purpose of DA activity in general and in DSSEE is for "*... identifying, collecting, organization, analyzing, and representing ... knowledge ... in support of the description and solutions* [of] *problems*". Whereas for the **NASA Ames** project the purpose was not for the derivation of a solution (solutions exist in the form of an existing system) but to develop static domain models of the system's input and output, that are used to specify a formal set of domain constraints.

6.2 Description of the Domain

The **NASA Ames** work involved the derivation of sets of constraints for the inputs and outputs of a system (*AutoFilter*) that synthesizes (generate) code that implements a Kalman Filter. Kalman Filters are specifications that are used to estimate the dynamic state of a system such as is need in the filed of autonomous or assisted navigation. An understanding of Kalman Filters is crucial to this case study, and as such a description is taken verbatim from the web-site published work of Greg Welch[1] and Gary Bishop[2], titled **An Introduction to the Kalman Filter**[3] in the following Section 6.2.1.

6.2.1 Description of Kalman Filter by Welch and Bishop

"In 1960, R.E. Kalman published his famous paper describing a recursive solution to the discrete-data linear filtering problem. Since that time, due in large part to advances in digital computing, the Kalman filter has been the subject of extensive research and application, particularly in the area of autonomous or assisted navigation. The Kalman filter is a set of mathematical equations that provides an efficient computational (recursive) solution of the

---

[1] *http://www.cs.unc.edu/~welch*

[2] *http://www.cs.unc.edu/~gb*

[3] *http://www.cs.unc.edu/~welch/kalman/kalmanfilter/kalman.html*

least-squares method. The filter is very powerful in several aspects: it supports estimations of past, present, and even future states, and it can do so even when the precise nature of the modeled system is unknown."

6.2.1.1 The Discrete Kalman Filter

The Process to be Estimated

The Kalman filter addresses the general problem of trying to estimate the state of a discrete-time controlled process that is governed by the linear stochastic difference equation:

$$x_k = Ax_{k-1} + Bu_k + w_{k-1} \tag{1.1}$$

with a measurement $z \in \Re^m$ that is

$$z_k = Hx_k + v_k \tag{1.2}$$

"The random variables w_{k-1} andv_k represent the process and measurement noise (respectively). They are assumed to be independent (of each other), white, and with normal probability distributions:

$$p(w) \sim N(0, Q) \tag{1.3}$$

$$p(v) \sim N(0, Q) \tag{1.4}$$

"In practice, the process noise covariance Q and measurement noise covariance R matrices might change with each time step or measurement, however here we assume they are constant. The `n x n` *matrix A in the difference equation* (1.1) *relates the state at the previous time step* `k - 1` *to the state at the current step i_k, in the absence of either a driving function or process noise. Note that in practice A might change with each time step, but here we assume it is constant. The matrix B relates the optional control input $u \in \Re^l$ to the state x. The* `m x n` *matrix H in the measurement equation* (1.2) *relates the state to the measurement z_k. In practice H might change with each time step or measurement, but here we assume it is constant."*

The Discrete Kalman Filter Algorithm

"The Kalman filter estimates a process by using a form of feedback control: the filter estimates the process state at some time and then obtains feedback in the form of(noisy)

measurements. As such, the equations for the Kalman filter fall into two groups: time update equations and measurement update equations. The time update equations are responsible for projecting forward (in time) the current state and error covariance estimates to obtain the a priori estimates for the next time step. The measurement update equations are responsible for the feedback–i.e. for incorporating a new measurement into the a priori estimate to obtain an improved a posteriori estimate. The time update equations can also be thought of as predictor equations, while the measurement update equations can be thought of as corrector equations. Indeed the final estimation algorithm resembles that of a predictor-corrector algorithm for solving numerical problems.

"The first task during the measurement update is to compute the Kalman gain, K_k. The next step is to actually measure the process to obtain z_k, and then to generate an a posteriori state estimate by incorporating the measurement. The final step is to obtain an a posteriori error covariance estimate.

"After each time and measurement update pair, the process is repeated with the previous a posteriori estimates used to project or predict the new a priori estimates. This recursive nature is one of the very appealing features of the Kalman filter - it makes practical implementations much more feasible."

Tables 6.1 ... *offers a complete picture of the operation of the filter".*

Table 6.1: **NASA** Kalman Filter operation

Time Update ("Predict") - A		**Measurement Update ("Correct") - B**
(1) Project the state ahead $\hat{x}^- = A\hat{x}_{k-1} + Bu_k$	$\Longrightarrow$	(1) Compute the Kalman Gain $K_k = P_k^- H_T (H P_k^- H_T + R)^{-1}$
(2) Project the error covariance ahead $P_k^- = AP_{k-1}A^T + Q$	$\Longleftarrow$	(2) Update estimate with measurement z_k $\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-)$ (3) Update the error covariance $P_k = (I - K_k H)P_k^-$
⇑ Initial estimates for $\hat{x}_{k-1}$ and P_{k-1}		

The **NASA Ames** system that was the subject of the internship work (*AutoFilter*) is a code generator that synthesis a fully executable program (a Kalman Filter). *AutoFilter* works by taking as input a mathematical specification of a set of stochastic equations, along with descriptions of the noise characteristics and filter parameters. *AutoFilter* then

generates code that implements a variant of the equations specified in the Tables 6.1**A** and **B**. Table 6.1 equations are but one representation of many possible configuration of filters that may be synthesized. Other variations include, *extended*, *information*, and *unscented*.

In conducting the work *AutoFilter* was treated as a black box, and the work focused on developing static models of the input and output of *AutoFilter*. The goal of this project was centered on being able to determine whether the structure of a filter program that was an output of *AutoFilter* conformed to to a specified framework. This issue arose out of work done on modifying *AutoFilter* to provide additional functionalities. It became a concern to the developers as to whether the previous synthesis process was adversely affected by the modifications, i.e. *Was AutoFilter still generating synthesized filters in the same manner after the modifications as it was before the modifications, and was the the program a representation of a valid Kalman Filter?*.

The **NASA Ames** researchers working with *AutoFilter* determined that one possible way to answer that question is to use a precise (formal) description of *AutoFilter*'s input and output as a metric for determining the validity of the output against the input. Thus the creation of this formal description and validation process was to be the deliverables of the internship work.

6.3 Specifying the Domain Models

The first two weeks of the internship was dedicated to obtaining an understanding of the domain of Kalman Filters. This domain knowledge was obtained from discussions with the domain experts and the reading of some published literature on Kalman Filters. This knowledge would be used in identifying the 'structure' of the input equations and output programs of *AutoFilter*. Initially it was decide that a set of static and dynamic models would be created, but the use of dynamic models was discounted as *AutoFilter* was being treated as a black-box *consumer/producer* that consumed *equation* specifications and produced *Kalman Filter* code. The next three of weeks were spent developing static models of *AutoFilter*'s input and output. These static models were in the form of UML class diagrams, a number of versions were generated before a satisfactory description was obtained.

The first three weeks of the second half of the internships was dedicated to developing the OCL constraints of the input and output models - With the final two weeks being spent on using an OCL validation too to verify the constraints that were developed.

The activities for defining the static domain models in DSSEE as presented in Section 2.4 was conducted in this project, and the static models are presented below in Sections 6.3.1. It is to be, again, noted that the definition of a DSML is independent of any domain analysis methodology that is used to define the domain models for DSML. Only design level CD domain models were developed in this **NASA Ames** project.

6.3.1 The design class diagram model

Figure 6.1 captures the static structure of the input equations for **NASA Ames** *AutoFilter*. The input equations consist of a *Process Model* and a *Measurement Model* equations. A *Process Model* may be associated with multiple *Measurement Model*, and a *Measurement Model* may be associated with multiple *Process Model*. A *Process Model* may be specialized into a *Linear Discrete Process* model, *Nonlinear Discrete Process* model, *Linear Continuous Process* model, or a *Linear Continuous Process* model. These four specialized types of models constitute the main classification types of Kalman Filters for the domain.

A *Process Model* is made up of (1) a *State Vector* that is composed of a set of *State Variables*, (2) a *Process Noise* that is composed of a set of *Process variables*, and (3) an optionally *Control Vector* that is composed of a set of *Control Variables*. The *State Variables* and *Process Variabless* are random variables [55], and the *Process Variable* has a *mean* of zero and white noise.

The *Measurement Model* may be either a *Linear Measurement Model* or a *Nonlinear Measurement Model*, and is made up of a *State Vector* and a *Measurement Noise*. The *State Vector* of the *Measurement Model* is the same *State Vector* of the *Process Model*. The *Measurement Noise* is composed of a set of *Measurement Variables* has a *mean* of zero and white noise.

The output of *AutoFilter*, a *Kalman Filter* program, is illustrated in Figure 6.2. A *Kalman Filter* is associated with one *Process Model* and one *Measurement Model*. The

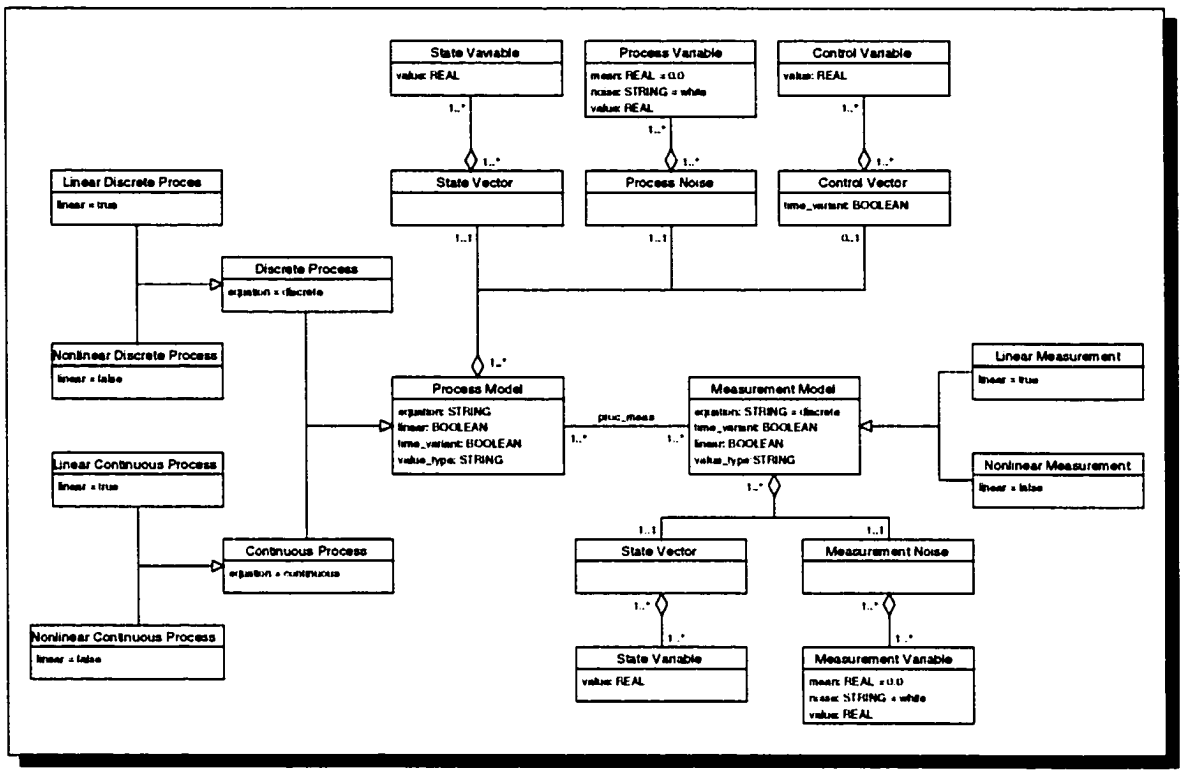

Figure 6.1: Design CD for **NASA Ames** *AutoFilter* system input

structure of synthesized filter program is composed in the following sequence:

- A *Declaration* that is composed of multiple *Matrixs*,
- An *Initialization* that is composed of multiple *Matrixs* and the set of *Declaration Matrixs* is included in the *Initialization Matrix* set,
- A main *Loop* that is composed in the following sequence:
  - A *Measurement Processing* that is composed of a *Measurement* and a *Measurement Tran Hold Matrixs*, and optionally a *Measurement Predict Matrix*,
  - An optional *Update Transition* that is composed of optional *Control Transition*, *Measurement Transition*, and *State Transition*,
  - One of each of a *State Estimate proj eq*, *State Estimate updt eq*, *Gain eq*, *Error Covariance proj eq* and a *Error Covariance updt eq*, where each is composed of

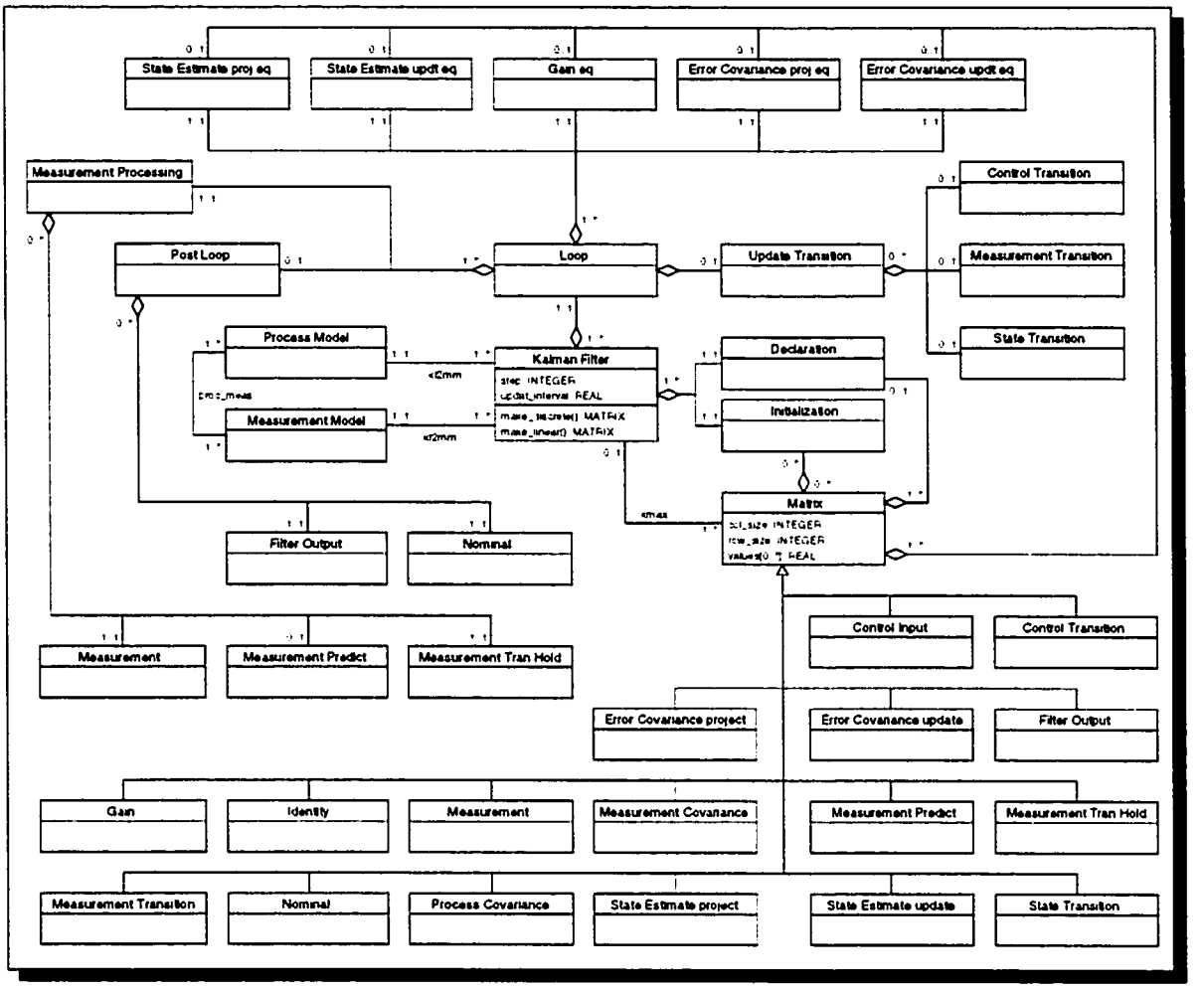

Figure 6.2: Design CD for **NASA Ames** *AutoFilter* system output

multiple *Matrixs*,

– An optional *Post Loop* that is composed of a *Filter Output* and a *Nominal Matrix*.

