CONFIGURATION WIZARDS AND SOFTWARE PRODUCT LINES

Ph.D. Dissertation by

Guillermo Jiménez Pérez

INSTITUTO TECNOLOGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

Monterrey, N. L. June 2003

CONFIGURATION WIZARDS AND SOFTWARE PRODUCT LINES

Ph.D. Dissertation by

Guillermo Jiménez Pérez

Instituto Tecnológico y de Estudios Superiores de Monterrey

June, 2003

Acknowledgments

Conducting the research work of a doctoral dissertation takes long time. Particularly, this work required longer time than a "normal" dissertation may involve, because it was necessary to synchronize and coordinate work with members working in different areas and in remote locations.

First of all, I would like to thank my family Saúl, Diego, and my wife Norma, whose love provided the support I needed to remain working in such long endeavor .

I am deeply indebted to my advisor, Don Batory who dedicated extra time to conduct my research. This dissertation would not be possible without all your support.

My local advisor, José Icaza, provided me with confidence to pursue this work and remained so confident of its successful end that motivated me to keep working besides all those additional responsibilities in my work. Thank you for your encouragement.

Special thanks to my other committee members, professors Raúl Pérez, Arturo Molina, and Ramón Brena. Thank you for your insightful feedback and your interest in my work.

Finally, I would like to thank Apolinar and Geraldine Luera, whose support and help were fundamental in the year we spent living in Austin, and in my countless travels to meet my foreign advisor. I appreciate your friendship and interest in the progress of my research.

Table of Contents

Chapter 1 Introduction. 1

2.1 Overview and Contribution 1

2.2 Outline 5

Chapter 2 Engineering Configuration Wizards 9

3.1 Background 10

3.2 Domain Engineering 11

3.2.1 Domain Analysis 11

3.2.2 Domain Design 13

3.2.3 Domain Implementation 14

3.2.4 Application Engineering 15

3.3 Proposed Method for Domain Engineering 15

3.3.1 Domain Modeling: Feature Models 16

3.3.2 Domain Design: GenVoca 19

3.3.3 Translating Feature Models to GenVoca Models 22

3.3.4 Domain Implementation: Wizlets as Components 24

3.3.5 Composition Validation 29

3.4 Configuration Wizards 34

3.5 Recap 39

Chapter 3 Vehicle Simulators Product-Line 41

4.1 Autonomous vehicle simulation 41

4.2 Static Parameterization in Java 43

4.3 Domain Model for Autonomous Vehicles 48

4.3.1 Kinematic Model of a Car 48

4.3.2 Kinematic Model of a Tank 50

4.3.3 Canonical forms of vehicle control 51

4.3.4 Feature Model 52

4.4 Domain design 54

4.5 Domain implementation 56

4.5.1 Wizlet implementation 56

4.5.2 Implementing Composition Verification 58

4.5.3 Configuration Wizard Implementation and Examples 61

4.6 Discussion 67

Chapter 4 Computer Numerical Control Systems. 69
5.1 Motivation . . . 70
5.2 Numerical Control . . . 71
5.2.1 NC Machine Tool Elements . . . 72
5.2.2 NC Programming . . . 75
5.2.3 Program codes (letter address) . . . 76
5.2.4 NC programming procedures . . . 78
5.3 Project's Goals . . . 83
5.4 A FODA Model for CNC Systems . . . 86
5.5 Hierarchical Models for CNC Systems . . . 87
5.5.1 MotionGenerator subsystem . . . 89
5.5.2 MotionControl Subsystem. . . . 94
5.6 GenVoca Models . . . 102
5.7 Compositional Implementation . . . 106
5.8 Design Wizard for CNC Systems . . . 107
5.9 Discussion . . . 109

Chapter 5 Credit Unions Product-Line . . . 110
6.1 Credit Union Management . . . 110
6.2 Static parameterization in Object Pascal . . . 113
6.3 Domain Model of Accounting Systems . . . 118
6.4 Domain design . . . 121
6.5 Domain implementation . . . 124
6.6 Discussion . . . 129

Chapter 6 Product-Line Evolution . . . 131
7.1 Evolution in product-lines . . . 131
7.2 Specifying product-line evolution . . . 133
7.3 Evolving product-lines with meta-generators . . . 138
7.4 Evolving configuration wizards . . . 140
7.5 Evolving a product-line of vehicle simulators . . . 151
7.6 Evolving a CNC product-line . . . 155
7.7 Limitations and advantages . . . 158
7.7.1 Limitations. . . . 159
7.7.2 Advantages . . . 161
7.8 Discussion . . . 162

Chapter 7 Related work 163
8.1 Product-Line Engineering Methods 163
8.1.1 FODA 163
8.1.2 FeatuRSEB 164
8.1.3 Feature Abstract Specification and Translation 164
8.1.4 Organization Domain Modeling 165
8.1.5 FODAcom 165
8.1.6 PuLSE 165
8.1.7 FORM 166
8.1.8 GenVoca 166
8.1.9 Discussion 167
8.2 Implementing Product-Lines 168
8.2.1 Aspects 168
8.2.2 Frameworks 168
8.2.3 Mixins 169
8.2.4 Components 169
8.2.5 Generators 170
8.2.6 Software Kits 170
8.2.7 Design wizards 170
8.2.8 Configuration environments 171
8.2.9 Discussion 171
8.3 Product-line evolution 172
8.3.1 Discussion 173
8.4 Recap 173

Chapter 8 Conclusions. 175
9.1 Proposed Methodology 175
9.2 Results and Contributions 179
9.3 Done Right 182
9.4 Done Wrong 184
9.5 Future Research 185
9.6 Recap 186

List of Figures

Figure 2.1 Software product-line engineering . . . 12
Figure 2.2 Feature diagram and elements . . . 17
Figure 2.3 GenVoca components in UML . . . 20
Figure 2.4 Types of components and implementations . . . 21
Figure 2.5 From feature models to GenVoca model . . . 23
Figure 2.6 Component compositions and inheritance. . . . 26
Figure 2.7 Composition verification . . . 32
Figure 2.8 Configuration wizard process model . . . 36
Figure 2.9 Configuration wizard's architecture. . . . 37
Figure 2.10 Specification interface for vehicle simulators. . . . 38
Figure 3.1 Type declarations and equations. . . . 44
Figure 3.2 Kinematic model for a car. . . . 49
Figure 3.3 Kinematic model for a tank. . . . 50
Figure 3.4 Trajectory control. . . . 51
Figure 3.5 Feature diagram of a vehicle simulator domain . . . 53
Figure 3.6 Hierarchical model of a vehicle simulators family . . . 54
Figure 3.7 Specification interface for vehicle simulators family . . . 61
Figure 3.8 Simple car. . . . 62
Figure 3.9 Specification interface car towing a trailer. . . . 63
Figure 3.10 Car with trailer. . . . 64
Figure 3.11 Specification of a tank simulator. . . . 65
Figure 3.12 Simulator of a tank. . . . 65
Figure 3.13 Specification of a motorcycle simulator. . . . 66
Figure 3.14 Two-wheel motorcycle with trailer. . . . 66
Figure 3.15 Tank towing a trailer is illegal. . . . 67
Figure 4.1 Canonical form of a NC system. . . . 75
Figure 4.2 . Machine tool motion path. . . . 81
Figure 4.3 Facing cycle (G72). . . . 82
Figure 4.4 Contour parallel (G73) . . . 83
Figure 4.5 Feature model of CNC domain . . . 86
Figure 4.6 Sub-feature model of MotionControl . . . 87
Figure 4.7 CNC's high-level architecture . . . 87
Figure 4.8 MotionGenerator subsystem . . . 90

Figure 4.9 MotionControl subsystem 95
Figure 4.10 MotionGenerator's types and implementations. 103
Figure 4.11 MotionControl's types and implementations. 103
Figure 4.12 Component instances for MotionGenerator. 104
Figure 4.13 Component instances for MotionControl. 105
Figure 4.14 CNCgen's user interface 107
Figure 5.1 The accounting process. 112
Figure 5.2 Type declarations and composition equations. 116
Figure 5.3 Feature diagram for general ledgers 120
Figure 5.4 Feature diagram for reports 121
Figure 5.5 Accounting systems architecture. 122
Figure 5.6 Hierarchical model for general ledgers 122
Figure 5.7 Class hierarchy of accounting systems 125
Figure 5.8 Specification interface for accounting systems 128
Figure 6.1 Product-lines can evolve into new product-lines 132
Figure 6.2 Initial feature diagram. 135
Figure 6.3 Simple feature diagram. 136
Figure 6.4 Complex feature diagram. 137
Figure 6.5 Meta-product-lines can produce application families 139
Figure 6.6 Specification and developer interface 141
Figure 6.7 Configuration wizards derivation from product line specifications . 146
Figure 6.8 Original interface of configuration wizard for vehicle simulators . . 153
Figure 6.9 Evolved interface of configuration wizard for vehicle simulators . . 154
Figure 6.10 Original interface of CNC configuration wizard 156
Figure 6.11 Evolved interface of CNC configuration wizard 157

Abstract

The idea of software product lines is suggested to reduce both development time and cost. In search of scalable approaches for deploying large-scale software product lines, researchers and practitioners have been conducting work in several largely intertwined fields. Two areas are component-based development and product-line architectures whose goal is that application families can be produced by integrating components as prescribed by the architecture. A third field is generator technology, whose aim is the automatic production of software from a (preferably) formal specification. A fourth technology is expert systems, developed in the artificial intelligence field, which demonstrated that when knowledge in restricted domains is well-understood, it can be conveniently structured, stored, and manipulated thus problems can be solved by following different reasoning chains appropriate to each particular problem, and explanations displayed to justify the resulting solution.

This dissertation shows that it is possible to define an approach that combines component-based development, product-line architectures, and generative technologies to construct expert tools for automatic software production called configuration wizards. A *configuration wizard* is a software assistant incorporating domain-specific topological knowledge (i.e. a product-line architecture) and a library of parameterized components which can be adapted to fit in different compositions, realizing members of a system family.

Configuration wizards have the additional advantage that with appropriate modularization, their evolution can be automatized by describing them as meta-models. These meta-models are processed by a meta-generator which produces specific configuration wizards.

Our work has several contributions in the area of product lines, as is showing that relatively simple generator-based tools are enough to produce application families, the identification of a general approach and minimal extensions to programming languages necessary to implement configuration wizards. Additionally, we show how the approach can be extended to produce configuration wizards from meta-specifications. This last idea is used to gracefully evolve configuration wizards to incorporate new features or exclude desired features.

Chapter 1

Introduction

1.1 Overview and Contribution

It is well-known that constructing software from available software modules can lead to higher productivity and quality than developing software from scratch [Par79]. It has been also observed that, software modules are better reused in different applications if they represent domain concepts [Gri00a] (i.e., *features* [KCH+90]). A *domain* is an area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area [BRJ98]. Domain features are useful in describing *application families*, which are groups of applications sharing characteristics but also exhibiting variability in their characteristics. In constructing application families, a set of existing and future software products can be analyzed to determine common and variable features, and a software architecture for the family can be derived, and an implementation strategy that represents this commonality in terms of a set of reusable software modules can be created [BCS00]. An application family designed and implemented to take advantage of a common structure, common features, and prescribed variabilities is known as a *software product-line* [WL99].

The idea of software product-lines is conceptually simple; however, its realization may not be. One reason is that applications from different domains may have fundamental differences that cannot be dealt with in the same way (e.g., real-

time versus information systems). However, the perspective that product lines can facilitate high-quality and economic application development, has started many research efforts to identify methods, tools, and models for software product-line development, both in industry and academia [Don00, Cha02]. As a result, different methods have been proposed to conduct product-line engineering. A common characteristic in these methods is that they are too general in the sense that almost any approach fits them (e.g. FAST [WL99]) yet others are tailored to be used in specific domains (e.g. FODAcom [VAM+98] in the telecommunications industry).

Other research in the area of software product-lines focuses in determining what a reusable module should be and how to implement reusable modules [Par72b, ASC00]. Two fundamental questions are how modules represent domain features, and what parameterization is necessary to allow modules to be used in different members of a product family. Answers to these questions provide guidelines on how to represent and implement modules from which product lines can be constructed.

Two complementary concepts associated to application construction in a product-line are configuration and composition. A *configuration* consists in the set of features and their arrangement in defining an application [SMB00, Sta00]; the set of modules implementing a specific configuration is a *composition* [Bat98]. How modules can be composed to construct applications from configuration specifications is determined at module design and implementation time.

In this dissertation we present a product-line development approach that combines best-of-breed approaches to analysis, design, and implementation of product-line infrastructures. Our approach introduces the concept of *configuration wizards*, which are tools to automatically synthesize software product lines in well-understood domains. A configuration wizard is a tool for application specification and generation from prebuilt parameterized software modules called wiz-

lets. A *wizlet* is a software module specifically designed and implemented to be composed with other wizlets using a configuration wizard as a tool to produce applications in a product line.

The specification interface of a configuration wizard displays a collection of domain features available to the application developer. A wizlet implements a feature and encapsulates semantic information describing its environmental requirements and constraints, and information describing the wizlet to the environment. When executed at application composition time, a wizlet analyzes its environment to determine any inconsistencies with respect to prescribed constraints for its proper functioning. If constraints are satisfied, the wizlet sends the configuration wizard a message describing itself.

The configuration wizard collects this information to produce a description (documentation) of the application, produces the application, and compiles it into executable code. If a necessary condition for a wizlet is not satisfied, the wizlet sends the configuration wizard a message so it can reject the specification.

This dissertation presents an approach to product-line engineering based on the implementation of configuration wizards, and describes its use in constructing three different configuration wizards in three disparate domains, using different programming languages. Our approach is unique in its proposed models and techniques for domain analysis, domain design, and infrastructure implementation.

An additional aspect considered in this dissertation is how the approach of configuration wizards supports product line evolution. Configuration wizards like the software they generate have to evolve to support changing requirements of a product-line. We propose a syntax for metaspecifying configuration wizards thus they can be adapted by metagenerators. We present two examples describing how configuration wizards can be evolved thus changes in requirements of the product line can be introduced.

Concretely, the main contributions of this dissertation are:

- A product-line engineering approach detailing the necessary steps to develop configuration wizards for product-lines, prescribing the work products along the process. The approach is generic thus configuration wizards can be designed and implemented to support product-line construction in different domains. Although our approach is not entirely unique in that it doesn't propose a completely new model, we integrate appropriate models for analysis (FODA's feature diagram [KCH+90]), design (GenVoca [BO92]), and implementation (configuration wizards and wizlets) for product lines.
- The notion of wizlets as parameterized reusable modules. Wizlets embody functionality and semantic information of how that functionality can be reused in different applications. A wizlet implements a domain feature [Esh98] and is parameterized by another wizlet (thus wizlets participate in collaborations or use case chains [JCJO93]). We show that requirements for wizlet parameterization are simple when wizlets are used as units of composition, so programming languages providing basic object-oriented capabilities (e.g., inheritance) and encapsulation facilities (e.g., classes, units, modules, etc.) could be extended to support wizlet parameterization using preprocessor tools. Further, we show how programming languages can be extended to support wizlet parameterization. In contrast, other approaches for component parameterization require complex extensions to existing programming languages and complex tools for application integration (e.g., subjects need a subject compositor [HO93], aspects need an aspect weaver [KLM+97, Gri00a], etc.).
- An approach to interface specification from variability and commonality described by feature diagrams. The identification of commonality and variability in the construction of product lines is essential for their automatiza-

tion. In a configuration wizard, commonality is encoded as topological (architectural) knowledge, variability is used to define simple specification interfaces. Using commonality and variability in this way simplifies the implementation of configuration wizards for automatic application production. Other approaches require more complex specifications (e.g., aspects require "aspectual" decompositions and relations [KLM+97], subjects require subject descriptions and composition rules [HO93]) or don't provide hints on how to build specification interfaces for different product families.

- A notation and approach to metaspecify configuration wizards, thus they can be evolved and adapted to produce applications from different sets of requirements. How to represent, manage, and deal with variability in product lines is an area of current research [Cha02]. Our approach is based on extending configuration wizards to implement metaconfigurators whose products are specific configuration wizards. Instead of using a single metaconfigurator to produce configuration wizards in multiple domains and software platforms, we show how different technologies can be used to implement metaconfigurators, and present two examples of metaconfigurators.