Because of the project time limitation it was not possible to develop a domain dictionary during the period of the internship. Albeit, this was done during the documentation of the experience for this case study report. From the domain design CDs of Figure 6.1 and 6.2 the static concept dictionary (SCD) is presented in Table 6.2 (static input concept) and 6.3 (static output concept). The separation of the static concepts into two tables was done for presentation purposes only (the complete table is too large to be conveniently presented on

one page).

Table 6.2: **AutoFilter** domain dictionary (static input concepts)

Term	*Property*	*Example*
process model	concrete, mandatory *specialization* discrete process, continuous process *association* proc_meas, kf2pm *attribute* equation: STRING, linear: BOOLEAN, time_variant: BOOLEAN, value_type: STRING	discrete process
measurement model	concrete, mandatory *specialization* linear measurement model, nonlinear measurement model *association* proc_meas, kf2mm *attribute* equation: STRING, linear: BOOLEAN, time_variant: BOOLEAN, value_type: STRING	linear measurement model
discrete process	concrete, optional *generalization* process model *specialization* linear discrete process, nonlinear discrete process	linear discrete process
continuous process	concrete, optional *generalization* process model *specialization* linear continuous process, nonlinear continuous process	linear continuous process
linear discrete process	concrete, optional *generalization* discrete process	linear discrete process
linear continuous process	concrete, optional *generalization* continuous process	linear continuous process
linear continuous process	concrete, optional *generalization* continuous process	linear continuous process
nonlinear continuous process	concrete, optional *generalization* continuous process	nonlinear continuous process
linear measurement model	concrete, optional *generalization* measurement model	linear measurement model
nonlinear measurement model	concrete, optional *generalization* measurement model	nonlinear measurement model
state vector	concrete, mandatory *aggregation* pm2sv, sv2svr, mm2sv	state vector
process noise	concrete, mandatory *aggregation* pm2pn, pn2pv	process noise
control vector	concrete, optional *aggregation* pm2cv, cv2cvr	control vector
state variable	concrete, mandatory *aggregation* sv2svr	state variable
process variable	concrete, mandatory *aggregation* pn2pv	process variable
control variable	concrete, optional *aggregation* cv2cvr	control variable
measurement noise	concrete, mandatory *aggregation* mm2mn, mn2mv	measurement noise
measurement variable	concrete, mandatory *aggregation* mn2mv	measurement variable

The application *AutoFilter* was not not developed with object-oriented (OO) principles, nor was there any intent to apply OO principles in defining the domain models and constraints.

Table 6.3: **AutoFilter** domain dictionary (static output concepts)

Term	*Property*	*Example*
process model	concrete, mandatory *specialization* discrete process, continuous process *association* proc_meas, kf2pm *attribute* equation: STRING, linear: BOOLEAN, time_variant: BOOLEAN, value_type: STRING	discrete process
measurement model	concrete, mandatory *specialization* linear measurement model, nonlinear measurement model *association* proc_meas, kf2mm *attribute* equation: STRING, linear: BOOLEAN, time_variant: BOOLEAN, value_type: STRING	linear measurement model
discrete process	concrete, optional *generalization* process model *specialization* linear discrete process, nonlinear discrete process	linear discrete process
continuous process	concrete, optional *generalization* process model *specialization* linear continuous process, nonlinear continuous process	linear continuous process
linear discrete process	concrete, optional *generalization* discrete process	linear discrete process
linear continuous process	concrete, optional *generalization* continuous process	linear continuous process
linear continuous process	concrete, optional *generalization* continuous process	linear continuous process
nonlinear continuous process	concrete, optional *generalization* continuous process	nonlinear continuous process
linear measurement model	concrete, optional *generalization* measurement model	linear measurement model
nonlinear measurement model	concrete, optional *generalization* measurement model	nonlinear measurement model
state vector	concrete, mandatory *aggregation* pm2sv, sv2svr, mm2sv	state vector
process noise	concrete, mandatory *aggregation* pm2pn, pn2pv	process noise
control vector	concrete, optional *aggregation* pm2cv, cv2cvr	control vector
state variable	concrete, mandatory *aggregation* sv2svr	state variable
process variable	concrete, mandatory *aggregation* pn2pv	process variable
control variable	concrete, optional *aggregation* cv2cvr	control variable
measurement noise	concrete, mandatory *aggregation* mm2mn, mn2mv	measurement noise
measurement variable	concrete, mandatory *aggregation* mn2mv	measurement variable

But with the intent of using UML is was decided that an OO approach would be taken in developing the models. It was thought that this would prove to be advantageous in using UML and OCL.

6.3.2 Creating the SC stereotypes

During development of the domain models, stereotypes were initially defined. This was done because of the varied use of a construct (the *matrix*) in the models. But this approach was revised during work with the OCL validation because of the absence of support for stereotypes in the tool. The usefulness of stereotypes in the domain models was noted and it was agreed that this would be further pursued in developing a DSML for the complete *AutoFilter* system.

Following is an example of a static concept (see Figure 6.1) stereotype for the *Process Model* class of **NASA Ames** (*AutoFilter*) system. Creation of the static concept stereotypes is carried out in an iterative manner with the design level CD and domain dictionary, and the profile base (the UML specification) as inputs. The process for creating the static concept stereotypes is illustrated in the activity diagram of Figure 4.3. The complete list of stereotypes is given in Section 6.4 at the end of this chapter.

Tables of the stereotypes for the attributes of the static concepts (classes) are also included in Section 6.4. The stereotype definition of these model element adds to the formal meaning (semantics) of the owning classes.

6.3.2.1 Packaging the stereotypes

The packaging of the stereotypes involves the definition of a name-spaces for the stereotypes (static concepts). The UML name-space concept ≪profile≫ is used to package the domain stereotypes, as illustrated in Figure 6.3(*Autofilter* Stereotypes).

All the stereotypes of Section 6.4 are contained in the profile of Figure 6.3.

6.3.3 Creating the meta-models

The process for creating the meta-models of the domain models calls for: (1) the definition of domain-specific graphical icons, and (2) the establishment of the relationships between the icons. In this domain (*AutoFilter* program synthesizer) domain-specific graphical icons are not created as such information was not obtained from the domain experts. Consequently,

Table 6.4: Class *Process Model* stereotype

Stereotype	process model
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪process_model≫ **inv:** self.isAbstract = true **and** self.isLeaf = false **and** self.isRoot = true **and** self.mandatory = true **and** self.≪state_vector≫→size() = 1 **and** self.≪process_noise≫→size() = 1 **and** (self.≪control_vector≫→size() = 0 **or** self.≪control_vector≫→size() = 1) **and** self.≪measurement_model≫→size() = 1 **and** self.≪equation≫.multiplicity = 1 **and** self.≪equation≫.changeability = frozen **and** self.≪equation≫.visibility = public **and** self.≪linear≫.multiplicity = 1 **and** self.≪linear≫.changeability = frozen **and** self.≪linear≫.visibility = public **and** self.≪time_variant≫.multiplicity = 1 **and** self.≪time_variant≫.changeability = frozen **and** self.≪time_variant≫.visibility = public **and** self.≪value_type≫.multiplicity = 1 **and** self.≪value_type≫.changeability = frozen **and** self.≪value_type≫.visibility = public **and** (self.≪equation≫→forAll(eq | eq.initialValue.body = 'discrete') **xor** self.≪equation≫→forAll(eq | eq.initialValue.body = 'continuous')) **and** (self.≪linear≫→forAll(lr | lr.initialValue.body = 'true') **xor** self.≪linear≫→forAll(lr | lr.initialValue.body = 'false')) **and** (self.≪time_variant≫→forAll(tv | tv.initialValue.body = 'true') **xor** self.≪time_variant≫→forAll(tv | tv'initialValue.body = 'false')) **and** (self.≪value_type≫→forAll(vt | vt.initialValue.body = 'absolute') **xor** self.≪value_type≫→forAll(vt | vt.initialValue.body = 'incremental')) **and** (self.≪control_vector≫→notEmpty() **implies** (self.≪control_vector.≫≪time_variant.≫→forAll(tv | tv.initialValue.body = 'true') **xor** self.≪control_vector.≫≪time_variant≫→forAll(tv | tv.initialValue.body = 'false'))
Description	The *constraint* specifies the meta-attributes, multiplicities and specializations of the stereotype.

the default UML icons are used. The associations are then established between the default graphical icons that represent the stereotypes of Table 6.6 through 6.37. The meta-model of the design CD input is illustrated in Figure 6.3.3. The complete *AutoFilter* meta-models are presented in Section 6.4.3 at the end of this chapter.

6.3.4 Packaging the meta-models

The packaging of the meta-models involves the definition of a name-spaces for the two meta-models (input and output design CDs). The UML name-space concept ≪*profile*≫ is

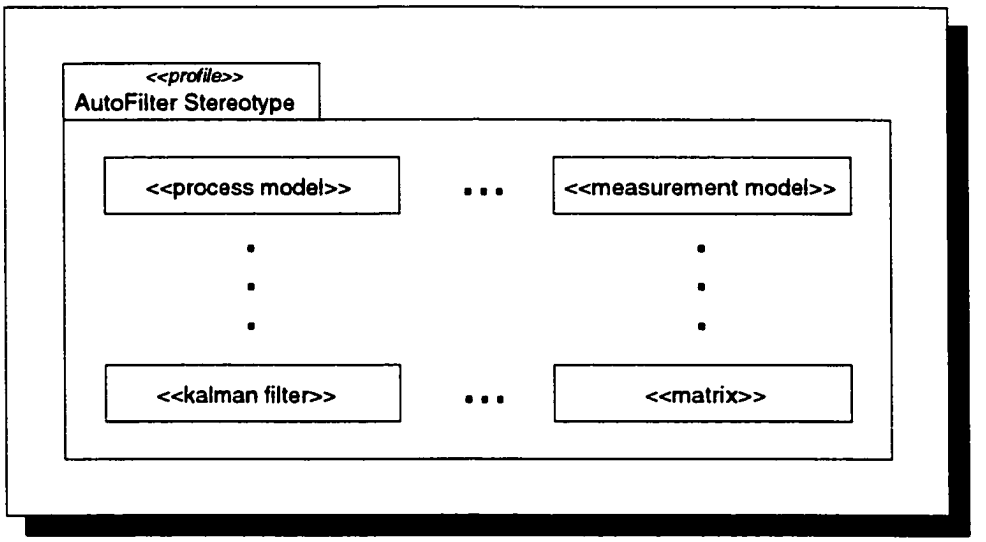

Figure 6.3: Domain stereotypes **NASA Ames** *AutoFilter* system

used to package the domain meta-models, as illustrated in Figure 6.4. The meta-models are individually packaged in profiles and these profiles are then packaged as a profile of profiles (*AutoFilter Domain Meta-Models*).

6.3.5 Packaging the DSML

The packaging of the DSML involves the definition of a name-spaces for *AutoFilters* stereotype profile (*AutoFilter Stereotype*, and the meta-model profile (*AutoFilter Meta-Model*). The UML name-space concept ≪*profile*≫ is used to package the domain stereotypes, and meta-models, as illustrated in Figures 6.3, and 6.4. These individually packaged profiles are then packaged as a profile of profiles the (*AutoFilter DSML*) as illustrated in Figure 6.5.

This concludes the creation of the domain-specific modeling language (DSML) for **NASA Ames**'s *AutoFilter* code synthesizer (generator) system.

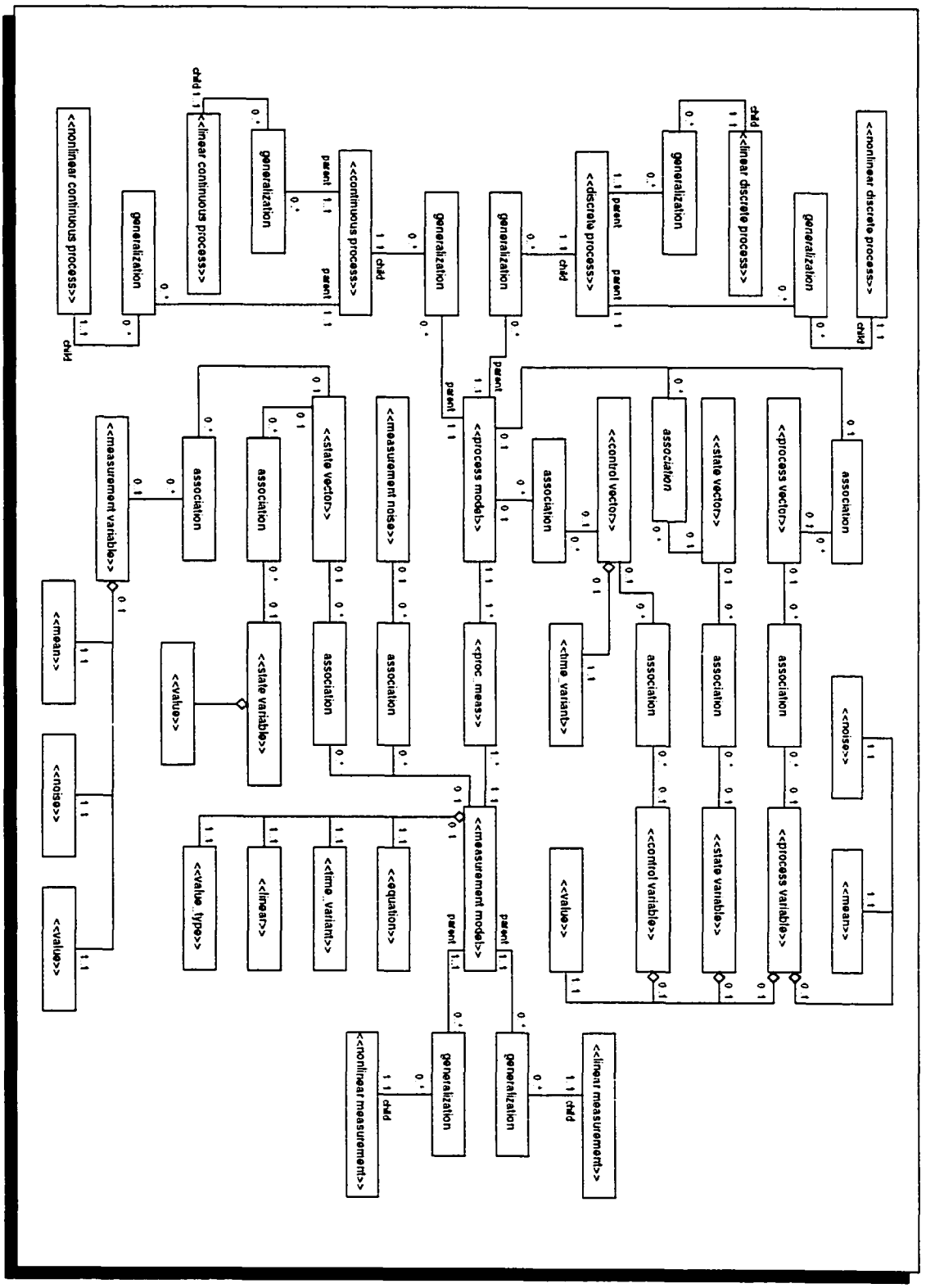

Figure6.3.3 AutoFilter DSML design CD input meta-model

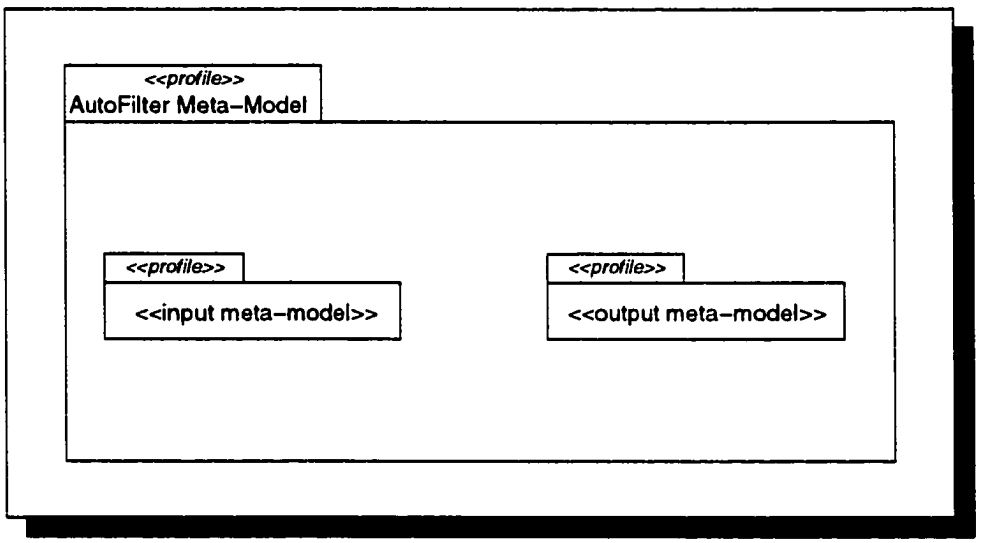

Figure 6.4: AutoFilter DSML meta-model profile packaging

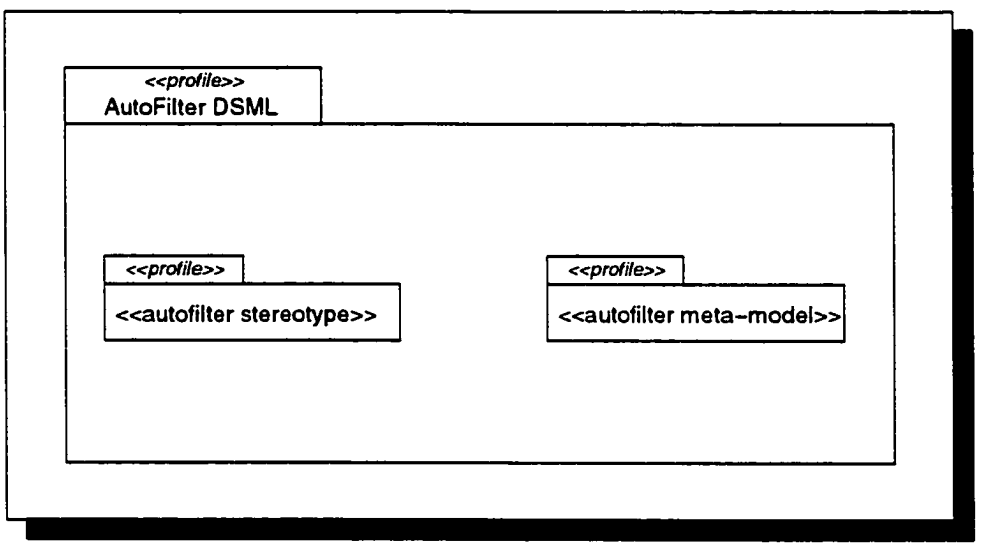

Figure 6.5: AutoFilter DSML profile packaging

6.4 AutoFilter DSML Specification

The following sub-sections contain the complete specification for the **NASA Ames** code (program) synthesizer (generator) system (*AutoFilter*) input.

6.4.1 AutoFilter input stereotypes

Table 6.5: Tag definition *Mandatory*

Tag	mandatory
Stereotype	item
Type	UML::Datatype::Boolean
Multiplicity	1
Description	Indicates whether a stereotyped class with this tag must be included in a domain application model.

Table 6.6: Class *Process Model* stereotype

Stereotype	process model
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪process_model≫ **inv:** self.isAbstract = true **and** self.isLeaf = false **and** self.isRoot = true **and** self.mandatory = true **and** self.≪state_vector≫→size() = 1 **and** self.≪process_noise≫→size() = 1 **and** (self.≪control_vector≫→size() = 0 **or** self.≪control_vector≫→size() = 1) **and** self.≪measurement_model≫→size() = 1 **and** self.≪equation≫.multiplicity = 1 **and** self.≪equation≫.changeability = frozen **and** self.≪equation≫.visibility = public **and** self.≪linear≫.multiplicity = 1 **and** self.≪linear≫.changeability = frozen **and** self.≪linear≫.visibility = public **and** self.≪time_variant≫.multiplicity = 1 **and** self.≪time_variant≫.changeability = frozen **and** self.≪time_variant≫.visibility = public **and** self.≪value_type≫.multiplicity = 1 **and** self.≪value_type≫.changeability = frozen **and** self.≪value_type≫.visibility = public **and** (self.≪equation≫→forAll(eq | eq.initialValue.body = 'discrete') **xor** self.≪equation≫→forAll(eq | eq.initialValue.body = 'continuous')) **and** (self.≪linear≫→forAll(lr | lr.initialValue.body = 'true') **xor** self.≪linear≫→forAll(lr | lr.initialValue.body = 'false')) **and** (self.≪time_variant≫→forAll(tv | tv.initialValue.body = 'true') **xor** self.≪time_variant≫→forAll(tv | tv'initialValue.body = 'false')) **and** (self.≪value_type≫→forAll(vt | vt.initialValue.body = 'absolute') **xor** self.≪value_type≫→forAll(vt | vt.initialValue.body = 'incremental')) **and** (self.≪control_vector≫→notEmpty() **implies** (self.≪control_vector.≫≪time_variant.≫→forAll(tv | tv.initialValue.body = 'true') **xor** self.≪control_vector.≫≪time_variant≫→forAll(tv | tv.initialValue.body = 'false'))
Description	The *constraint* specifies the meta-attributes, multiplicities and specializations of the stereotype.

Table 6.7: Class *Discrete Process* stereotype

Stereotype	discrete process
Base Class	Class
Parent	Process Model
Tag	mandatory
Representation	default
Constraint	**context** ≪discrete_process≫ **inv:** self.mandatory = false **and** self.≪equation≫→forAll(eq | eq.initialValue.body = 'discrete')
Description	The *constraint* specifies the specialization of a parent stereotype.

Table 6.8: Class *Continuous Process* stereotype

Stereotype	continuous process
Base Class	Class
Parent	Process Model
Tag	mandatory
Representation	default
Constraint	**context** ≪continuous_process≫ **inv:** self.mandatory = false **and** self.≪equation≫→forAll(eq | eq.initialValue.body = 'continuous')
Description	The *constraint* specifies the specialization of a parent stereotype.

Table 6.9: Class *Linear Discrete Process* stereotype

Stereotype	linear discrete process
Base Class	Class
Parent	Discrete Process
Tag	mandatory
Representation	default
Constraint	**context** ≪linear_discrete_process≫ **inv:** self.mandatory = false **and** self.≪linear≫→forAll(lr | lr.initialValue.body = 'true')
Description	The *constraint* specifies the specialization of a parent stereotype.

Table 6.10: Class *Nonlinear Discrete Process* stereotype

Stereotype	nonlinear discrete process
Base Class	Class
Parent	Discrete Process
Tag	mandatory
Representation	default
Constraint	**context** ≪nonlinear_discrete_process≫ **inv:** self.mandatory = false **and** self.≪linear≫→forAll(lr | lr.initialValue.body = 'false')
Description	The *constraint* specifies the specialization of a parent stereotype.

Table 6.11: Class *Linear Continuous Process* stereotype

Stereotype	linear continuous process
Base Class	Class
Parent	Continuous Process
Tag	mandatory
Representation	default
Constraint	**context** ≪linear_continuous_process≫ **inv:** self.mandatory = false **and** self.≪linear≫→forAll(lr | lr.initialValue.body = 'true')
Description	The *constraint* specifies the specialization of a parent stereotype.

Table 6.12: Class *Nonlinear Continuous Process* stereotype

Stereotype	nonlinear continuous process
Base Class	Class
Parent	Continuous Process
Tag	mandatory
Representation	default
Constraint	**context** ≪nonlinear_continuous_process≫ **inv:** self.mandatory = false **and** self.≪linear≫→forAll(lr | lr.initialValue.body = 'false')
Description	The *constraint* specifies the specialization of a parent stereotype.

Table 6.13: Class *Measurement Model* stereotype

Stereotype	measurement model
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪measurement_model≫ **inv:** self.isAbstract = true **and** self.isLeaf = false **and** self.isRoot = true **and** self.mandatory = true **and** self.≪state_vector≫→size() = 1 **and** self.≪measurement_noise≫→size() = 1 **and** self.≪measurement_model≫→size() = 1 **and** self.≪equation≫.multiplicity = 1 **and** self.≪equation≫.changeability = frozen **and** self.≪equation≫.visibility = public **and** self.≪linear≫.multiplicity = 1 **and** self.≪linear≫.changeability = frozen **and** self.≪linear≫.visibility = public **and** self.≪time_variant≫.multiplicity = 1 **and** self.≪time_variant≫.changeability = frozen **and** self.≪time_variant≫.visibility = public **and** self.≪value_type≫.multiplicity = 1 **and** self.≪value_type≫.changeability = frozen **and** self.≪value_type≫.visibility = public **and** self.≪equation≫→forAll(eq | eq.≪initialValue.body = 'discrete') **and** (self.≪linear≫→forAll(lr | lr.initialValue.body = 'true') **xor** self.≪linear≫→forAll(lr | lr.initialValue.body = 'false'))
Description	The *constraint* specifies the meta-attributes, multiplicities and specializations of the stereotype.

Table 6.14: Class *Linear Measurement Model* stereotype

Stereotype	linear measurement model
Base Class	Class
Parent	Measurement Model
Tag	mandatory
Representation	default
Constraint	**context** ≪linear_measurement_model≫ **inv:** self.mandatory = false **and** self.≪linear≫→forAll(lr | lr.initialValue.body = 'true')
Description	The *constraint* specifies the meta-attributes, multiplicities and specializations of the stereotype.

Table 6.15: Class *Nonlinear Measurement Model* stereotype

Stereotype	nonlinear measurement model
Base Class	Class
Parent	Measurement Model
Tag	mandatory
Representation	default
Constraint	**context** ≪nonlinear_measurement_model≫ **inv:** self.mandatory = false **and** self.≪linear≫→forAll(lr | lr.initialValue.body = 'false')
Description	The *constraint* specifies the meta-attributes, multiplicities and specializations of the stereotype.

Table 6.16: Class *State Vector* stereotype

Stereotype	state vector
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪state_vector≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪process_model≫→size() = 1 **and** self.≪measurement_model≫→size() = 1 **and** self.≪state_variable≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and specializations of the stereotype.

Table 6.17: Class *Process Noise* stereotype

Stereotype	process noise
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪process_noise≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪process_model≫→size() = 1 **and** self.≪process_variable≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and specializations of the stereotype.

Table 6.18: Class *Measurement Noise* stereotype

Stereotype	measurement noise
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪measurement_noise≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪measurement_model≫→size() = 1 **and** self.≪measurement_variable≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and specializations of the stereotype.

Table 6.19: Class *Control Vector* stereotype

Stereotype	control vector
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪control_vector≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = false **and** self.≪time_variant≫.multiplicity = 1 **and** self.≪time_variant≫.changeability = frozen **and** self.≪time_variant≫.visibility = public **and** (self.≪process_model≫→size() = 0 **xor** self.≪process_model≫→size() = 1) **and** self.≪control_variable≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and specializations of the stereotype.

Table 6.20: Class *State Variable* stereotype

Stereotype	state variable
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪state_variable≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪time_variant≫.multiplicity = 1 **and** self.≪time_variant≫.changeability = frozen **and** self.≪time_variant≫.visibility = public **and** self.≪state_vector≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and specializations of the stereotype.

Table 6.21: Class *Process Variable* stereotype

Stereotype	process variable
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪process_variable≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪mean≫.multiplicity = 1 **and** self.≪mean≫.changeability = frozen **and** self.≪mean≫.visibility = public **and** self.≪mean≫→forAll(mn | mn.mn.initialValue.body = 0.0) **and** self.≪noise≫.multiplicity = 1 **and** self.≪noise≫.changeability = frozen **and** self.≪noise≫.visibility = public **and** self.≪noise≫→forAll(ns | ns.mn.initialValue.body = 'white') **and** self.≪value≫.multiplicity ≥ 1 **and** self.≪value≫.changeability = frozen **and** self.≪value≫.visibility = public **and** self.≪process_noise≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and specializations of the stereotype.

Table 6.22: Class *Measurement Variable* stereotype

Stereotype	measurement variable
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪measurement_variable≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪mean≫.multiplicity = 1 **and** self.≪mean≫.changeability = frozen **and** self.≪mean≫.visibility = public **and** self.≪mean≫→forAll(mn | mn.mn.initialValue.body = 0.0) **and** self.≪noise≫.multiplicity = 1 **and** self.≪noise≫.changeability = frozen **and** self.≪noise≫.visibility = public **and** self.≪noise≫→forAll(ns | ns.mn.initialValue.body = 'white') **and** self.≪value≫.multiplicity ≥ 1 **and** self.≪value≫.changeability = frozen **and** self.≪value≫.visibility = public **and** self.≪measurement_noise≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and specializations of the stereotype.

Table 6.23: Class *Control Variable* stereotype

Stereotype	control variable
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪control_variable≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = false **and** self.≪value≫.multiplicity ≥ 1 **and** self.≪value≫.changeability = frozen **and** self.≪value≫.visibility = public **and** self.≪control_vector≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and specializations of the stereotype.

6.4.2 AutoFilter output stereotypes

The following sub-sections contain the complete specification for the **NASA Ames** code (program) synthesizer (generator) system (*AutoFilter*) output.