1.2 Outline

Subsequent chapters in this dissertation explore problems associated to producing configuration wizards for software product lines, explain our proposed approach, and contrast it to other work in the research literature. Our approach is characterized by four steps: domain analysis, product-line architecture definition, wizlet and configuration wizard implementation, and finally, application generation. These development steps are described and their work products presented for moving

from domain modeling to architectural modeling to wizlet implementation, and finally to configuration wizards. The approach, its simplicity, and scalability are shown by presenting three configuration wizards for different domains which were implemented using different programming languages. To show how our approach to product-line engineering support evolution, we introduce the notion of metaconfiguration wizards, and present examples of how these can be useful to evolve a product-line. We present results and conclusions of our work, and future research that remains to be done in the field of software product lines in general and configuration wizards in particular. The dissertation concludes with a presentation of technologies and techniques related to software product lines implemented as configuration wizards. The presentation of our work is organized as follows:

Chapter 2 sets out the foundation of our approach. It describes the steps associated with the implementation of software configuration wizards to assist in the construction of software product lines. We show that the result of applying product-line engineering is a set of domain models and architectural models describing the commonality and variability of applications in a family, representations of applications as wizlet compositions, and topological information describing valid and invalid compositions. Product-line engineering is extremely helpful in constructing the necessary infrastructures for implementing configuration wizards. Our approach for product-line engineering is based in the use of feature models and the GenVoca model of hierarchical systems. Chapter 2 describes both models for product-lines analysis and design. Additionally, Chapter 2 presents our implementation approach of parameterized components (wizlets) and their integration in applications by configuration wizards. Wizlets are similar to mixin layers [Sma99]. *Mixin layers* are large-scale components that can be used to directly implement collaboration-based designs using parameterized inheritance[1]. We describe how to implement wizlets as C++ templates. Finally, Chapter 2 concludes

with a discussion on configuration wizards as tools for product family production and their advantage over development environments lacking semantic verification capabilities.

To demonstrate the scalability of our approach of configuration wizards, in Chapters 3-5 we describe the development of three different configuration wizards in disparate domains, implemented using different development environments and programming languages. Every chapter discusses how the implementation technology used is appropriate to fulfill the requirements in each product line. First we describe each domain in detail and justify how that domain benefits from a product-line approach. We then show domain models and a brief discussion of wizlet implementation and composition for each domain. Finally, we discuss a configuration wizard and its capabilities in producing members of the family. These chapters are organized as follows.

Chapter 3 presents our first example, a configuration wizard for producing a family of autonomous vehicle simulators implemented in Java. This example is interesting in that it is simple and illustrative. It shows why Java extensions are necessary and how to proceed for implementing product lines supported by configuration wizards. Proposed Java extensions are a simple way to extend other programming languages not supporting template-like parameterization capabilities.

Our second example, presented in Chapter 4, defines a product line of computer numerical control systems. This product line is characterized by real-time constraints; corresponding wizlets and configuration wizard are implemented in C++.

---

1. Wizlets' syntax is equivalent to (Smaragdakis's) mixin layers. However at least two differences can be identified. First is that wizlets are meant to be instantiated by configuration wizards, not compilers. Second is the additional verification possibility that wizlets provide by using composition predicates (which are evaluated by the configuration wizard).

A third example is described in Chapter 5, which consists in a configuration wizard for a product line of general ledger systems implemented in Object-Oriented Pascal. Object Pascal does not support template-like parameterization, thus a language extension mechanism, similar to that presented in Chapter 3 is implemented.

In Chapter 6 we describe how software product lines infrastructures implemented as configuration wizards can be evolved by constructing metaconfiguration wizards, whose generated applications are configuration wizards. A notation is introduced to metaspecify configuration wizards and examples are presented describing how requirements are removed or introduced to the product line.

There are broad literature in the area of software product lines, both in methods and software technology [Don00, Cha02]. In Chapter 7 we discuss how our work is related to that of others.

Finally, in Chapter 8 we discuss our results, draw our conclusions, and provide insights on future work that remains to be done in the area of configurations wizards in particular and software product lines in general.

Chapter 2

Engineering Configuration Wizards

This dissertation proposes that software product lines can be constructed from pre-fabricated software modules called *wizlets* which are assembled by a tool called a *configuration wizard*. The input to a configuration wizard is a specification and the output is an application which is synthesized from wizlets that implement the specification.

The general approach involves a set of activities necessary to engineer a software product-line and is commonly known as *software product-line engineering* [WL99,CE00]. In general, the engineering of product lines encompasses two steps: domain engineering and application engineering. *Domain engineering* activities produce models and infrastructures (i.e., tools) by analyzing commonality in an application family and prognosticating its variability; our approach for domain engineering is concerned with implementing configuration wizards and their accompanying wizlets. *Application engineering* uses domain models and infrastructures to construct different product family members [SEI01]; in our approach, application families are generated from input specifications by combining and adapting wizlets implementing the specifications.

This chapter describes detailed activities, modeling notations, and work products in our proposed method for conducting product-line engineering based on configuration wizards and wizlets as components. Section 2.1 defines important

concepts used in this dissertation. Section 2.2 presents generic steps for engineering product lines without specifically committing to any particular notation, the idea is to set out the elements necessary to implement product-lines. Section 2.3 explains our proposed modeling notations and implementation techniques for product-line infrastructures. Section 2.4 describes configuration wizards, proposes an architecture and process for configuration wizards, and discuses its possibilities for constructing application families.

2.1 Background

A *software architecture* is the description of an application in terms of software modules and relations among those modules [GS93,SG96,BCK98]. An architecture defines the structure of an application, and provides some rationale for design decisions [PW92]. The structure helps to understand how modules are interconnected and how every module depends on others to fulfill application requirements.

Essential for the realization of product-lines is designing applications as sets of interacting modules [Par72b], that is, obtaining their software architectures. By inspecting application commonalities in a domain, a common architecture can be defined for an application family. This architecture is called a *domain-specific software architecture* (or simply *domain architecture*) [Tek94]. Every member in the family defined by the domain architecture will contain a subset of the modules in the domain architecture [BCGS95]. Domain architectures are blue-prints for constructing application families, they act as templates that can be adapted for implementing members in an application family. Different family members could differ in the modules they use (and may be in configuration parameters, such as memory size, performance, etc. associated to particular versions of modules).

Ideally, we would like to have a library of modules that can be used in constructing new family members. The domain architecture is helpful in constructing new applications by assisting developers in selecting the appropriate module implementations. However, it would be impractical to have different module implementations for every domain characteristic. More convenient would be to define parameters thus a single module can be instantiated in different applications by providing these parameters.

The following sections describe a generic method for domain-specific architecture definition and application construction for software product-lines.

2.2 Domain Engineering

Domain engineering involves the set of activities for modeling, architecting, and implementing infrastructures necessary to produce application families. As Figure 2.1 shows, domain engineering is the first step in producing application families, and involves three activities: domain analysis, domain design, and domain implementation. These activities are described in the following sections.

2.2.1 Domain Analysis

The goal of *domain analysis* is to scope and define a set of reusable requirements for applications in a domain. A *domain* is a set of concepts and terminology in an area that are understood by practitioners in it, and includes the knowledge of how to build software systems (or subsystems) to satisfy user requirements. Users are individuals, departments or organizations with a particular interest in an application family. Examples of users include end-users whose interest will be on usabil-

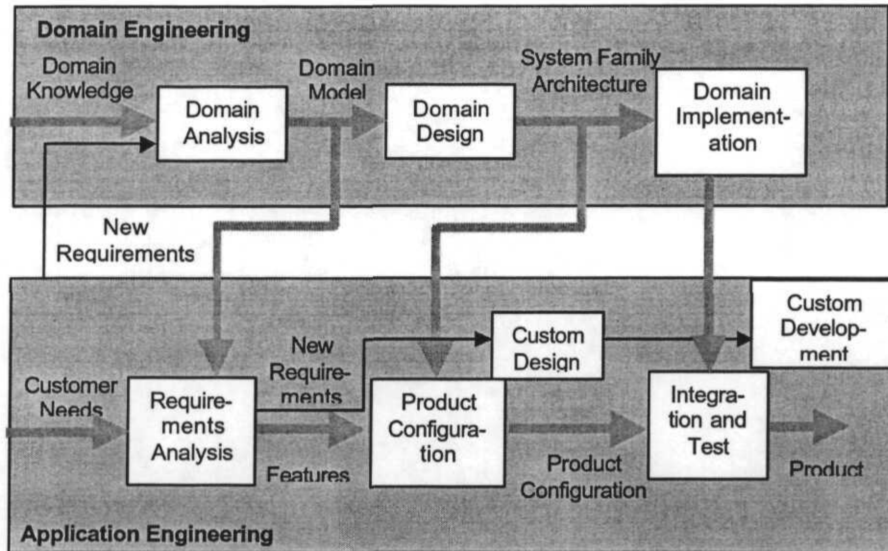

Figure 2.1: Software product-line engineering

ity and performance, those who perform maintenance that needs an evolvable implementation, those who are in marketing needing a timely product, etc.

The first step in domain engineering consists in analyzing the similarities and differences of several applications in a domain, and talking with domain experts to gather domain commonality and variability [CSPK91]. The *domain definition* determines the scope of a domain and characterizes its contents by giving examples of existing systems in the domain, counter-examples (i.e., systems outside the domain), and generic rules and rationale of inclusion and exclusion of a given system or capability [GFD97]. Once the domain has been selected, it is necessary to decide what belongs to it and what doesn't.