Table 6.24: Class *Kalman Filter* stereotype

Stereotype	kalman filter
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪kalman_filter≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪step≫.multiplicity = 1 **and** self.≪step≫.changeability = frozen **and** self.≪step≫.visibility = public **and** self.≪update_interval≫.multiplicity = 1 **and** self.≪update_interval≫.changeability = frozen **and** self.≪update_interval≫.visibility = public **and** self.≪process_model≫→size() = 1 **and** self.≪measurement_model≫→size() = 1 **and** self.≪matrix≫→size() ≥ 1 **and** self.≪declaration≫→size() = 1 **and** self.≪initialization≫→size() = 1 **and** self.≪loop≫→size() = 1 **and** self.≪state estimate project≫→forAll(sep | sep.≪row_size≫.initialValue = self.≪process model≫.≪state vector≫.≪state variable≫→size()) **and** self.≪state estimate project≫→forAll(sep | sep.≪col_size≫initialValue = 1) **and** self.≪state estimate update≫→forAll(seu | seu.≪row_size≫.initialValue = self.≪process model≫.≪state vector≫.≪state variable≫→size()) **and** self.≪state estimate update≫→forAll(seu | seu.≪col_size≫initialValue = 1) **and** self.≪state transition≫→forAll(st | st.≪row_size≫.initialValue = self.≪process model≫.≪state vector≫.≪state variable≫→size()) **and** self.≪state transition≫→forAll(st | st.≪col_size≫initialValue = st.≪row_size≫initialValue) **and** self.≪error covariance project≫→forAll(ecp | ecp.≪row_size≫.initialValue = self.≪process model≫.≪state vector≫.≪state variable≫→size()) **and** self.≪error covariance project≫→forAll(ecp | ecp.≪col_size≫.initialValue = self.≪process model≫.≪state vector≫.≪state variable≫→size()) **and** self.≪error covariance update≫→forAll(ecu | ecu.≪row_size≫.initialValue = self.≪process model≫.≪state vector≫.≪state variable≫→size()) **and** self.≪error covariance update≫→forAll(ecu | ecu.≪col_size≫.initialValue = self.≪process model≫.≪state vector≫.≪state variable≫→size()) **and** self.≪process covariance≫→forAll(pc | pc.≪row_size≫.initialValue = self.≪process model≫.≪state vector≫.≪state variable≫→size()) **and** self.≪process covariance≫→forAll(pc | pc.≪col_size≫.initialValue = self.≪process model≫.≪state vector≫.≪state variable≫→size()) **and**

Table 6.25: Class *Kalman Filter* stereotype cont'd...

Stereotype	kalman filter
	self.≪measurement transition≫→forAll(mt | mt.≪col_size≫.initialValue = self.≪process model≫.≪state vector≫.≪state variable≫→size()) **and** self.≪gain≫→forAll(g | g.≪row_size≫.initialValue = self.≪process model≫.≪state vector≫.≪state variable≫→size()) **and** self.≪gain≫→forAll(g | g.≪col_size≫.initialValue = self.≪process model≫.≪state vector≫.≪state variable≫→size()) **and** self.≪filter output≫→forAll(fo | fo.≪col_size≫.initialValue = self.≪steps≫.initialValue) **and** self.≪measurement covariance≫→forAll(mc | mc.≪row_size≫.initialValue = self.≪measurement model≫.≪measurement noise≫.≪measurement variable≫→size()) **and** self.≪measurement covariance≫→forAll(mc | mc.≪col_size≫.initialValue = self.≪measurement model≫.≪measurement noise≫.≪measurement variable≫→size()) **and** self.≪measurement≫→forAll(m | m.≪row_size≫.initialValue = self.≪measurement model≫.≪measurement noise≫.≪measurement variable≫→size()) **and** self.≪measurement≫→forAll(m | m.≪col_size≫.initialValue = 1) **and** self.≪measurement transition≫→forAll(mt | mt.≪row_size≫.initialValue = self.≪measurement model≫.≪measurement noise≫.≪measurement variable≫→size()) **and** self.≪gain≫→forAll(g | g.≪col_size≫.initialValue = self.≪measurement model≫.≪measurement noise≫.≪measurement variable≫→size()) **and** self.≪process model≫≪.state vector≫ = self.≪measurement model≫≪state vector≫ **and** self.≪process model≫.≪measurement model≫→exists(mm | mm = self.≪measurement model≫ **and** self.≪measurement model≫.≪process model≫→exists(pm | pm = self.≪process model≫ **and** self.≪process model≫.≪control vector≫→notEmpty **implies** self.≪control transition≫→forAll(ct | ct.≪row_size≫.initialValue = self.≪process model≫.≪state vector≫.≪state variable≫→size()) **and** self.≪control transition≫→forAll(ct | ct.≪col_size≫.initialValue = self.≪process model≫.≪state vector≫.≪state variable≫→size()) **and** self.≪control input≫→forAll(ci | ci.≪row_size≫.initialValue = self.≪process model≫.≪state vector≫.≪state variable≫→size()) **and** self.≪control input≫→forAll(ci | ci.≪col_size≫.initialValue = 1 **and** self.operations→includesAll(make_discrete, make_linear)
Description	The *constraint* specifies the meta-attributes, and operations of the stereotype.

Table 6.26: Class *Matrix* stereotype

Stereotype	matrix
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪matrix≫ **inv:** self.isAbstract = true **and** self.isLeaf = false **and** self.isRoot = true **and** self.mandatory = true **and** self.≪col_size≫.multiplicity = 1 **and** self.≪col_size≫.changeability = frozen **and** self.≪col_size≫.visibility = public **and** self.≪row_size≫.multiplicity = 1 **and** self.≪row_size≫.changeability = frozen **and** self.≪row_size≫.visibility = public **and** self.≪values≫.multiplicity ≥ 0 **and** self.≪values≫.changeability = changeable **and** self.≪values≫.visibility = private **and** self.≪kalman_filter≫→size() = 1 **and** self.≪state_estimate_proj_eq≫→size() = 1 **and** self.≪state_estimate_updt_eq≫→size() = 1 **and** self.≪gain≫→size() = 1 **and** self.≪error_covariance_proj_eq≫→size() = 1 **and** self.≪error_covariance_updt_eq≫→size() = 1 **and** self→forAll(mx | mx.values.multiplicity = mx.col_size × mx.row_size)
	self.operations→includesAll(multiply, transpose)
Description	The *constraint* specifies the meta-attributes, and operations of the stereotype.

Table 6.27: Class *State Estimate proj eq* stereotype

Stereotype	state estimate proj eq
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪state_estimate_proj_eq≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪matrix≫→size() ≥ 1 **and** self.≪loop≫→size() = 1 **and** self.≪matrix≫→includesAll(State_Estimate_project, State_Estimate_update)
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.28: Class *State Estimate updt eq* stereotype

Stereotype	state estimate updt eq
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪state_estimate_updt_eq≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪matrix≫→size() ≥ 1 **and** self.≪loop≫→size() = 1 **and** self.≪matrix≫→includesAll(State_Estimate_update, State_Estimate_project, Gain, Measurement_Tran_Hold)
Description	The *constraint* specifies the meta-attributes, and operations of the stereotype.

Table 6.29: Class *Gain eq* stereotype

Stereotype	gain eq
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪gain_eq≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪matrix≫→size() ≥ 1 **and** self.≪loop≫→size() = 1 **and** self.≪matrix≫→includesAll(Gain, Error_Covariance_project, Measurement_Transition Measurement_Covariance)
Description	The *constraint* specifies the meta-attributes, and operations of the stereotype.

Table 6.30: Class *Error Covariance proj eq* stereotype

Stereotype	error covariance proj eq
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪error_covariance_proj_eq≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪matrix≫→size() ≥ 1 **and** self.≪loop≫→size() = 1 **and** self.≪matrix≫→includesAll(Error_Covariance_project, State_Transition, Process_Covariance, Error_Covariance_Update)
Description	The *constraint* specifies the meta-attributes, and operations of the stereotype.

Table 6.31: Class *Error Covariance updt eq* stereotype

Stereotype	error covariance updt eq
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪error_covariance_updt_eq≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪matrix≫→size() ≥ 1 **and** self.≪loop≫→size() = 1 **and** self.≪matrix≫→includesAll(Error_Covariance_update, Process_Covariance)
Description	The *constraint* specifies the meta-attributes, and operations of the stereotype.

Table 6.32: Class *Measurement Processing* stereotype

Stereotype	measurement
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪measurement_processing≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪matrix≫→size() ≥ 1 **and** self.≪loop≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and operations of the stereotype.

Table 6.33: Class *Post Loop* stereotype

Stereotype	post loop
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪post_loop≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪matrix≫→size() ≥ 1 **and** self.≪loop≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and operations of the stereotype.

Table 6.34: Class *Transition Update* stereotype

Stereotype	transition update
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪transition_update≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪matrix≫→size() ≥ 1 **and** self.≪loop≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and operations of the stereotype.

Table 6.35: Class *Loop* stereotype

Stereotype	loop
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪loop≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪kalman_filter≫→size() = 1 **and** self.≪state_estimate_proj_updt≫→size() = 1 **and** self.≪state_estimate_updt_updt≫→size() = 1 **and** self.≪gain≫→size() = 1 **and** self.≪error_covariance_proj_eq≫→size() = 1 **and** self.≪error_covariance_updt_eq≫→size() = 1 **and** (self.≪update_transition≫→size() = 0 **or and** self.≪update_transition≫→size() = 1) **and** (self.≪loop≫→size() = 0 **or** self.≪loop≫→size() = 1)
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.36: Class *Declaration* stereotype

Stereotype	declaration
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪declaration≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪matrix≫→size() ≥ 1 **and** self.≪kalman_filter≫→size() = 1 self.≪matrix≫→includesAll(Gain, State_Estimate_project, State_Estimate_update, Error_Covariance_project, Error_Covariance_update, Measurement_Transition, Measurement_Covariance, Measurement_Tran_Hold, Process_Covariance, State_Transition) (self.≪kalman_Filter≫.≪{process_Model≫→forAll(pm | pm.≪linear≫.initialValue.body = 'false') **or** self.≪kalman_Filter≫.≪measurement_Model≫→forAll(mm | mm.≪linear≫.initialValue.body = 'false')) **and** self.≪kalman_Filter≫.≪process_Model≫→forAll(pm | pm.≪value_type.body = 'absolute') **implies** self.≪matrix≫→exists(mx | mx.name.body = 'Measurement_Predict')
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.37: Class *Initialization* stereotype

Stereotype	declaration
Base Class	Class
Parent	N/A
Tag	mandatory
Representation	default
Constraint	**context** ≪initialization≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = true **and** self.mandatory = true **and** self.≪matrix≫→size() ≥ 1 **and** self.≪kalman_filter≫→size() = 1 **and** self.≪matrix≫→includesAll(self.≪kalman_Filter≫.≪declaration≫.≪matrix≫)
Description	The *constraint* specifies the meta-attributes, and operations of the stereotype.

Table 6.38: Class *State Transition* stereotype

Stereotype	state transition
Base Class	Class
Parent	Matrix
Tag	mandatory
Representation	default
Constraint	**context** ≪state_transition≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = false **and** self.mandatory = true **and** self.≪error_Covariance_proj_eq≫→size() = 1 **and** (self.≪transition_Update≫→size() = 0 **or** self.≪transition_Update≫→size() = 1)
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.39: Class *Control Transition* stereotype

Stereotype	control transition
Base Class	Class
Parent	Matrix
Tag	mandatory
Representation	default
Constraint	**context** ≪control_transition≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = false **and** self.mandatory = false **and** (self.≪transition_Update≫→size() = 0 **or** self.≪transition_Update≫→size() = 1)
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.40: Class *Measurement Transition* stereotype

Stereotype	measurement transition
Base Class	Class
Parent	Matrix
Tag	mandatory
Representation	default
Constraint	**context** ≪measurement_transition≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = false **and** self.mandatory = true **and** self.≪gain_eq≫→size() = 1 **and** (self.≪transition_Update≫→size() = 0 **or** self.≪transition_Update≫→size() = 1)
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.41: Class *Process Covariance* stereotype

Stereotype	process coverance
Base Class	Class
Parent	Matrix
Tag	mandatory
Representation	default
Constraint	**context** ≪process_covariance≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = false **and** self.mandatory = true **and** self.≪error_Covariance_proj≫→size() = 1 **and** self.≪error_Covariance_proj_eq≫→size() = 1 **and** self.≪error_Covariance_updt_eq≫→size() = 1)
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.42: Class *Measurement Covariance* stereotype

Stereotype	measurement covariance
Base Class	Class
Parent	Matrix
Tag	mandatory
Representation	default
Constraint	**context** ≪measurement_covariance≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = false **and** self.mandatory = true **and** self.≪gain_eq≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.43: Class *Measurement* stereotype

Stereotype	measurement
Base Class	Class
Parent	Matrix
Tag	mandatory
Representation	default
Constraint	**context** ≪measurement≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = false **and** self.mandatory = true **and** self.≪measurement_Processing≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.44: Class *Identity* stereotype

Stereotype	identity
Base Class	Class
Parent	Matrix
Tag	mandatory
Representation	default
Constraint	**context** ≪identity≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = false **and** self.mandatory = true
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.45: Class *Nominal* stereotype

Stereotype	nominal
Base Class	Class
Parent	Matrix
Tag	mandatory
Representation	default
Constraint	**context** ≪nominal≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = false **and** self.mandatory = true **and** self.≪post_Loop≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and operations of the stereotype.

Table 6.46: Class *Control Input* stereotype

Stereotype	control input
Base Class	Class
Parent	Matrix
Tag	mandatory
Representation	default
Constraint	**context** ≪control_input≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = false **and** self.mandatory = false
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.47: Class *State Estimate project* stereotype

Stereotype	state estimate project
Base Class	Class
Parent	Matrix
Tag	mandatory
Representation	default
Constraint	**context** ≪state_estimate_project≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = false **and** self.mandatory = true **and** self.≪state_Estimate_proj_eq_eq≫→size() = 1 **and** self.≪state_Estimate_updt_eq≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.48: Class *State Estimate update* stereotype

Stereotype	state estimate update
Base Class	Class
Parent	Matrix
Tag	mandatory
Representation	default
Constraint	**context** ≪state_estimate_update≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = false **and** self.mandatory = true **and** self.≪state_Estimate_proj_eq≫→size() = 1 **and** self.≪state_Estimate_updt_eq≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.49: Class *Gain* stereotype

Stereotype	gain
Base Class	Class
Parent	Matrix
Tag	mandatory
Representation	default
Constraint	**context** ≪gain≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = false **and** self.mandatory = true **and** self.≪state_Estimate_updt_eq≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.50: Class *Error Covariance project* stereotype

Stereotype	error covariance project
Base Class	Class
Parent	Matrix
Tag	mandatory
Representation	default
Constraint	**context** ≪error_covariance_proj≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = false **and** self.mandatory = true **and** self.≪error_Covariance_proj_eq≫→size() = 1 **and** self.≪gain_eq≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.51: Class *Error Covariance update* stereotype

Stereotype	error covariance update
Base Class	Class
Parent	Matrix
Tag	mandatory
Representation	default
Constraint	**context** ≪error_covariance_update≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = false **and** self.mandatory = true **and** self.≪error_Covariance_update_eq≫→size() = 1 **and** (self.≪error_Covariance_updt_eq≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.52: Class *Filter Output* stereotype

Stereotype	filter output
Base Class	Class
Parent	Matrix
Tag	mandatory
Representation	default
Constraint	**context** ≪filter_output≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = false **and** self.mandatory = false **and** self.≪post_Loop≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.53: Class *Measurement Predict* stereotype

Stereotype	measurement predict
Base Class	Class
Parent	Matrix
Tag	mandatory
Representation	default
Constraint	**context** ≪measurement_predict≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** (self.isRoot = false **and** self.mandatory = false **and** self.≪measurement_Processing≫→size() = 0 **or** (self.≪measurement_Processing≫→size() = 0
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

Table 6.54: Class *Measurement Tran Hold* stereotype

Stereotype	measurement tran hold
Base Class	Class
Parent	Matrix
Tag	mandatory
Representation	default
Constraint	**context** ≪measurement_tran_hold≫ **inv:** self.isAbstract = false **and** self.isLeaf = true **and** self.isRoot = false **and** self.mandatory = false **and** self.≪measurement_Processing≫→size() = 1
Description	The *constraint* specifies the meta-attributes, and associations of the stereotype.