Once the scope of a domain has been demarcated, it is necessary to describe the domain concepts and their properties. A domain model helps to do this. A *domain model* explicitly represents common and variable properties of sys-

tems in a domain, semantics of the properties and domain concepts, and the dependencies among variable properties [JGJ97]. Domain models describe domain concepts using modeling formalisms. One particular representation of a domain model is a tree of domain features, called a *feature diagram*, which defines a set of reusable requirements for specifying systems in a domain by prescribing which feature combinations are meaningful, and which of them are preferred under which conditions and why [CSPK91]. Feature models represent the configuration aspect of the domain models and thus the configuration aspects of the whole reusable software. Section 2.3.1 describes feature diagrams in detail.

2.2.2 Domain Design

The second activity in domain engineering is *domain design*. During domain design we develop a domain-specific software architecture (i.e., a generic architecture) for the application family and devise a production plan for it. The architecture is defined in terms of a collection of software modules, interactions among these modules, and constraints on their interaction patterns [Tek94, GW94].

During domain design we also develop a *production plan*, which describes how concrete systems will be produced form the common architecture and the software modules. The production plan describes interfaces to the customers ordering concrete systems and the process of assembling the modules (i.e., manual or automatic assembly). A notation for representing product-line architectures are GenVoca models [BO92] which consist of hierarchical module compositions. GenVoca is described in Section 2.3.2. In our case, production plans are tightly coupled to configuration wizards, described in Section 2.4.

2.2.3 Domain Implementation

Domain implementation involves domain architecture, software modules, and a production plan implementation using appropriate technologies. Domain implementation may involve writing developer's guides, implementing domain-specific languages and graphical user interfaces (GUI's), generators, and establishing the software production process [GK96, WL99]. Software tools are selected and the way applications will be produced is chosen and implemented. Implementation approaches have to be carefully selected by observing domain commonality and variability for different members in the application family and the available infrastructure (e.g., compilers, frameworks, etc.).

Three possibilities arise when modules are integrated to construct applications. First is the case of simple domains in which modules can be (re-)used as is. For slightly complex domains, modules may require simple changes. In complex domains modules may require sophisticated adaptations to fit in particular compositions [Gri00b].

When modules are used as is or when simple modifications are enough, modules can be integrated by hand (i.e., the developer writes the specification and performs necessary adaptations). However, when modules have complex interdependencies or necessary adaptations are complex, it may be impractical to write specifications and perform module adaptations by hand. In such cases, it would be convenient to automate application production by implementing infrastructures to integrate modules into applications [GW94, GK96]. Several alternatives have been proposed to simplify module implementation and automatic adaptation at integration time (e.g., aspect-oriented programming [KLM+97], subjectivity [OH92, HO93, BG97], intentional programing [Sim95a], and software generators [BST+94]).

Technology is not the only factor involved, domain engineers are still responsible of effective domain implementation. Elegant representations can produce flexible and scalable module implementations yielding optimized components that can be combined in different ways (e.g., STL classes [MS96r], P2 components [Tho98], etc.) [BSST93, Big94, BR87]. Other approaches use more granular modules and there may be necessary to specify more complex compositions to construct application systems (e.g., role-based design [VN96a, VN96b, RG98]).

2.2.4 Application Engineering

The second step in a general product-line engineering approach is application engineering. *Application engineering* is the process of building concrete applications that implement customer needs based on the domain model (see Figure 2.1). This process is supported by the infrastructures developed in domain engineering [JGJ97].

As Figure 2.1 shows, for producing new applications in the product line, requirements of the new applications have to be determined and compared to those already implemented. The result of this comparison is a set of features already present in the modules plus, possibly new requirements not yet implemented [CSPK91]. Customer requirements not found in the domain model will require custom development, new modules may need to be implemented, and even part of the design may need to be customized.

2.3 Proposed Method for Domain Engineering

Section 2.2 described the general steps and activities involved in conducting domain engineering. The following sections describe the models we use in

conducting domain engineering that are used to develop configuration wizards. Previously we observed that our approach does not propose any new modeling notation, rather it uses models that have demonstrated their capability to represent domains.

2.3.1 Domain Modeling: Feature Models

Several different domain engineering methods have been proposed [Ara94]. They vary in how domains are represented, and how they make use of available domain architecture and application expertise. *Feature-Oriented Domain Analysis* (FODA) is a popular domain analysis method [KCH+90, CSPK91, GFD98, KKL+98, LKCC00]. Its success is due to the fact that different domains can be represented and understood by a set of features [Gri00b]. In FODA, a *feature* is any distinguishable characteristic of a concept (or element) in the domain of interest, usually features are end-user visible aspects of a software system [GFD97, Gri00a]. The FODA method represents features using a hierarchical model called a *feature diagram*, indicating whether features are mandatory, alternative, or optional [LKK+00]. A feature diagram includes information of which feature combinations are valid and which are not, and rationale for choosing or not a given feature in a particular instance of the diagram. The feature diagram specifies the common and variable properties of concept instances and their interdependencies and organizes them into a coherent representation.

Feature models consist of a domain description, a feature diagram and a domain dictionary [KCH+90]. The *domain description* provides a description of the problem space in the domain. The *problem space* defines what belongs to the problem at what lies outside it. A *feature diagram* captures a customer's or end-user's understanding of the general capabilities of applications in the domain[1]. The *domain dictionary* contains terms and/or abbreviations that are used in describing

the features and entities in the model and a textual description of the features and entities.

Figure 2.2 shows the notation of a feature diagram and its elements. A feature diagram consists of a set of nodes, a set of directed edges, and a set of edge decorations. The nodes and the edges form a tree. The root of a feature diagram denotes an application (or part of an application). The nodes in a feature diagram represent features and are referred to as *feature nodes* [KCH+90,Wit96,CE00]. The edge decorations characterize features and relationships as being of different types (see Figure 2.2):

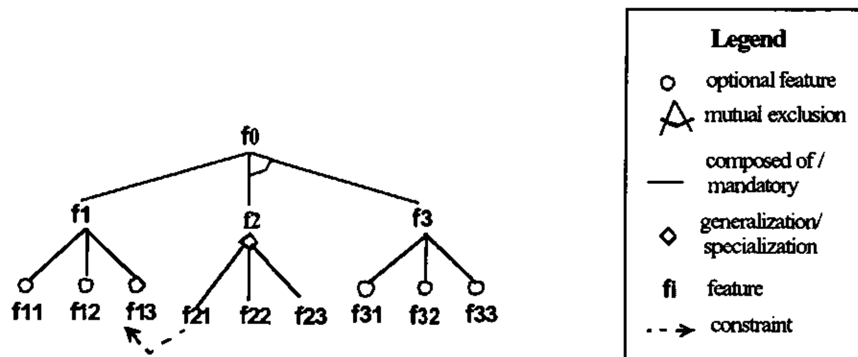

Figure 2.2 Feature diagram and elements

- *Optional features* represent features that may be disregarded from a feature model for particular configurations. An optional feature is represented with a circle at the end of the edge linking it to another feature.

1. A user may be a human user or another system with which applications in a domain typically interact.

- *Mandatory features* must be selected in all the configurations of the feature model. They represent the commonality among different model instances. A mandatory feature composed exclusively of optional features means that at least one of them has to be selected in all configurations (i.e., applications). No special edge decoration characterizes this type of feature.
- *Composed-of relationships* express features that are composed of several sub-features, following a decomposition/aggregation abstraction mechanism. This relationship is represented by drawing a line from the super-features to each of its sub-features. No edge decoration is required for this representation.
- *Generalization/specialization relationships* represent abstract concepts that can be concretized (i.e., specialized) in several ways. A generalization relationships is represented by a diamond at the generic's feature end. A line is drawn to each available specialization feature from the diamond. Generalizations are substitutable by one or more of their specialization features.
- *Constraint* relationships represent *requires* or *mutual exclusion* constraints. These semantic constraints are defined on operational features and variants, to give the model consistency. A dashed line may interconnect two features indicating that one requires the presence of the other in the model. Mutual exclusion is specified by an arc joining mutually exclusive features. Additionally, incompatibility in the choice of two features can be expressed as separate relationships with respect to the diagram, or using *requires features* or *excludes features* attributes (i.e., labels) in the relationships.

The notation used to specify feature models is exemplified by Figure 2.2 which shows that: f_0 is the higher level domain feature (which can be an application or subsystem). An application should always contain feature f_1 and one of f_2 or f_3 (but not both). Feature f_1 can have as sub-features zero, one, two, or three of

the features f_{11}, f_{12}, and f_{13}. Feature f_2 is specialized by features f_{21}, f_{22} and f_{23}; f_2 is a generic feature. Feature f_3 has a meaning similar to that of f_1. Finally, the presence of f_{21} in a feature model, requires the presence of f_{13} in the model (however, the presence of 13_6 doesn't require that f_{21} be present).