6.4.3 AutoFilter meta-models

The meta-models for **NASA Ames** *AutoFilter* Kalman Filter program synthesizer (generator) the design CD input, and design CD output are presented in Figure 6.4.3**A** and 6.4.3**B** respectively.

Figure 6.6**C** is a continuation of Figure 6.4.3**B**. The separation of the output meta-model into two diagrams was done for the purpose of presentation (fitting the entire model into one diagram would make it difficult to read, because of the scaling).

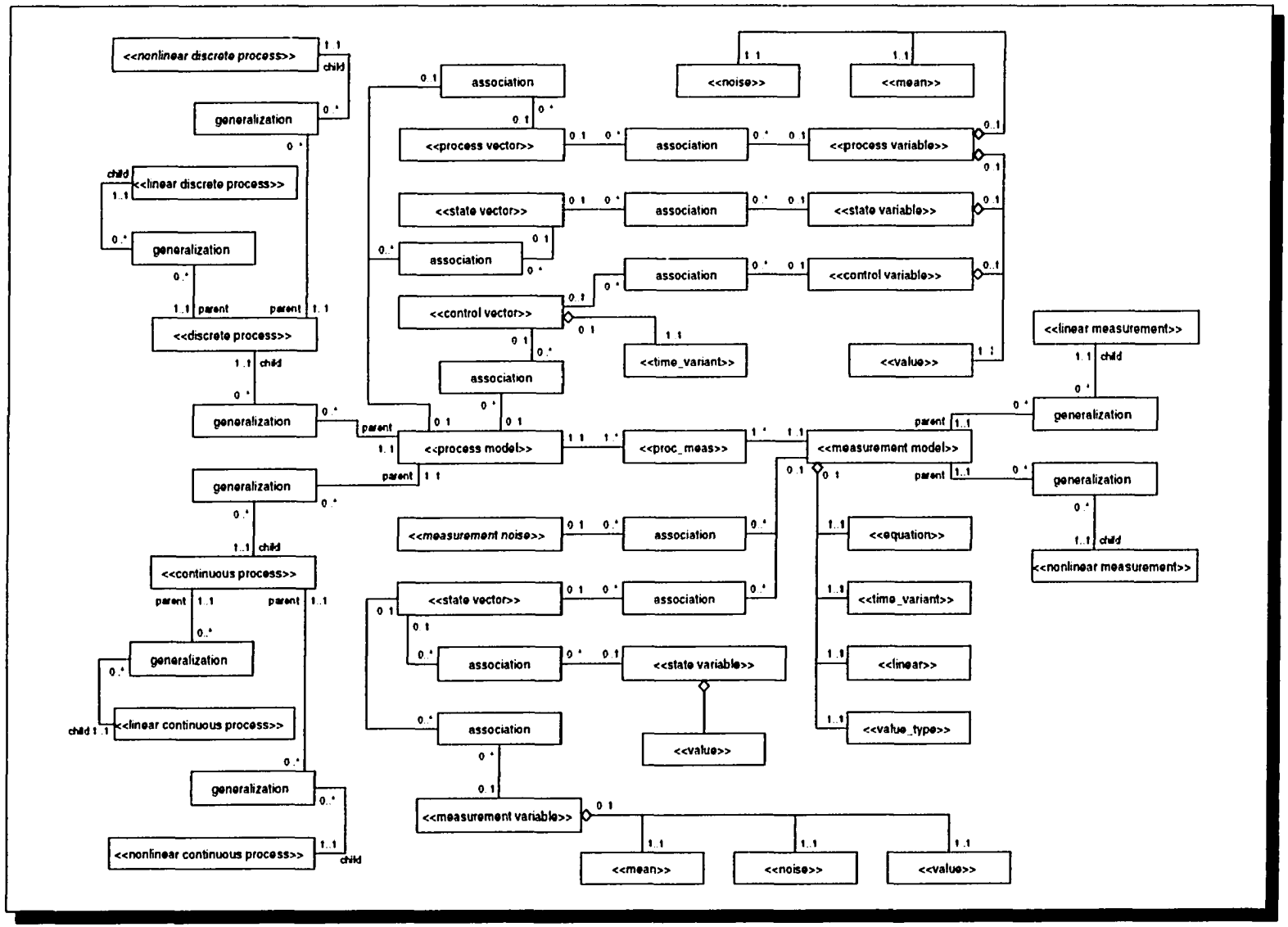

Figure 6.6A AutoFilter DSML design CD input meta-model

Figure 6.6B AutoFilter DSML design CD input meta-model

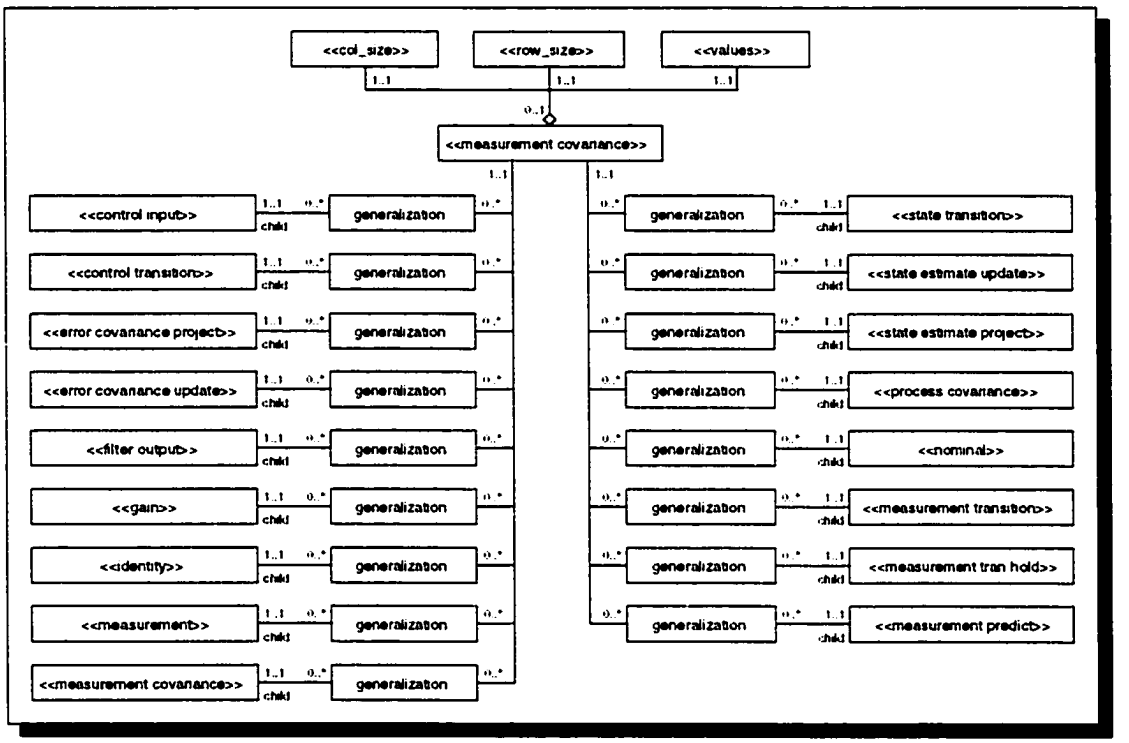

Figure 6.6: C *AutoFilter* DSML design CD output meta-model cont'd

1. In order for experience to be reused it has to be applicable to more than a single application (program). If experience gained during the development process is to be reused then there must be other required development task that are similar to the first.

2. Application of reusable experience has to occur within some definable '*domain*', i.e. the application of experience (gained from work on one task) in solving a similar but different task results in confinement of that experience to a family of tasks that have some definable commonalities and variabilities.

3. As the reusable experience applies to a family of software development task then that experience must abstract away from the *specifics* of each task in the domain. Consequently, it can be concluded that reusable experience is best applied at a stage in the development process that is above the lowest level of software development.

4. For reuse of experience to be viable there has to be some '*precise*' means of representing and expressing that experience.

The above list of issues raised a number of additional questions that had to be answered either as a prelude to the research or as a component of the research. The more pertinent of these questions as they evolved from the above issues respectively, are:

1. How are the similarities of the tasks to be identified and documented for the application of reuse experience?
2. What constitutes a family of application in a domain, and what are the issues of representation and documentation?
3. At what level of abstraction is the reuse experience to be applied, and what are the format and representation of this abstraction.
4. How formal can the representation of development experience be made, so as to facilitate precise inter- and intra- development team communication?

Armed with these issues and questions the research proceeded along the outline presented in Chapter 1.

The process of domain analysis and design (DA&D) [56, 96] was identified as a mechanism for addressing the issue of identification of commonalities and variabilities in a family

of application requirements. DA&D would also determine the scope (range) of application requirements that would be included in a domain in which the reuse of development experience is to be exploited. From an analysis of the the quantity and quality of the research that had to be conducted to address the remaining issues and questions, it was determined that DA&D would not be researched in this work, but applied in formulating the theory and application of the research work. This lead to a review of established DA methodologies from which a sub-set of processes and products that would satisfy the requirements of this research were synthesize. This review and selection of processes and products is set out in Section 2.4.

The research next focused on the examination of work related to the representation and use of information about families of application requirements. This, supported by the knowledge obtained from the aforementioned review of DA&D led to an examination of domain-specific languages (DSL). Based on the research on DSL, as documented in Section 2.2, the following observations about the general trend in the definition and use of DSLs are noted :

- DSLs are textual, and are usually directly translated to code.
- DSLs are designed for application at the implementation phase of software development.
- DSLs lack a formal definition of their syntax and semantics.

These observation led to the conclusions that current trends in DSL development do not provide much support for the : (1) abstraction of application requirements, or architectural design above the implementation phase of software development; (2) modeling of systems, because of their mainly textual notation; (3) incorporation into CASE tool, because of their incomplete syntax and imprecise semantics.

The perceived need for a '*modeling*' notation that is graphical in nature as a mechanism for abstracting away from the implementation level of software development necessitated an examination of the current notations in use. The *Unified Modeling Notation* (UML) [44] was identified as the most suitable notation for this work, because it:

1. "*Provide users with a ready-to-use expressive visual modeling language to develop and exchange meaningful models.*

2. "*Furnish extensibility and specialization mechanisms to extend the core concepts.*
3. "*Support specification that are independent of particular programming languages and development processes.*
4. "*Provide a formal basis for understanding the modeling language.*
5. "*Support higher-level development concepts such as components, collaborations, frameworks and patterns.*
6. "*Integrate best practices.*" [43]

The next stage in the research was the definition and description of the framework that bounded the scope of the intended work. This was outlined in the introductory section of Chapter 1 both textually and graphically.

Along with the description of the research framework an operational framework for the use of the deliverables was developed as described in Section 1.2. The focus of this research is on *Domain Language Engineering*, where techniques for defining and using DSMLs are specified.

The next phased of the research looked at the formulation of a strategy for accomplishing the goals and producing the deliverables. The strategy decided on called for the definition of the concepts that are to be developed in support of the thesis. These definitions are: (1) *language* as it is to be used in this research; (2) *domain-specific modeling languages* (DSML); and (3) *precise syntax* and *semantics* for DSML. Along with the above definitions, strategies for formulating the following processes were also developed: (1) specialization of the UML so as to constitute a DSML; (2) defining the components of DSML; (3) developing DSML; and (4) using the DSML to create ASM.

The decision was taken at an early point in the research that the abstraction of commonalities and variabilities in a domain would be most advantageous to software development if it could be expressed at or near the highest (earliest) point in the development process. This view is based on the premise that as development progresses through the different stages of software development errors are introduced at each stage, this is in addition to error that are carried through from the earlier stages of development.

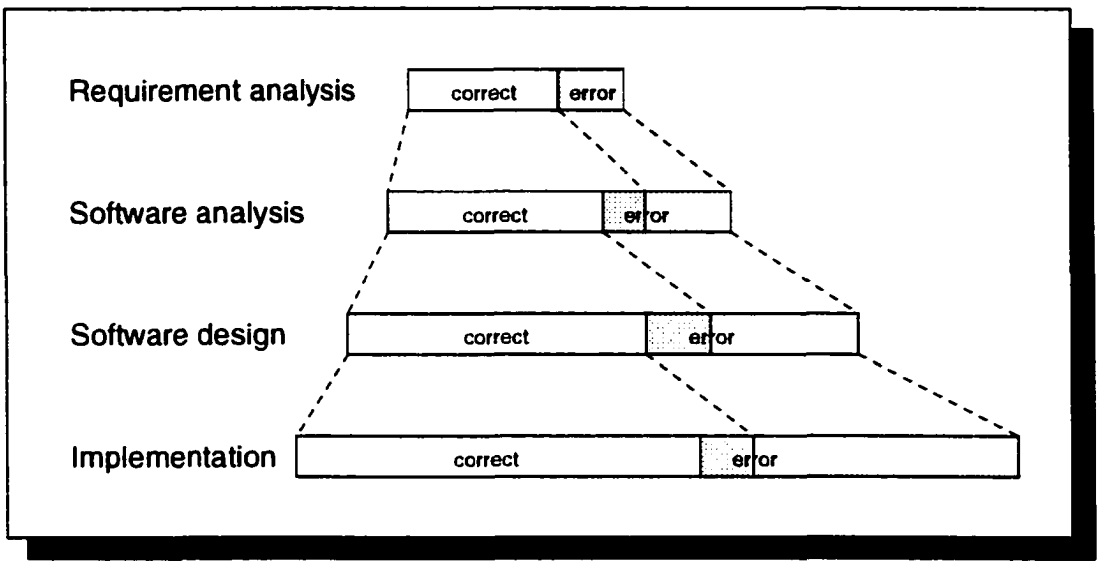

Figure 7.1: Error propagation in software development

This idea is illustrated in Figure 7.1[1]. Thus the intent is to use standardized constructs starting as early as possible (earliest being at requirement analysis) in the development cycle. These standardized constructs would be less error prone, having being tested and formally verified (to some degree), before being used in application development. Thus with a reduction in the number of errors introduced at each phase of the development cycle a more reliable, and maintainable application that is delivered in a timely manner would be produced.

Experience gained from prior work on integrating formal notation (the Z notation [77]) with informal notation (the UML [43]) [30, 31, 32] provided a springboard for the use of UML's Object Constraint Language (OCL) in specifying the domain-specific semantics of DSMLs in a precise format. The syntax of DSML is applied as it is defined in the UML specification [43] and constrained, when necessary, with the extension mechanisms. The semantics for DSML is similarly defined in the UML, and may be further refined in a denotational [84] manner. The syntactic model elements of DSMLs (original or specialized

[1]Reproduced from Dr. Robert B. France's lecture notes with modification.

UML model elements) may be mapped to a concept in a formal domain. In this manner the '*meaning*' of the DSML model element is refined by the semantics of the concept that it is mapped to in the formal domain. This mapping **must not** violate the original UML semantics of the syntactic concept, as defined in the specification. This mapping, of UML model element to a formal concept is illustrated in Figure 2.2.

It was not possible to define complete precise (formal) semantics for all DSML constructs. Formal definition for these constructs (model elements) is **limited** to the precision offered in the UML specification in conjunction with any additional mapping (denotational) from the model elements to concepts in a formal semantic domain . Because of this limitation the semantic definitions for DSML model elements may not be in a one-to-one correspondence with the informal textual specification of the elements' meanings. This is because it may not be possible to map all textually specified *meaning* to a formal concept. Hence the precise (formal) semantics can be supplemented with textual description. The mapped semantics and constraints must not violate the UML specification semantics. The practice of supplementing the formal semantics with textual description is currently how the UML semantics is defined. This limitation is one that was identified as being of fundamental importance to this work and will be referred to again in the following Section 8.2 on future work.

The research done on language development looked at the definition and description of the concept of language in various domains and identify the common aspects of such definitions and descriptions. These aspects (syntax and semantics) were then defined in terms of DSML. The conclusion reached is that for DSML, the syntax of the language is textually captured in a set of stereotype definition and graphically represented as domain meta-models. Such meta-models are analogous to the UML abstract syntax, i.e. they both reside at the M2 level of the MOF (Meta Object Facility) [74]. The semantics of DSML is expressed in the stereotypes as OCL statements, which may be supplemented with textual statements. The OCL statements specify the semantics in a formal notation - by means of constraining the existing formal UML semantic statements and/or refining them by mapping informal concepts to formal concepts (denotational [84] mappings).

The informal domain-specific descriptions contained in the system requirements that are input to the DA&D process are transformed into formal domain-specific development rules (adaptation, composition, and refinement rules). *Composition rules* specify which UML concepts are used in the domain and how the syntax and semantics of the UML concepts are constrained so as to express the domain-specific semantics. *Composition rules* are those that specify what domain-specific constructs must be present in domain application specifications (domain commonality constructs), and *refinement rules* specify which domain-specific constructs may be, optionally, included in domain application specifications (domain variability constructs).

During the early years of the research it was envisaged that there would have been an additional rule for specifying how the DSML constructs could be further extended to account for requirements that were unknown at the time that the language was being specified. But this idea was abandoned because of the high degree of uncertainty involved. It was decided that the best way to deal with domain requirements that arose after DSML development is to provide *feedback* to the *Domain Analysis and Design* process, which would in turn generate additional/modified *Domain Models* and *Domain-Specific Development Rules*. These new/modified models and rules would then be used as input to the *Domain Language Engineering* process for language definition update. This is illustrated in Figure 1.2 by the *feedback* arrows.

It was decided that DSML in this work would be as fully UML compliant as is possible, thus DSML have to be able to represent the UML models (use case, class diagram, activity diagram, etc), using specialized UML model elements. Hence, DSML components are defined in terms of the UML models that are required for the domain. These models are represented as meta-models (the DSML syntax) in DSML. These meta-models abstract out the commonalities and variabilities of the domain applications and present them as domain architectural models (domain pattern models). Use of these domain architectural models (patterns) are governed by the composition and refinement rules contained in the domain stereotypes.

DSML, as presented in this work, are intended to be used by application develop-

ers/engineers in a CASE tool environment. Tool developers would encode the DSML in the CASE tool in a manner that is identical to that which is currently done with the UML, with one major exception. This exception is with respect to the interpretation of the semantics of the DSML. The syntax of the DSML is presented to the tool developer in an similar format to that of the UML, i.e. a set of meta-models (abstract syntax). But in the case of DSML the meta-models are domain-specific architectural and design patterns in the format of the UML model used in the domain.

The semantics of DSML is offered to a tool developer in the format of a set of stereotypes with OCL expressions and denotational mappings that constraint the model elements to the specific requirements of the domain. This is more precise and directed towards a single interpretation. The goal is that the encoding of the semantics of a DSML in all CASE tools will result in all such tools expressing the semantics of the DSML in an identical manner.

A simulation of a CASE tool environment that an application developer/engineer will work in is illustrated in Figure 4.9. Usage of DSML within a CASE tool environment is similar to current usage of the UML in the same environment. The difference is that with DSML the user (software developer) can leverage the development experience of domain experts, by use of the embedded domain architectural and design patterns that constitute the DSML.

7.2 Research Application

The work done in developing DSML for two real world industries have been documented in Chapters 5 and 6. In this section the experience gained from conducting the two case studies will be documented. This documentation is to identify how the actual development of DSML impacted the theoretical principles and assumptions of DSML that were developed from the research.

7.2.1 Service provisioning system

The service provisioning system (SPS)is used by a telecommunication company to automatically allocate and maintain service to it's customer base. In the initial stage of the work

there was a concern that working within one company would not provide a wide enough basis for defining *domain* models. But there were two factors which allayed this concern. The first is that the work involved two systems that were from two previously independent organizations. Second, within the domain of telecommunications a wide range of the systems have been standardized. This standardization has been under the auspices of the International Telecommunication Union (ITU)[2]. Specifically, the ITU Telecommunication Standardization Section (ITU-T) was established to replace the International Telegraph and Telephone Consultative Committee (CCITT) back in 1983. The mission of the ITU-T is "*... to ensure an efficient and on-time production of high quality standards covering all fields of telecommunications*"[3]. It was thus concluded that the work on this case study would be representative of the domain.

One of the first things learned about SPS in the telecommunication industry is its complexity. The complexity of such system is mainly due to its interfacing with multiple systems. Such interfacing systems include: (1) billing, (2) customer service, (3) marketing, (4) credit/background checking, and (5) network element processors. In addition SPS is made up of independent service provisioning sub-units for each service supported by the system. The first major task was identifying the bounds of the system under study. This was only possible after numerous meeting with different engineering groups, which added to the complexity of modeling such a system. The various engineering groups viewed the system from different perspective, thus making it necessary to be able to identify overlaps in the descriptions of the system given by the different groups.

Alpha, Inc. had made a commitment to modeling their systems using the UML notation and this commitment eased the work that had to be done as most members of their software development teams had some knowledge of the notation. This made communication much easier than had there been no understanding of the modeling language. The

---

[2]See their web-site at *http://www.itu.int/home/*

[3]*http://www.itu.int/ITU-T/info/itu-t/info.html*

extensive documentation for one of the systems examined not only providing a comprehensive description of that system, but also aided in filling in *gaps* in the documentation of the lesser documented system. This was due in part to both system apparently being developed along ITU-T standards.

Because DSML was not within the remit of the project with **Alpha, Inc.** all such work was done off-line during the life of the project. Consequently, issues that arose with DSML development (especially those that required input from **Alpha**'s engineers' input) were only dealt with tangentially during meeting with the engineers to discuss aspects of the project. This proved to be a major handicap to the DSML development process. Consequently, a lot of needed information had to be extrapolated from what information was available. This would bring into question the usefulness of the DSML so developed. This concern can be addressed by noting that the intent is to demonstrate that a DSML comprising a set of the components identified in Chapter 3 can be engineered by using the process outlined in Chapter 4.

Some questions on the usefulness of such a DSML are: (1) Were all of the commonalities and variabilities in the system modeled? (2) Were all the need models for this domain included in the DSML? These questions could be answered through consultation with the domain experts (**Alpha**'s engineers) or through an examination of the ITU-T standard. The outcome of either exercise would not invalidate the work done on developing the DSML for SPS domain - It may necessitate an expansion of the DSML components, but the underlying processes would still be used. Such modifications to a DSML, as developed under the process and principle in this work, have been considered. This is illustrated in Figure 1.2 by the feedback arrows, which indicate the path for language modification.

7.2.2 Code generator (synthesizer) system

The work at **NASA Ames** Software Engineering group is chiefly involved with the development of advanced tools for automated software synthesis and automated software verification and validation. The internship work involved the derivation of sets of constraints for the inputs and outputs of a system (*AutoFilter*) that synthesizes (generate) code that

implements a Kalman Filter.

The domain expert was intimately involved with the development of the domain models, and all aspect of the work over the ten week period. Development of the models went through a series of iteration which resulted in the modeler obtain a great depth of knowledge on the domain requirements. This acquisition of domain knowledge proved extremely useful as the work progress, as it provided the modeler with not just the vocabulary of the domain but also the ability to conduct discussions about the domain in the language of the domain expert. As a result of this level of discourse the team (both modeler and domain expert) was able to gain insight into the domain that was not apparent at the start of the exercise. It was from this collaborative effort that the team was able to re-scope the bounds of the domain at a point in the project that would allow a set of goals to be achieved within the time-frame. At the start of the internship it was expected that a complete set of static and dynamic models of the system would be developed. But once the modeler began to gain an understanding of the domain requirements, and the domain expert an understanding of the purpose of the various models the re-scoping of the domain and the work to be accomplished was done.

The internship team refocused its attention on the input and output of the code synthesizer (generator) system, *AutoFilter*. This approach to system modeling turned out to be a very insightful one, as there was a fair amount of skepticism as to whether the models would be an accurate representation of *AutoFilter*'s input and output if *AutoFilter* was treated totally as a *blackbox*. As it turned out it was a good decision.

At the point when there was satisfaction that the models were representative of the input/output an initial set of constraints were written up by the domain expert. Prior to that (during the iteration of versions of models) the domain expert realized that a number *constraint* issues were being highlighted by the modeling exercise. *The modeling activity was forcing the team to think about constraint issues beyond just those of association and multiplicities!* One such constraint issue that arose because of the modeling exercise led to the question of: How is *time variance* in the *Control Vector* related to *time variance* in the *Process Model*?

With respect to the *Process Model* time variance is related to the entire process equation and not a particular term of the process equation. A control input (*Control Vector*) to the process may or may not be present. If it is present it usually is *time variant*, and this *time variance* is different from the *Process Model time variance*, notwithstanding the fact that the control input (*Control Vector*) is a component of the process equation (*Process Model*).

This necessitated that the *Control Input* and *Process Model* be modeled with *time_variant attributes*, and constraints written so as to permit the *Process Model* to be *time_variant true* when a *Control Vector* is present with *time_variant false* and for a *Process Model* to be *time_variant false* when a *Control Vector* is present with *time_variant true*, even though the *Control Vector* (control input) is a component of the *Process Model* (process equation).

A number of other insightful discoveries that arose from the modeling activity, suggest that the activities of domain modeling and deriving domain constraints (domain rules) are not independent of each other. This is to the extent that it may not be possible to define a set of domain models and then defined the domain constraints, but rather to incrementally develop both domain features (model and constraints). This was also discovered in the **Alpha** case study where the models had to be modified once the constraints (domain rules) were being added to the models, vis-á-vis the domain stereotypes. In that case it was assumed that the needed modifications were as a result of having incomplete information about the domain models.

The decision that the static model (class diagram) would be sufficient to satisfy the established goals of the project was arrived at after the task was clearly defined as:

> **Develop a set of UML models of the input and output of *AutoFilter* that will be supplemented with domain-specific constraints for the purpose of verifying that the output is a valid representation of a filter that is derived from the input.**

During the early stage of the work it was thought that the exercise would have evolved into a static model checking [16] task, but this goal established requirements that were different from that of formal model checking [49]. In this case the models were worked on until it was

determined that they fully expressed the aspects of the system that were to be constrained and validated. The validation task involved ensuring that (1) the structure of the input equation and the output synthesized program structures satisfied the respective models, and (2) the output synthesized code variables and constant values are consistent with the input equation parameters and values.

After completing the constraint specification for the input and output models of *AutoFilter* the team looked at means to automatically check these constraints against instances of input equations and output filter programs. An UML OCL validation tool that was as a possible candidate. This tool (USE[4])

> ... *"is a system for the specification of information systems. It is based on a subset of the Unified Modeling Language (UML) [43]. A USE specification contains a textual description of a model using features found in UML class diagrams (classes, associations, etc.). Expressions written in the Object Constraint Language (OCL) are used to specify additional integrity constraints on the model. A model can be animated to validate the specification against non-formal requirements. System states (snapshots of a running system) can be created and manipulated during an animation. For each snapshot the OCL constraints are automatically checked. Information about a system state is given by graphical views. OCL expressions can be entered and evaluated to query detailed information about a system state."*[5]

A graphical illustration of USE's operational framework is presented in Figure 7.2, which has been reproduced from *http://www.db.informatik.uni-bremen.de/projects/USE/*.

Figure 7.2 illustrates that a *Developer*, using USE, first *specifies* the *Model with USE Specification*. The *USE Specification* is textual *UML* and *OCL*. The *Developer* then

---

[4] *http://www.db.informatik.uni-bremen.de/projects/USE/*

[5] *http://www.db.informatik.uni-bremen.de/projects/USE/*

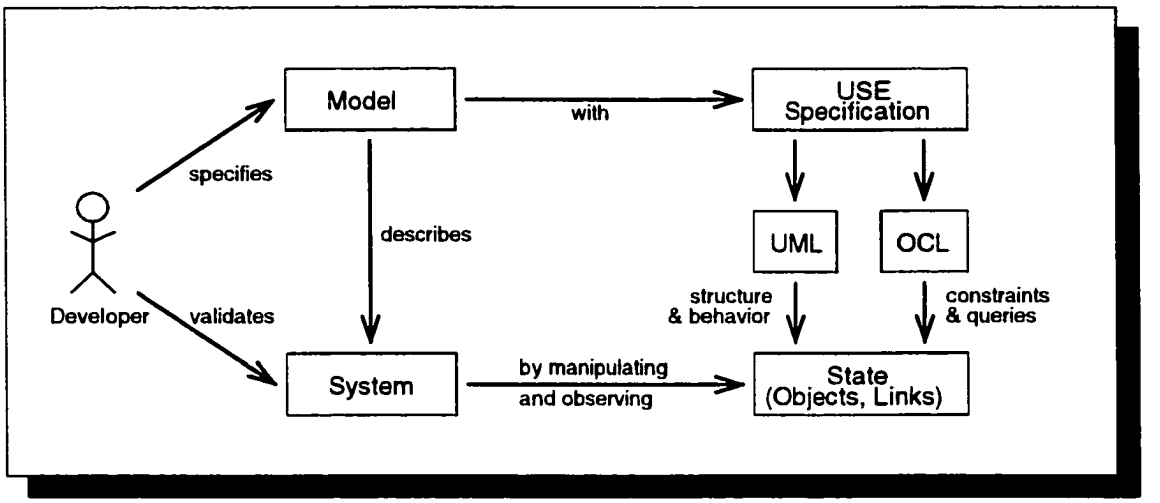

Figure 7.2: USE operational framework

validates the *System* (which is *described* by the *Model*) *by manipulating and observing* the *State (Objects, Links)*. UML specifies the *structure & behavior* of the *Model* and the *OCL* specifies the *constraints & queries* on the *System*.