2.3.2 Domain Design: GenVoca

GenVoca [BO92] is a domain-independent model for designing application families. Applications are modeled as hierarchical compositions of layers. GenVoca layers can be considered functionally as representing either refinements or extensions. A *refinement* or *extension* adds data and operations to the input to produce the output; each layer contains a number of object classes and the layer below extends the layer above it by adding new classes, or adding new methods and attributes to existing classes [Sin96, Sma99].

Each GenVoca layer implements a *type*, represented by its interface (that is, the set of services it offers to its higher-level layer). Additionally, every layer (except the layer at the bottom) is dependent upon a lower level layer implementing a type. Each layer declares the type it implements (the *implemented type* or simply its type) and the type it requires (the *required type*) its lower-level layer to implement. Layers are implemented as software modules (generically known as *components* in GenVoca). A GenVoca component encapsulates a suite of interrelated functions, variables, and classes that work together as a unit to implement a particular feature [Tho98] (i.e., in GenVoca features are types, thus there may be several implementations of a feature). Different components can implement the same type (i.e., layer definition), thus becoming interchangeable.

GenVoca components can be represented using an UML component icon [BRJ98] with its type represented by an UML interface icon, as shown on Figure

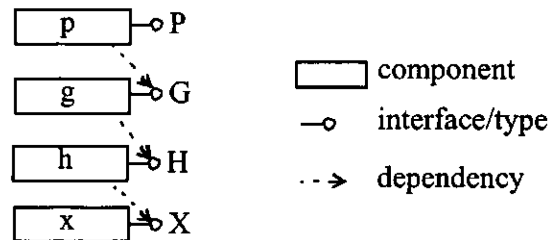

Figure 2.3 GenVoca components in UML

2.3. This representation shows that a GenVoca component is a particular implementation of a type. In Figure 2.3 upper-case letters represent types, and lower-case letters represent components (type implementations). The dotted lines in Figure 2.3 represent the dependency of a layer/component from the lower-level layer / component (type) that is required by a component. Note that dependency is on the type, which means that any component implementing that type can be used as an actual parameter.

The similarities and differences among members of a family are exposed by comparing the component compositions that define them. A number of generators for product lines have been based on the GenVoca model (Genesis [Bat88] a compositional generator for a database product line, P2 [Tho98] C-based transformational generator for a data structures product line, P++ [Sin96] a C++-based compositional generator for a data structures product line, and DiSTiL [SB97] a Java-based transformational generator of a data structures product line). An important result from GenVoca product line implementations was the realization that the number of fundamental domain abstractions is typically rather small, but many different implementations are possible for every abstraction [BSST93].

The concepts described can also be represented as shown on Figure 2.4. The type *R* has three implementations (components *a*, *b*, *c*), type *S* has three

$R = \{ a, b, c \}$

$S = \{ d(x{:}R), e(x{:}R), f(x{:}R) \}$

$T = \{ n(x{:}T), m(x{:}T), p, q(S) \}$

Figure 2.4 Types of components and implementations

(components *d, e, f*), and type *T* has four implementations (components *n, m, p, q*).

Each component of type *R* is distinct (i.e., it encapsulates its own algorithms, has its own unique performance characteristics, has its own unique memory footprint, etc.). All components of *R* implement the same type and thus are plug-compatible. The same holds for *S* and *T*. Note on Figure 2.4 that a parameter denotes the type of a required component.

An application is a named composition of components called a *composition equation*. Component composition is accomplished by instantiating component parameters. For example, consider the following three composition equations (components corresponding to Figure 2.4):

app1 = d(b);

app2 = d(a);

app3 = f(a);

Application *app1* composes components *d* with *b*, *app2* composes *d* with *a*, *app3* composes *f* with *a*. All three applications are composition equations of type *S* (because their outermost components implement type *S*). This means that app1, app2, and app3 are interchangeable implementations of *S*.

Components whose required type is the same as its implemented type are *symmetric*. Symmetric components can be composed in arbitrary ways. In type *T* of Figure 2.4, components *n* and *m* are symmetric whereas *p* and *q* are not. This

means that compositions *n(m(p))* and *m(n(p))* are possible. In general, the order in which symmetric components are composed matters.

Syntactic compatibility between components is easily checked by verifying that for each component the type implemented by its actual parameter corresponds with its required parameter type. Thus *app1* is syntactically valid, because *b*'s implemented type and *d*'s required type are both of type *R*.

Component *semantic* compatibility is more complicated. Note that some combinations of components may be syntactically but not semantically correct. That is, each pair of components in the application requires and implements compatible types, but the resulting algorithms may be invalid for some reason. To verify the semantic correctness of an application, each component must supply domain-specific information that describes the assumptions and constraints for using that component [BG97]. Such information can be handled by a generator to verify semantic correctness. Later in this chapter we describe how semantic verification is dealt with.

2.3.3 Translating Feature Models to GenVoca Models

Up to this point we have presented two modeling notations. Feature diagrams to describe domain entities of interest, and GenVoca for designing product lines. However, we haven't explained how GenVoca designs can be obtained from a feature diagram. In this section we describe the process to map feature diagrams to GenVoca designs.

Figure 2.5 shows the translation from a feature model to a GenVoca design. Figure 2.5(a) is a feature diagram in which feature *f2* is mandatory, feature *f3* is a generalization of features *f31* and *f32*, and feature *f4* is optional. Figure 2.5(b) is a hierarchical representation of a GenVoca design —Figure 2.5(c) and Figure 2.5(d) are instances of hierarchical systems obtained from the GenVoca

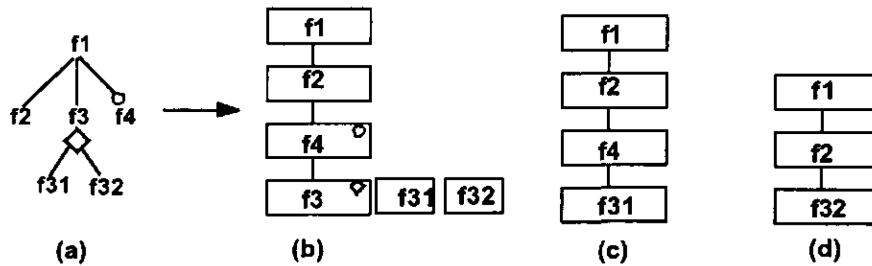

Figure 2.5 From feature models to GenVoca model

design described by Figure 2.5(b). Note that an annotation symbol in Figure 2.5(b) describe optional and generic components[2].

In transforming feature diagrams to design diagrams, one should acknowledge that a feature diagram shows how features are related in a structural way, the hierarchical diagram will show how features are related at execution time. Features can be abstracted into individual design entities. These design entities consist of groups of interacting classes (thus they can be interpreted as patterns in a design pattern approach [GHJV94], collaborations in a collaboration-oriented approach [VN96, SB98], role-models in role-based design [RG98], subjects in a subject-oriented approach [OH92], etc.). Each design entity is a GenVoca component (or more appropriately, a layer, because a single feature can have multiple implementations).

The root feature in a feature diagram will be the layer at the top in the hierarchy. How the other features map to the hierarchical model requires analyzing how features relate to one another at application execution time. For instance, let's say that for the feature model in Figure 2.5(a), feature *f1* may use the functionality (i.e., classes, methods, and attributes) from feature *f2*, and *f2* may use the function-

2. Hierarchical diagrams can be represented are a direct graphical representation of GenVoca compositions equations. For instance, the diagram in Figure 2.5(c) represents the composition f1(f2(f4(f31))).

ality provided by *f4* (when present, since *f4* is optional), and finally that *f3* provides the lower-level functionality for the application. This analysis produces Figure 2.5(b). Note that different feature interactions produce different hierarchical models. However, application families are characterized by their features and their interactions, thus application families are in fact described by hierarchical models obtained from feature models.

Note that Figure 2.5(b) doesn't include features *f31* and *f32*. The reason is that both are specializations of feature *f3*, and they will be used when the specific characteristics they define are needed. Note also in Figure 2.5(b) that *f3* is never used, but instead one of its specializations, *f31* or *f32*. Figure 2.5(c) shows an instance of the composition for an application implementing features *f1*, *f2*, *f4*, and *f31*; another application which implements features *f1*, *f2*, and *f32* is depicted by Figure 2.5(d).

The annotations used in Figure 2.5(b) use symbols similar to that used in a feature diagram to emphasize optionality (a circle) and selection (a diamond). These conventions help to keep consistency in the interpretation of diagrams used to construct product line infrastructures.