The developers of USE alluded to ambiguities in the interpretation of the OCL due to "...*semi-formal definition*..." in some of its constructs. This became apparent when conflicts arose between the interpretation used in USE and the interpretation used in developing the *AutoFilter* input output models. Since it would be difficult to modify USE to accommodate the interpretation used on the models, the models and constraints were modified to accommodate USE's interpretations. This did not conflict with any of the written constraints, but necessitated that a number of the constraints had to be rewritten.

In the initial formulation of the constraints they were all specified within the single context of the *Kalman Filter* class, as this class was viewed as the *focal* class of the system. But because of an apparent error or limitation this could not be successfully done in USE. USE seems to sets a limit on the number of constraints that may be stated in a single OCL context. This limitation proved to be helpful, as the rewriting of the constraints within multiple contexts for the respective classes produced a more easily understood and modifiable set of constraints. Each constraint was now written as a *labeled* constraint within the context of a particular class for the model/constraint system. Examples from the set of

Declaration class constraints is presented in Table 7.1.

Table 7.1: Class *Kalman Filter* stereotype

```
context Declaration inv KalmanDeclaration10:
  self.kalman_Filter.loop.error_Covariance_proj_eq.oppositeAssociationEnds()→forAll(oae |
    Set{'Error_Covariance_project', Error_Covariance_update', 'State_Transition',
      'Process_Covariance'}→includes(oae.name.body))
context Declaration inv KalmanDeclaration12:
  self.kalman_Filter.loop.error_Covariance_updt_eq.oppositeAssociationEnds()rightarrowforAll(oae |
    Set{'Error_Covariance_update', 'Process_Covariance'}→includes(oae.nae.initialValue.body))
context Declaration inv KalmanDeclaration14:
  self.kalman_Filter.process_Model→forAll(pm | pm.linear.initialValue.body = 'false') and
    self.kalman_Filter.measurement_Model→forAll(mm | mm.linear.initialValue.body = 'false')) and
  self.kalman_Filter.process_Model→forAll(pm | pm.value_type.initialValue.body = 'absolute') implies
    self.oppositeEnds()→exists(oae | oae.name.body = 'Measurement_Predict)
```

The constraints of Table 7.1 are also specified in the stereotype Table 6.24 but have been written in more general OCL constructs in Table 6.24 than that of Table 7.1. The constraints of Table 7.1 are specific to USE's implementations (with minor additions). USE does not implement UML meta-attributes (ex. $initialValue$) but defines a set of functions (ex. oppositeAssociationEnds) that allow access to the values of UML meta-attributes.

Once all the constraints were written and syntactically checked with USE a set of constraints values were created. These constraints values represented an instance of a Kalman equation and filter. These values were equation/filters of actual domain problem/solution pairs that had previously been used in the domain. The examples ran successfully under the domain model/constraint system. When errors were introduced in these data sets the model/constraint system successfully detected the errors (conflicting model or constraint values).

Development of the DSML for input/output (equation/filter) models of the code generator (synthesizer) system was relatively simple once the constraints had been compiled. The only models developed for this system (and represented in the DSML) are the design level CDs. The meta-models of the CD defined the DSML syntax and the constraints constituted the semantics of the DSML. The semantics specify a formal meaning of the terms necessary for the successful definition of the DSML as outlined in this research. This provides for a more precise (less ambiguous) interpretation of the meanings of the language terms.

At the end of the NASA Ames Summer 2002 internship a presentation of the work done over the ten week period was made to the full research group, under which the project fell. This group comprised teams of researchers who are working on a number of different but related projects. At the end of the presentation a consensus was reached that the work presented had great potential for assisting the other research teams in accomplishing a common project goal of system verification. This is a fundamental goal for all software projects at NASA, as all software has to be formally certified before it is phased into productive use.

Chapter 8

Conclusion

The thesis of this research as quoted from Chapter 1 is that:

> **The definition of domain-specific object-oriented modeling languages in mature and well-defined software development domains facilitates extensive reuse of modeling experiences that are intrinsically embedded in the language components, and will result in the delivery of software systems that are produced from standardized domain modeling artifacts.**

The measure of the realization of this thesis is the extent to which the stated goals of the research have been achieved. These goals as established in Chapter 1 is for the deliver of:

- A relatively precise representation of the syntax and semantics of domain-specific modeling languages by using the UML extension mechanisms, and denotational mappings on the UML model elements used in defining the domain-specific modeling language.
- Techniques for defining domain-specific modeling languages as UML profiles.
- Demonstration of the development of domain-specific modeling languages.
- Application of the research concepts on case studies from real world organizations.

The remainder of this chapter is as follows:

- Sections 8.1 summaries the results of the research in terms of the experiences documented in Chapter 7, and the theories documented in Chapters 3 and 4.
- In Section 8.2 a proposal for future work that has been identified as having potential to advance some of the ideas and concepts discovered in this work will be presented. This proposal will also include ideas for the validation of some of the conclusions presented in this section of the report.

- Section 8.3 list a number of contributions that this work is deemed to have made to the research related to defining DSML, and the application of this technology in industrial environments.

8.1 Results

The results of this research work demonstrate that it is possible to define a process for developing DSMLs that are package as UML profiles, and such DSMLs are derived from the specialization of the UML meta-model elements by use of the extension mechanisms of stereotype, tag definitions, tag value, and OCL constraints.

This conclusion of the result of this work is arrived at based on the earlier review of the development of the theory for DSML definition and the application of this theory (Chapter 7). The review presented in Section 7.1 document the experience of developing the theory for defining DSML in the form of UML profiles. It has been demonstrated that the theory for defining DSML is consistent with the specification of the UML. The process for defining DSML, as outlined in Chapter 4 conforms to the specified usage of the UML model elements and the UML extension mechanisms. In addition, the components of DSMLs, as outlined in Chapter 3, are all UML components that are either used as they are defined in the UML Specification [43] or specialized into domain-specific components by the application of the UML extension mechanisms on the UML model element.

The works documented in Chapters 5 and 6 demonstrate that the process for defining DSML (as outlined in Chapter 4) that are composed of various components (as outlined in Chapter 3) is consistent and produces a DSML that conforms to the stated goals of Chapter 1.

DSMLs, as defined in this research, comprise a syntax and semantics. The syntax is defined by the meta-models of the domain models that are developed for the associated domain. These meta-models (the DSML syntax) capture the patterns of the domain, and thus implicitly embed some aspects of development decisions that were made in defining the domain models. The semantics of the DSML is specified in the stereotypes of the DSML. The semantics is obtained (firstly) from the inherited semantics of the UML model elements

that appear in the DSML, and (secondly) by the denoted semantics of the formal concept that the DSML model elements are mapped to. This inherited and denoted semantics may be further constrained by the use of OCL statements. Similar to that of the DSML syntax, the DSML semantics implicitly embed aspects of development decisions for the domain.

The document results support the conclusion that this research work was conducted in a manner (as outlined in Section 1.2) that successfully led to the accomplishment of the stated goals of Section 1.2.

8.2 Future Work

During this research a number of related areas of research were explored. Some of these, which impacted significantly on this work are: (1) Integration of informal modeling notation with formal notation [30, 32]; (2) Domain analysis and design [56, 95, 96]; (3) The standardization of modeling notations [43]; and (4) Modeling language development [44]. These areas of research were identified in the opening paragraphs of Chapter 1 and are illustrated in Figure 1.1.

Other areas of research that were explored, but had a lesser impact on the core goal of this research are: (1) Internet-based software system development [37]; (2) Formal notation graphical editor [36]; (3) CASE tools software modeling inter-operability [38]; and (4) Automatic code generation from UML models [37]. Some of these lesser related areas of research may still impact the core goals of this research as it is continued beyond what has been documented in this dissertation.

The main areas of possible continuing research that are expected from this initial work are outlined in the following sections.

8.2.1 NASA Ames code generator system

During discussions with the research team that supervised this internship it was planned that continuing research with this work should be attempted in the following manner:

1. The chief modeler (the intern) would proceed with a full documentation of the work that was accomplished during the internship. The intent of this is for the preparation

of a number of papers for submission to relevant academic or professional conferences and or journals. In addition the intern would be able to use the documented experience towards personal academic work.

2. A review of the internship work is to be done towards formulating a strategic plan for the definition of a *full* DSML for *AutoFilter* family of systems. This DSML, is not only intended to be used for the validation of the input equations against the output filter code, but also for the development of other filter code generators. Thus, models beyond those that were developed in this internship (class diagrams) are to be considered.

3. Along with the definition of a DSML for filter code generator systems, further research consideration should include enhancing and extending the USE tool so as to be able to automatically check the full UML specification with respects to the models used defined in the DSML. Alternatively, other OCL validation tools are to be identified and considered for possible use in place of USE.

4. A short term plan is to be developed for the continuation of the summer 2002 work during the summer 2003 period. This plan will focus on completing some aspects of the summer 2002 work that required additional components, and delivering a near production ready *AutoFilter* input/output (equation/filter) validation system.

5. At a suitable date in the future the principal NASA researcher, intern, and other relevant parties would meet to discuss compiling proposals for the funding of continuing research of this work. The intent is to make this a joint academic and industrial funded project, with the involvement of multiple universities, professional institutions and funding agencies. This will be pursued after there has been some feedback from other researchers, vis-á-vis conference/journal publications of the work done over the summer 2002 internship.

This single proposal for continued research work is vast and will require the commitment of a large amount of resources to successfully achieve some of the anticipated benefits of

such a project. Should this project be realized it would provide validation for the DSML principles documented in this research.

8.2.2 OCL verification

One of the lessons learn during this research is that for the use of OCL (or any other formal notation) to be successfully applied in a productive system there has to be a means to automatically check (validate) the OCL (or any other formal notation) that are specified. The experience gained in using USE to validate the stereotype constraints defined in the *AutoFilter* code generator system clearly demonstrated that without automatic checking more errors could be introduced in the application models developed using such a DSML. It is therefore necessary for continued research to be conducted in this area (*verification and validation of formal constraint specification*).

A starting point in this area of continuing research is with the proposal for extending and enhancing the USE tool. Based on the experience gained during the NASA internship with USE it is expected that extending the capabilities of USE so as to be able to specify additional aspects of UML (e.g. handling the specification of UML model elements meta-attributes) would make USE an efficient tool for OCL validation and verification. This enhancement to USE would require extensive research on the interpretations to be taken for UML concepts that may have imprecise specification. This is necessary as verification and validation of OCL statement has to be consistent in usage.

8.2.3 Language syntax and semantics research

There is an intense amount of research currently taking place within the software development community on the definition of modeling language syntax, and more intensely on the definition of modeling language semantics[1]. Chief among this is the work on formalizing the semantics of the UML by the OMG.

---

[1]See the web-sites *http://www.omg.org* and *http://www.puml.org*

Within the *p*UML research group there is ongoing discussions[2] on the use of a denotation semantics for formal specification of the UML semantics. The ideas being discussed in this forum are similar to the approach in this work, and outlined in Section 2.1.2.

8.2.4 Other research areas

In addition to the above main areas for continuing research that have been presented in Sections 8.2.1, 8.2.2, and 8.2.3 the research areas listed in the opening paragraphs of Section 8.2 are strong areas for continuing research. Since these research areas offer support for the definition of domain-specific modeling languages that are based on the UML, any advancement in research in any of these areas should result in advancement in research on defining UML-based DSML.

8.3 Contribution

A number of academic and industrial contributions have been by this work to the research fields examined as identified in Figure 1.1. The main contributions are as follows:

1. Definition of a UML-based Concept Context Model (CCM). Prior to this research there was no known explicitly defined UML-base model diagram that captures the relationship between the static and dynamic concepts at the analysis level of system development. This CCM may be viewed as a variation of the UML use case diagram, where the use cases are supplemented with the relationships of static concepts rather than actor-use case relationships.

2. The process for defining a *precise* semantics for the concepts used in DSML, i.e. (1) inheritance of UML semantics, (2) denotational-like mappings, and (3) refining the semantics of (1) and (2) by constraining them with OCL statements, is a process that is being heavely debated and there is no one technique that dominates. This work has put forward a process that affords a relative degree of precision that has proven to be

[2] *p*UML email list at *puml−list@cs.york.ac.uk*

sufficient to meet the requirements of the NASA case study. Further, the process for defining precise semantics in this work takes into consideration the situation where *total* precision is not achievable.

3. DSML that have been researched in this work are specified in terms of a specified set of domain artifacts (usually parameterized). This is similar to defining a language as a set of sentences with subsitutability of some words. This work defines a DSML in the formal sense of a language, i.e. *syntax* and *semantics*. The syntax is given in terms of a set of UML models that capture patterns of the domain and the semantics is as outlined above. These features make DSML (of this work) applicable in multiple domains as the components are derived from the UML and not artifacts of any domain it is developed for.

4. The process presented in this work for defining DSML is applicable to a wider range of domains. Its an implicit hypothesis of this work that *whereever the UML is applicable this type of DSML is also applicable* - with some limitation. None of the DSMLs reviewed in this work seems able to make such a claim of applicability of their development technique across diverse domains. This feature is fundamental to research on reuse (which holds great potential for making advances in software development). The DSML development process of this work meets the requirements for the application of DSML technology, as outlined in Section 2.3.

The contributions listed above are complemented by the successful application of this DSML technology in the case studies of Chapter 5 and 6, and in addition the committment of the NASA research group to advance this work by developing a more extensive version of the initial DSML defined during the case study. This demonstrates that the DSML technology of this work has made significant contributions to work in the field of software development.

REFERENCES

[1] Sinan Si Alhir. What is the Unified Modeling Language (UML)? Technical report, Sinan Si Alhir web site, http://home.earthlink.net/~salhir, August 1998.

[2] Guillermo Arango and Rubé Prieto-Díaz, editors. *Domain Analysis Concepts and Research Directions.* IEEE, 1991.

[3] Guillermo Arango and Rubén Prieto-díaz. Introduction and overview: Domain analysis concepts and research directions. In *Domain Analysis and Software Systems Modeling.* IEEE Press, 1991.

[4] Maher Awad, Juha Kuusela, and Jurgen Ziegler. *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion.* Prentice Hall, Inc., 1996.

[5] Victor R. Basili and H. D. Rombach. Support for comprehensive reuse. Technical Report UMIACS-TR-91-23, CS-TR-2606, University of Maryland at College Park. Department of Computer Science, 1991.

[6] Don Batory. Product-line architecture. In *International Conference on Smalltalk and Java in Industrie und Ausbildung.* Eifury, Germany, 1998.

[7] J.A. Bergstra and C.A. Middelburg. Process algebra semantics of phiSDL. In C. Verhoef A. Ponse and S.F.M. van Vlijmen, editors, *Proceedings ACP '95*, pages 309–346. Eindhoven University of Technology, Department of Computing Science, 1995.

[8] Gary Blahnik. Linguistics theory, foundation, and modern development: An overview of linguistics and linguistics applications. Technical report, The Union Institute, 1995.

[9] Grady Booch, James Rumbaugh, and Ivar Jacobson, editors. *The Unified Modeling Language User Guide.* Addison-Wesley Publishing Co, 1999.

[10] David Bruce. What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation. In *DSL'97 First ACM SIGPLAN Workshop on Domain-Specific Languages*, pages 17–35. ACM SIGPLAN, ACM, January 1997.