Feature diagrams have been broadly used in domain modeling and different mappings to design entities are suggested. For instance, feature diagrams can be translated to reference architectures [KKL+98] and object diagrams [LKCC00, GFD98, GFD97]. Our selection of translating feature diagrams to GenVoca is based on the fact that GenVoca diagrams represent product-line architectures and thus can be used to implement product lines[Bat98].

2.3.4 Domain Implementation: Wizlets as Components

An implementation approach for product lines should meet several goals. Firstly, it must clearly reflect a product line architecture. Secondly, the implementation

approach must express domain commonality and variability, thus commonality can be exploited in different applications and components can be adapted to meet domain variability.

Another aspect of architecture implementation is domain evolution [Bos99]. The implementation approach should provide support to evolve as domains evolve [ML98, RJ97], because by clearly reflecting the product line architecture, the implementation simplify evolution [Par79, VN96b, Sma99].

In particular, an approach for implementing GenVoca models should reflect hierarchical component stacking. It should allow module composition in a linear stacking, and should support component swapping inherent in GenVoca models. An implementation approach that has shown to be adequate for implementing GenVoca models is that proposed by Smaragdakis and Batory [SB98], which has its roots on the work by VanHilst [VN96a,VN96b], and Bracha and Cook [BC90]. The basic technique implements every component as a *mixin class*. Mixins (also commonly known as abstract subclasses [BC90]) represent a mechanism for specifying classes that will eventually inherit from a superclass. This superclass, however, is not specified at the site of the mixin's definition. Thus a single mixin can be instantiated with different superclasses yielding widely varying classes. This property of mixins makes them appropriate for defining uniform incremental extensions for a multitude of superclasses. When a mixin is instantiated with one of these superclass, it produces a class extended with the additional behavior.

A *wizlet* is a mixin encapsulating a group of related classes, instead of a single class[3]. Wizlets may use programming language extensions in their implementation, requiring it to be preprocessed before compiled into applications. A wizlet extends several classes in its super-wizlet. These ideas can be depicted graphically as in Figure2.6 (a conceptual way of representing static component

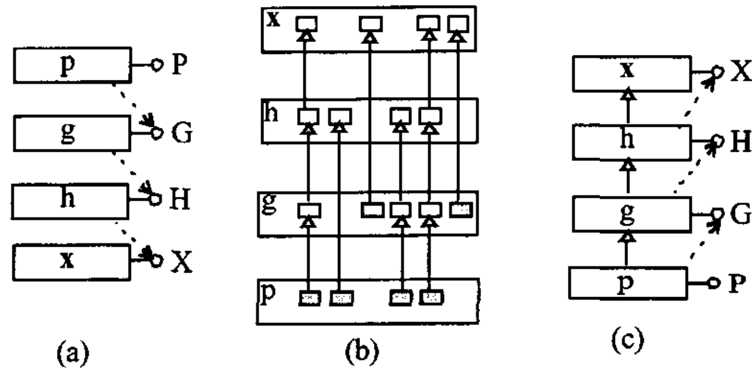

Figure 2.6: Component compositions and inheritance.

compositions [Bat98]). First, Figure 2.6a is a reproduction of the component composition presented on Figure 2.3. Figure 2.6b shows an inheritance hierarchy equivalent to composition on Figure 2.6a. Note that class *x* is the top class in the hierarchy, which corresponds to the lower-level component in Figure 2.6a. The following class in the hierarchy (in Figure 2.6c) is *h*, which corresponds to the layer on top of layer *x* (in Figure 2.6a). Proceeding in that way we reach the top level of the layer hierarchy and the bottom level of the inheritance hierarchy. While this notation and correspondence seems strange, it is due to historical reasons. That is, in general the ordering of a class hierarchy is the inverse of the order of a layer hierarchy.

Note in Figure 2.6b how every class/wizlet extends its super class by adding functionality to an inner class in its superclass, or adding several new inner classes. Inner classes that are not extended by a lower level class are *final classes.*

3. For those familiar with *mixin layers* as proposed by Smaragdakis and Batory [SB98], the difference between mixin layers and wizlets may not be apparent. The main difference is that mixin layers are limited to be implemented using existing programming language mechanisms for class parameterization. Wizlets may use extensions to programming languages, thus require preprocessing before compiled (C++ already supports class parameterization). Wizlets are designed and implemented to be integrated by a generator tool in constructing applications.

Thus, when wizlets are composed, a forest of inheritance hierarchies is created. Adding a new wizlet (stacking a new layer) causes the forest to get progressively broader and deeper. It is through the use of inheritance that new operations/methods can be added to multiple application classes merely by plugging in a component [Edw95].

Figure 2.6c unifies the representation for components as types and classes, that is, every wizlet implements a type and is meant to inherit a super-wizlet implementing a type. The actual super-wizlet is not known at wizlet implementation time, so its specified as a type.

Wizlets can be easily implemented in C++ using parameterized inheritance. In this case, a wizlet is a parameterized class with the parameter becoming its superclass[4]. Using C++ syntax we can write a wizlet as:

```
template class <class WizletSuper>
class Wizlet : public WizletSuper
{
   public:
   class InnerClass1:
      public WizletSuper::InnerClass1
        { ... };
    class InnerClass2:
      public WizletSuper::InnerClass2
        { ... };
    ...
};
```

Here `Wizlet` is the abstract subclass being defined, and `WizletSuper` is a parameter defining `Wizlet`'s superclass.

4. C++ was chosen here to explain wizlet implementation because its direct support for class parameterization. Wizlet implementation in programming languages not supporting class parameterization is described in Chapter 3 - Chapter 5.

Wizlets are composed by instantiating one wizlet with another as its parameter. The wizlets are then linked as a parent-child pair in the inheritance hierarchy. C++ provides a direct mechanism for expressing wizlet compositions, for instance:

```
typedef w1< w2< ... < wn > ... > > C                                  (2.1)
```

is a template composition where `w1`, `w2`, ..., `wf` are wizlets, "<...>" is the C++ operator for template instantiation, and **C** is the name given to the class that is produced by this composition[5].

Composition (2.1) has a direct counterpart in GenVoca, (2.1) has the exact form used in GenVoca for composition equations, except for syntax ("(..)" replaces "<...>"). Thus, (2.1) corresponds to equation (2.2):

```
C = w1 (w2 ( ... ( wn ) ... )                                          (2.2)
```

where `w1`,... are wizlets representing GenVoca components.

This section described how wizlets (GenVoca components) are implemented by parameterized classes, showing implementation code in C++, because C++ directly supports class parameterization. We demonstrate in following chapters how other object-oriented programming languages supporting inner class encapsulation, but not necessarily class parameterization can be extended to support class-like parameterization. As examples, subsequent chapters describe how Object-Oriented Pascal and Java can be extended to implement GenVoca components.

---

5. Note that other programming languages not supporting template-like parameterization do not provide a template instantiation operator similar to that of C++. Besides designing a mechanism for wizlet parameterization, it may be necessary to analyze how to implement wizlets in other programming languages.

2.3.5 Composition Validation

Having components correctly implemented doesn't imply that all their combinations define valid applications. A fundamental problem for all component-based software development technologies is whether component compositions are consistent/valid [BG97, DP98, Szy98].

The implementation of automatic composition validation of hierarchical component compositions requires representing the properties that components add and need from their environment using configuration predicates. *Configuration predicates* describe a component to its environment and prescribe properties that should be met for that component to participate in a composition.

Configuration predicates can be of three types: assertions, boolean or numeric.

- **Numeric**: this type of predicate can provide information concerning the component algorithmic implementation such as its algorithmic complexity or memory requirements. Other predicates may define actual parameters declaring component properties such as maximum number of elements allowed in a data structure; values distinguishing the component, such as the numbers of wheels in a vehicle being simulated by the resulting application, number of axes along which a cutting tool can move in the machine tool that will be controlled by the resulting application, etc.
- **Boolean**: boolean expressions are represented by logic predicates and operators. Predicates in a boolean expression represent constraints that need to be valid for using components in a composition.

- **Assertions**: assert an important property of the component, such as declaring the presence of the component in a composition. Assertions are similar to numeric predicates but the assessed values are boolean in nature. Note that assertions are a special kind of boolean expression whose value is set to true when a component is found in a composition.