[11] Wray Buntine. Will domain-specific code synthesis become a silver bullet. *Intelligent Systems*, 13(2):9–12, March/April 1998.

[12] Tony Clark, Andy Evans, and Stuart Kent. Using profiles to re-architect the UML. In *Third International Conference on the Unified Modeling Languages (UML2000)*. York, England, October, 2000.

[13] Tony Clark, Andy Evans, Stuart Kent, and Paul Sammut. The MMF approach to engineering object-oriented design languages. In *Workshop on Language Description, Tools and Application*. Dipartimento di Informatica e Scienze dell'Informazione (DISI) Università di Genova, April 2001.

[14] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press and McGrew-Hill Book Company, 1990.

[15] David A. Cuka and David M. Weiss. Specifying executable commands: An example of FAST domain engineering. Technical Report Submitted to ACM on Software Reuse, Lucent Technologies, 1998.

[16] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report Research Report 159, Compaq Systems Research Center, December 1998.

[17] Jorge L. Diaz-Herrera, Peter Knauber, and Giancarlo Succi. Issues and models in software product lines. *International Journal of Software Engineering and Knowledge Engineering*, 10(4):527–539, 2000. World Scientific Publishing Company.

[18] Mahesh H. Dodani. Teaching practical object-oriented software engineering. In *OOPSLA '92 Educators' Symposium*. ACM, 1992.

[19] Desmond DSouza. Model-driven architecture and integration. Technical report, Kinetium, March 2001.

[20] Desmond F. D'Souza, Aamod Sane, and Alan Birchenough. First class extensibility for UML - Packaging of profiles, stereotypes, patterns. In Robert B France and Bernhard Rumpe, editors, *Second International Conference on the Unified Modeling Language* $\ll UML \gg$ *'99*, Lecture Notes in Computer Science, pages 265–277, Ft. Collins, Colorado, USA, October 1999. Springer-Verlag.

[21] Desmond F. D'Souza and Alan C. Wills, editors. *Objects, Components, and Frameworks with UML: The Catalysis*^{SM} *Approach*. Addison-Wesley Publishing Co, 1998.

[22] Martin Erwig. Abstract syntax and semantics of visual languages. *Journal of Visual Languages and Computing*, 9(v1980098):461–483, 1998.

[23] Andy Evans and Stuart Kent. Core meta-modeling semantics of UML: The pUML approach. In Robert B. France and Bernhard Rumpe, editors, *Second International Conference on the UML 1999*, volume 1723 of *Lecture Notes in Computer Science*, pages 140–155, Ft. Collins, Colorado, Oct 1999. Springer-Verlag.

[24] Andy S. Evans and Stuart Kent. Core meta-modelling semantics of UML: The pUML approach. In Robert France and Bernhard Rumpe, editors, *Second International Conference on the Unified Modeling Language*, Lecture Notes in Computer Science, pages 140–155, Colorado, USA, October 1999. IEEE Computer Scoiety, Springer-Verlag.

[25] Rickard E. Faith, Lars S. Nyland, and Jan F. Prins. KHEPERA: A system for rapid implementation of domain specific languages. In *Proceedings of the Conference on Domain-Specific Languages*, 1997.

[26] Software Technology for Adaptable Reliable Systems (STARS). STARS conceptual framework for reuse processes, Volume 1: Definition, version 3.0. Technical Report STARS-VC-A018/001/00, Unisys STARS Technology Center, October, 1993.

[27] Analysis & Design Performance Task Force. White paper on the profile mechanism. Technical Report ad/99-04-07, Object Management Group (OMG), Framging, MA, USA, April 1999.

[28] William Frakes and Carol Terry. Software reuse: Metrics and models. *ACM Computing Surveys*, 28(2):415–435, Jun 1996.

[29] Robert B. France, Maha Boughdadi, and Robert Busser. An industrial application of an integrated uml and sdl modeling technique. In *Proceedings of the 23rd International Computer Software and Applications Conference (COMPSAC'99)*. IEEE, 1999.

[30] Robert B. France, Jean-Michel Bruel, Maria M. Larrondo-Petrie, and Emanuel Grant. Rigorous object-oriented modeling: Integrating formal and informal notations. In *Proceedings of the 6th International AMAST Conference*, 1997.

[31] Robert B. France, Jean-Michel Bruel, Monika Saksena, Emanuel Grant, and Maria M. Larrondo-Petrie. Towards a rigorous object-oriented analysis and design method. In IEEE Computer Society Press, editor, *Proceedings of the 1st IEEE International Conference on Formal Engineering Methods*, 1997.

[32] Robert B. France, Emanuel S. Grant, and Jean-Michel Bruel. UMLtranZ: An UML-based rigorous requirements modeling technique. Technical report, Colorado State University, Ft. Collins, Colorado, January 2000.

[33] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable-Objected Software*. Addison-Wesley Publishing Co, 1994.

[34] John H. Gennari, Samson W. Tu, Thomas E. Rothenfluh, and Mark A. Musen. Mapping domains to methods in support of reuse. *International Journal of Human-Computer Studies*, 41:339–424, 1994.

[35] Martin Gogolla. Using ocl for defining precise, domain-specific uml stereotypes. In Aybuke Aurum, editor, *6th. Australian Workshop on Requirement Engineering (AWRE'2001)*, Sydney, Australia, 2001. University of New South Wales, UNSW Printer Room.

[36] Emanuel Grant. A graphical editor for the z notation. Technical report, Florida Atlantic University, 1997.

[37] Emanuel Grant, Robert B. France, Ramchander Varadarajan, Adam Carheden, and Jean-Michel Bruel. UML2Z: An UML-based modeling tool for an Internet integrated formalization process. In S. Patel D. Patel, I. Choudhury and S. de Cesare, editors,

6th International Conference on Object Oriented Information Systems, pages 280–289. Springer-Verlag, December 2000.

[38] Emanuel S. Grant, Jean-Michel Bruel, and Adam Carheden. HTUML: An UML-based object-oriented modeling notation for an internet integrated process. Technical report, Colorado State University, Ft. Collins, Colorado, May 1999.

[39] Martin L. Griss. Software reuse:From library to factory. *IBM System Journal*, 32(4), 1993.

[40] Martin L. Griss, John Favaro, and Massimo d'Alessandro. Featuring the reuse-driven software engineering business. *Object Magazine*, 1997.

[41] William E. Grosso, Henrik Erikson, Ray W. Fergerson, John H. Gennari, Samson W. Tu, and Mark A. Musen. Knowledge modeling at the millennium (The design and evolution of Protégé-2000). Technical Report SMI-99-0801, Stanford Medical Infomatics, Stanford University, 1999.

[42] Object Management Group. Action semantics for the UML. Technical report, Object Management Group, Framingham Corporation Center, MA, USA, 1999.

[43] Object Management Group. *OMG Unified Modeling Language Specification.* Object management Group, Needham, Massachusetts, USA, v 1.4 edition, September 2001.

[44] Object Management Group. *OMG Unified Modeling Language Specification (Action Semantics).* Object management Group, Needham, Massachusetts, USA, uml 1.4 with action semantics edition, January 2002.

[45] Profile Working Group. Requirements for UML profiles. Technical Report version 1.0, Object Management Group (OMG), Framingham, MA, USA, October 1999. Analysis and Design Platform Task Force.

[46] David Harel and Bernhard Rumpe. Modeling languages:Syntax, semantics and all that stuff. Technical report, The Weizmann Institute of Science, Rehovot, Israel, 2000.

[47] Elliotte Harold. *XML Bible.* IDG Books Worldwide, 1999.

[48] F. Hayes-Roth, D. Waterman, and D. Lenat, editors. *Building Expert Systems.* Addison-Wesley Publishing Co, 1983.

[49] Gerald J. Holzmann. The model checker SPIN. *IEEE Transaction on Software Engineering*, 23(5), May 1997.

[50] Institute for Software Integrated Systems, Vanderbilt University. *GME 2000 User's Manual*, version 1.1 edition, April 2001.

[51] Ivar Jacobson. *Objet-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley Publishing Co, 1992.

[52] Ivar Jacobson and Martin Griss, editors. *Software Reuse: Architecture Process and Organization for Business Success.* Addison-Wesley Publishing Co, 1997.

[53] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, 1996.

[54] Fredrick P. Brooks Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[55] Peyton Z. Peebles Jr. *Probability, Random Variables, and Random Signal Principles.* McGraw Hill, 4th. edition, July 2000. ISBN: 0073660078.

[56] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analaysis FODA: Feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, CMU, 1990.

[57] B. Kirwan and L. K. Ainsworth, editors. *A Guide to Task Analysis.* Taylor and Francis Publishers, 1992.

[58] Cris Kobryn. A standardization odyssey. *Communications of the ACM*, 42(10):29–37, October 1999.

[59] Charles W. Krueger. Software reuse. *ACM Computing Survey*, 24(2):131–183, Jun 1992.

[60] Robert L. Kruse. *Data Structures and Program Design.* Prentice Hall, Inc., Englewood Cliffs, NJ 07632, USA, third edition, 1994.

[61] Robert Krut and Nathan Zalman. Domain analysis workshop report for the automated prompt and response system domain. Technical report, Software Engineering Institute, Carnegie Mellon University, May 1996.

[62] Craig Larman. *Applying UML and Patterns: An introduction to Object-Oriented Analysis and Design and the Unified Process.* Prentice Hall PTR, Upper Saddle River, New Jersey, USA, 2001. ISBN 0-13-092569-1.

[63] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Proceedings of WISP'2001*, Budapest, Hungary, May 2001. IEEE.

[64] Mark Lutz and David Ascher. *Learning Python.* Oreilly & Associates, April 1998.

[65] Ruth Malan, Reed Letsinger, and Derek Coleman. *Object-Oriented Development at Work: Fusion in the Real World.* Hewlett Packard professional books. Prentice Hall, Inc., Upper Saddle River, NJ, November 1995.

[66] Hiroshi Maruyama, Kento Tamura, and Naohiko Uramoto, editors. *XML and Java : Developing Web Applications.* Addison-Wesley Publishing Co, May 1999.

[67] Masao Matsumoto, Rubén Prieto-Díaz, and Wilhelm Schäfer, editors. *Software Reuseability.* Ellis Horwood Limited, 1994.

[68] John M. Moore and Sidney C. Bailin. Domain analysis: Framework for reuse. *IEEE Computer Society*, 1991.

[69] Paul Morrow and Michael Alexander. Domain-specific languages - Tools for better programming. *PCAI Magazine*, 13(1), Jan/Feb 1999.

[70] Lloyd H. Nakatani, Mark A. Ardis, Robert G. Olson, and Paul M. Pontrelli. Jargons for domain engineering. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, Austin, Texas, October 1999. USENIX Association.

[71] Peter Naur. Revised report on the algorithmic language ALGOL 60. *ACM*, 3(5):299–314, May 1960.

[72] Victoria Neufeldt and David B. Guralnik, editors. *Webster's New World College Dictionary*. Simon & Schuster, Inc., 1997.

[73] Greg Nordstrom, Janos Sztipanovits, Garbo Karsani, and Akos Ledeczi. Metamodeling - Rapid design and evolution of domain-specific modeling environments. In *Proceedings of the IEEE ECBS'99 Conference*, pages 68–74, Nashville, Tennessee, April 1999.

[74] Object Management Group (OMG). *Meta Object Facility (MOF) Specification*. Object Management Group (OMG), version 1.4 edition, April 2002.

[75] A. Paoluzzi and C. Sansoni. Programming language for solid variational geometry. *Computer Aided Design*, 24:349–366, 1992.

[76] John Y. Park, John H. Gennari, and Mark A. Musen. Mapping for reuse in knowledge-based systems. Technical Report SMI-97-0801, Stanford Medical Infomatics, Stanford University, 1997.

[77] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z*. C.A.R. Hoare Series Editor. Prentice Hall, Inc., second edition edition, 1996.

[78] Terrence W. Pratt. *Programming Languages Design & Implementation*. Prentice Hall, Inc., Engelewood Cliffs, NJ, USA, 1975.

[79] A. Rau-Chaplin, B. MacKay-Lyons, and P. Spierenburg. The LaHave house project:Towards an automated architectural design service. In *Proceedings of the International Conference on Computer-Aided Design (CADEX'96)*, pages 25–31, Silver Springs, MD, USA, 1996. IEEE Computer Society, IEEE Computer Society Press.

[80] Grigore Rosu and Jonathan Whittle. Towards certifying domain-specific properties of synthesized code. In *Verification and Computational Logic (VCL'02)*, Pittsburgh, PA, USA, October 2002.

[81] Bernhard Rumpe. A note on semantics (with an emphasis on UML. In Haiem Kilov and Bernhard Runpe, editors, *Second ECOOP Workshop on Precise Behavioral Semantics*. Technische Université at Múnchen, 1998.

[82] Stephen R. Schach, editor. *Classical and Object-Oriented Software Engineering*. WCB/McGraw-Hill Limited, 1998.

[83] Wilhelm Schäfer, Rubén Prieto-díaz, and Masao Matsumoto, editors. *Software Reusability*. Ellis Horwood Workshop Series. Ellis Horwood, West Sussex, England, 1994.

[84] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., Newton, Massachusetts, USA, 1986.

[85] Mary Shaw and David Garlan, editors. *Software Archirecture: Perspective on an Emerging Discipline.* Prentice Hall, Inc., 1996.

[86] Diomidis Spinellis. Lightweight languages as software engineering tools. In *Conference on Domain-Specific languages*, Santa Barbara, California, Oct 1997. USENIX Association.

[87] Thomas A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science.* Computer Science. Addison-Wesley Publishing Co, Reading, Massachusetts, USA, 1991.

[88] Mikael Svahnberg and Jan Bosch. Evolution in software product lines: Two cases. *Journal of Software Maintenance: Research and Practice*, 11:391–422, 1999.

[89] Scott Thibault, Renaud Marlet, and Charles Consel. A domain specific language for video divice drivers: from design to implementation. *IEEE Transactions on Software Engineering*, 25(3), May/June 1999.

[90] A. Uselton. Structured operational semantics for concurrency and hierarchy. In *10th STACS, Lecture Notes in Computer Science.* Springer-Verlag, 1993.

[91] Arie van Deursen and Paul Klint. Little languages: Little maintenance? In *Fourth International Conference on Foundations of Object-Oriented Languages.* Paris, France, 1997.

[92] Standard VHDL. *Language Reference Manual.* IEEE, std. 1076.1-1999 edition, March 1999.

[93] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl.* Oreilly & Associates, 3rd. edition edition, July 2000.

[94] Steven Wartik and Rubén Priéto-Diaz. Criteria for comaparing reuse-oriented domain analysis approaches. *International Journal of Software Engineering and Knowledge Engineering*, 2(3):403–431, 1992.

[95] David Weiss. Commonality analysis: A systematic process for defining famalies. In *Proceedings of the 2nd International Workshop on Development and Evolution of Software Architecture for Product Families*, 1998.

[96] David M. Weiss. Defining families: The Commonality Analysis. Technical Report Submitted to IEEE TSE, Lucent Technologies, 1998.

[97] Davis S. Wile and J. Christopher Ramming. Introduction to the soecial section: Domain-specific languages (DSL). *IEEE Transactions on Software Engineering*, 25(3):289–290, May/June 1999.

Index