Configuration predicates can be additionally characterized as being of four types: pre-conditions, post-conditions, post-restrictions and pre-restrictions; such classification depends on where properties defined by predicates are used to verify composition validity. To discuss configuration predicate types, suppose the general case in which **k** is a component:

- **Post-conditions**: Post-conditions are properties of **k** that are exported to layers beneath **k** in a component composition.
- **Pre-condition**: Pre-conditions define properties that must hold for components to work properly; they test the cumulative post-conditions of layers that lie above **k** in a composition. It is common to have components whose pre-conditions and obligations are not satisfied locally (i.e., by components that are not adjacent in a composition equation).
- **Post-restrictions**: are properties of **k** that are exported to layers above **k** in a composition.
- **Pre-restrictions**: (also known as obligations) are pre-conditions for instantiating layer parameters; they test the cumulative post-restrictions of layers that lie beneath **k** in a composition.

It is important to remark that pre-conditions and obligations of a component **k** can be satisfied "at a distance", that is, by components that either lie (far) beneath **k** or (far) above **k** in a composition equation. Moreover, the properties exported by **k** to "higher" layers are generally not the same properties that are exported to "lower" layers.

Two commonly used boolean predicate types are (note that these can be further characterized as being of one of the four types described):

- **Requires**: specifies that a component of particular type and/or supporting certain specific characteristics is required below or on top of the current component being verified.
- **Prohibits**: the presence of a particular component type or properties defined by other components are not allowed in the composition.

Given configuration equations representing component compositions, configuration validation involves:

- A top-down propagation of post-conditions, and the testing of component preconditions, and
- A bottom-up propagation of post-restrictions and the testing of parameter prerestrictions.

To implement configuration validation, configuration equations can be encoded in different ways in programming languages. To propagate configuration state up and down the component hierarchy, configuration state can be maintained in a separate *state component.* Every time the validity of a component in a composition needs to be verified, the environment information discovered up to the moment can be extracted from that state component. For instance, in an object-oriented programming language, an object may store environmental information as attribute values of a composition. Every object implementing the validation can receive such state object as a parameter, and retrieve and store state attributes which can be of interest to other components.

Figure 2.7 shows a partial implementation of composition validation. Validation proceeds in both directions (top to bottom and bottom to top), storing and retrieving configuration information in the State object.

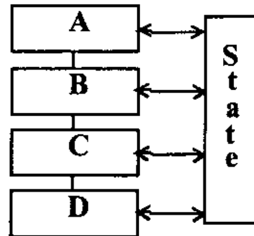

Figure 2.7 Composition verification

Inside every component, the validation code is the code that asserts the information of the component, or the code that verifies or prevents the presence of other components. The following code fragments exemplify each of these types, in the example prefix `state` represents an instance of the State component, and `component` would be represented by the component name whose predicates are being defined:

Assertion: For components that need to assert its presence or properties to other components (for instance, the following code, *component* can be substituted by `StepMotor` if a stepper motor is being used in a composition):

```
state.componentSet = true;
state.someProperty = 2;
```

Requires: in the code that follows, the word *requiredComponent* would be replaced by a corresponding component name[6]:

```
if( ! state.requiredComponentSet)
    error("Component needs a requiredComponent in the composition")

if( state.numAxis !=  this->numAxis)
    error("Number of axis in Component doesn't match the defined
          number of axis")
```

6. Note that requires can be implemented as a pre-condition or a pre-restriction, according to the layered representation of a feature diagram.

Prohibits: In the following code, the identifier *prohibitedComponent* would be replaced by a component name:

```
if( state.prohibitedComponentSet)
    error("prohibitedComponent can't be used in combination with
           Component")
```

The previous code fragments show how explanation code is included together with verification code. Such information can be extremely useful at application configuration time, for producing well defined applications.

Depending on how components are planned to be composed to build applications, and facilities in the development environment, validation expressions (i.e., configuration predicates) can be implemented together with application code inside a single component, or the implementation be split in one *application component* (containing the code a component adds to the application) and a counterpart *verification component* (containing the code implementing the configuration predicates for that component). We can justify the separation of code based on how both parts are used. Verification code is executed only once in the life span of an application (to verify the validity of a component composition specifying the application), however the application code will be executed (potentially) many times. Note that such separation of implementation is suggested for cases in which components are to be integrated in a compositional way, instead of in a transformational way. Generative components may contain both the generative code and the validation code inside a single component, because both parts are executed at generation time.

Our implementation approach of wizlets as parameterized components is compositional in nature, parameters have to be concretized at application composition. This observation, and our stated purpose that wizlets can be implemented using different programming languages, guided us to chose a separate implementa-

tion for wizlets: a functional (application) part, and a validation part. (or stated another way, the verification code is evaluated statically at configuration time, while application code is evaluated dynamically at application runtime).

2.4 Configuration Wizards

The synergistic combination of feature models with GenVoca hierarchical product-line architectures and parametrized wizlets sets the foundation for implementing product lines. However, there are limitations for the broad applicability of the implementation approach described earlier. The module/class parameterization mechanism is fundamental for implementing components as parameterized classes. However, mainstream object-oriented programming languages in use today do not provide support for class parameterization[7].

In general, compilers offer limited support needed at product-line application-engineering time (i.e., composition time). In the presence of a syntax error, it would be convenient that compilers provide developers with hints on what is causing errors. A similar situation occurs for the more sophisticated need of composition verification (i.e. validate that components in the composition precisely define an application). Although it is possible to implement limited composition consistency validation by type declarations in the class hierarchies [Sma98], industrial level application systems require more complex configuration predicates than can be implemented in one direction (top to bottom).

Unfortunately, even programming languages supporting parameterization capabilities offer limited support for composition verification (syntactic and semantic). For instance, C++ compilers produce large and intimidating messages

7. One exception is of course C++, and that is why we frequently cite it as example for explaining most of our ideas.

for syntactic errors in the presence of erroneous template compositions[8]. Component compositions producing syntax error messages are difficult to debug. If a developer is unfamiliar with the nuts and bolts of component implementation for the domain at hand, it is a monumental work to find error sources. Although configuration predicates provide the developer with support for dealing with semantically incorrect compositions, it is a difficult endeavor to manually verify that compositions don't violate any predicate.

Syntax errors are unavoidable, but they occur at component development time (i.e., component developers will solve them when implementing components). It is at application engineering time (i.e., when constructing members of the application family) that semantic verification will be of great help for the application developer.

We argue that application engineering in a product-line environment can be simplified by the use of configuration wizards. Configuration wizards are software tools containing wizlets as components, predicates prescribing valid compositions, a specification interface, and a generator.

Figure 2.9 depicts our proposed generic architecture for a configuration wizard. Figure shows how the two parts in which we divided wizlets, are stored in two different containers: a *wizlet repository* storing the application code part, and the *domain knowledge* repository containing the verification part of wizlets (i.e., the topological information of the product-line architecture and configuration information prescribing valid wizlet compositions). A *specification interface* offers the application developer editing facilities to specify the application at hand[9]. The *generator* receives application specifications from the developer, trans-

8. One example is the following message indicating a missing variable declaration in a component: <<<include example>>>

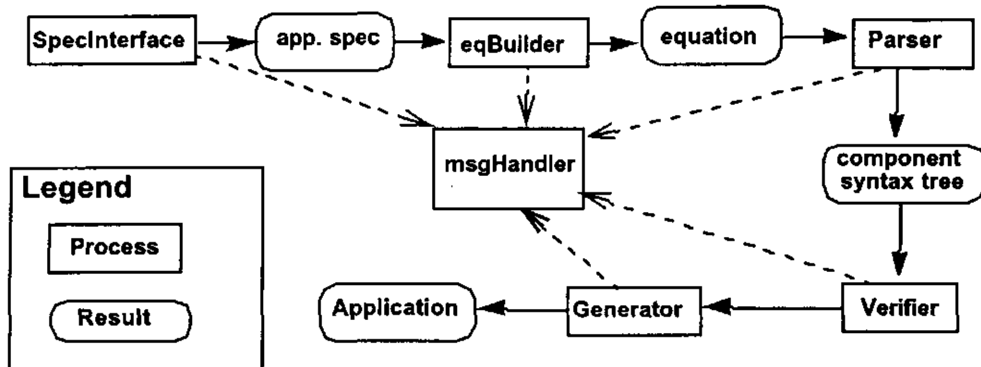

Figure 2.8: Configuration wizard process model

lates them to concise representations (i.e., layer stacking), verifies their semantic correctness and generates the application by adapting wizlets implementing specified requirements.

Process models helps to describe how a sequence of steps (i.e., transformations) produce a final result. A process model describing involved steps and intermediate results between steps in a configuration wizard, is shown in Figure 2.8. The process model Figure 2.8 shows that a configuration interface is used for requirement specification, the result is an application specification, consisting in the required features specified by a developer. SpecInterface allows specifying only non-mandatory features. The requirements set is the input to the eqBuilder module, which translates specifications to equivalent composition equations. The composition equation produced by eqBuilder includes elements representing mandatory domain features inserted at appropriate positions (as defined by the product-line architecture).

9. Most of the details the developer has to provide consist in domain features the intended application should incorporate, and parameters to concretize these features to particular requirements (e.g., memory sizes, storage requirements, special date formats, etc.)

The next module in the process is Parser, whose task is to build a syntax tree from the composition equation[10]. After that, a Verifier module checks (by evaluating configuration predicates) that the syntax tree corresponds to a valid wizlet composition. Finally, the Generator module parses the syntax tree to instantiate wizlet parameters and produce any extra information necessary thus the generated application can be compiled.

Along the configuration and generation process, modules are informing the user/developer of the progression by sending messages to a message handler.

Specification interfaces could be designed very differently for different domains and product lines. What we affirm is that specification interfaces can facilitate requirement specification by explicitly presenting domain features (types) and their specializations (different implementations). For instance, Figure 2.10 shows a possible specification interface for a product-line of vehicle simulators[11]. In the example, specializations are grouped using different grouping facili-

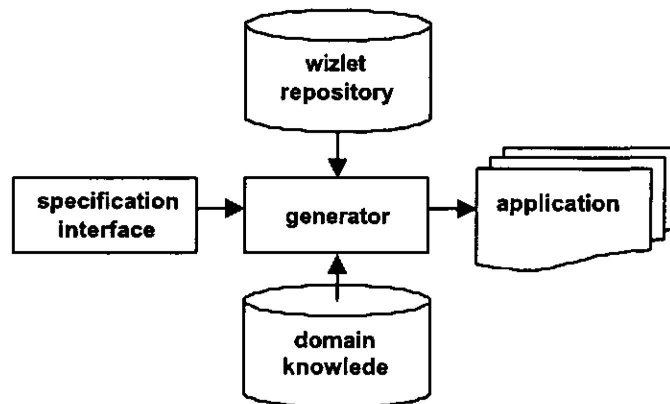

Figure 2.9: Configuration wizard's architecture.

10.Complex domains may have layer representations consisting in branches thus needing parse trees for their analysis [BCGS95].

11.The example is described in detail in Chapter 3, here we include it to help us explain the general ideas on implementing product-lines as configuration wizards.

ties (as available in the development environment). For instance, movement types are in a group called "Movement type" whose choices are "Normal" and "Differential"; "Controller type" grouping include specializations "Intuitive" and "Fuzzy"; "Model" defines a list of vehicle types the developer can choose from; finally, there is a single "Option" to specify if the vehicle is towing a "Trailer" or not.

The specification interface includes a visualization area which displays all messages informing the user/developer about the process progression. Several buttons in the interface allow the user to initiate different actions. For instance, a "Generate" button can be pressed after a set of requirements has been specified; "Generate" button starts the process shown in Figure 2.10. A button labeled "Compile" allows compiling the application just generated. After the application is com-

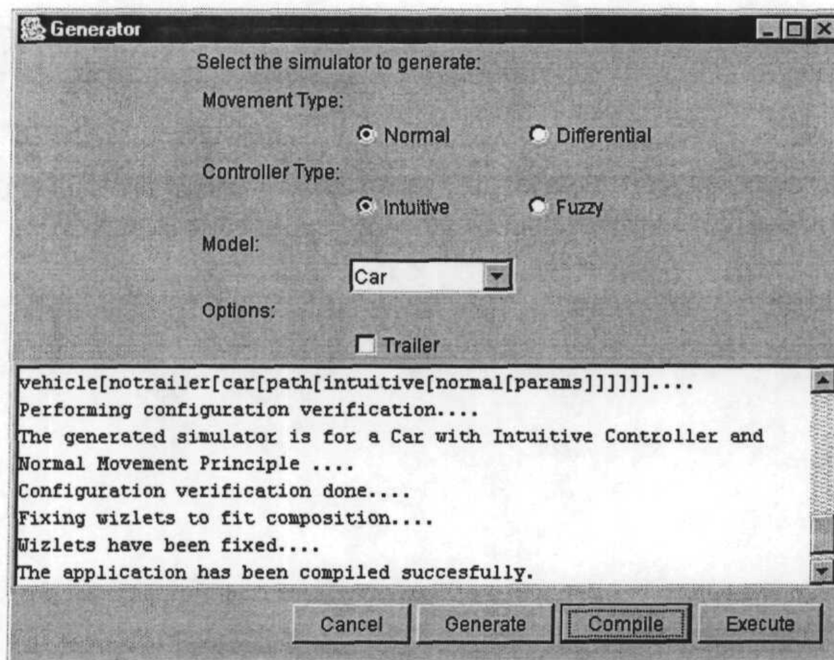

Figure 2.10: Specification interface for vehicle simulators.

piled, it can be executed directly from the configuration wizard using the "Execute" button.

By integrating a suite of validation and generator tools inside configuration wizards, we substantially enhance the individual capabilities and effectiveness of these tools. Incorporating domain knowledge in the way of configuration predicates may offer expert guidance to application developers so that design blunders can be avoided. With the assistance of configuration wizards, non-domain experts would consistently produce high-quality designs (i.e., configuration wizards can critique specifications so designs can be improved).

2.5 Recap

This chapter presented the general concepts related to software product-line engineering and proposed a method based on the consistent use of feature modeling for domain modeling, GenVoca as the architectural framework to describe product-lines as hierarchical organizations of plug compatible components called wizlets, a systematic approach for wizlet implementation using module parameterization (classes for C++ in particular), and finally, explained our thesis that a configuration wizard can be used to generate applications in product families.

We proposed that software product lines can be implemented as configuration wizards which:

- Free application engineers from manually building composition equations (i.e., component compositions). Wizards help to construct composition equations from input specifications using a product-line architecture.
- Assist application engineers in specifying correct applications (compositions) by displaying inconsistencies in a friendly manner.

- Free application engineers from knowing programming language composition syntax to construct compositions.

The next chapters demonstrate how configuration wizards can help to construct software product-lines in different domains using different implementation technologies. Three different product lines will be shown: a simulator for autonomous vehicles, a computer numerical control system, and a general ledger for credit unions. Each product line will be implemented using a different programming language, thus helping us to identify programming language extensions necessary for implementing configuration wizards for product lines in different environments.

Our goal is to demonstrate that configuration tools with limited generative capabilities, supported by programming languages providing encapsulation and inheritance mechanisms, are enough to implement product lines.

The following chapter describes the use of our approach to construct a product line using Java as programming language, and our proposed extensions for implementing class parameterization in Java.

Chapter 3

Vehicle Simulators Product-Line

We present our first example of a configuration wizard implementing a software product-line in this chapter. The example consists of a family of software simulators for autonomous vehicles having two, three, and four wheels. The example is interesting in that it shows in detail the application of our proposed method for product-line engineering. The configuration wizard and parameterized components are implemented using Java as programming language. As Java does not directly supports class parameterization, we extend it to allow wizlets (necessary for a configuration wizard) to be defined as parameterized classes.

3.1 Autonomous vehicle simulation

Software simulators are important when, for any reason, it is difficult to have real prototypes for experimentation (e.g., physically constructing simulated devices). There may be various reasons why constructing devices is impractical. One reason is economy; constructing physical prototypes is very expensive. Other reason is time; building hardware prototypes can take too long. Another reason is danger; when dangerous materials have to be handled injuries may result, etc. Simulators offer the advantage of being able to run different experiments by adjusting parameters and executing the software prototype, without the hassles involved in building real prototypes. The vehicle simulators family we describe in this chapter was

constructed due to a combination of the aforementioned reasons. Having components of the family implemented in software would simplify the construction of different simulators by composing components that implement the required features.

The Research Laboratory of the Artificial Intelligence Center at ITESM was interested in analyzing different algorithms for autonomous vehicle control, to experiment how vehicles move to reach a predefined target point from an arbitrary initial position and direction. It was expensive and time consuming to perform all modifications and implement the corresponding algorithms in real vehicles to analyze their behavior. One problem is that vehicles with different shapes and sizes behave differently for similar control algorithms. For instance, two-wheels vehicles behave very differently from four-wheels vehicles, thus the necessity of having different movement algorithms for different types of vehicles.

The goal of our simulators product-line is to allow researchers to specify simulators incorporating a desired set of features for a vehicle and to build the corresponding simulator from that specification. Once constructed, the simulator can be used in analyzing the path followed by the corresponding vehicle for a particular algorithm. Among the features a researcher may want to specify are: the vehicle type (e.g., motorcycle or car), the vehicle tows a trailer, use a specific movement algorithm, etc. Simulators should provide researchers with user-friendly interfaces to allow them perform different specifications within a given simulator (e.g., initial orientation, path to be followed, speed, etc.).

The main requirements of the simulator family were that the product-line infrastructure should simplify the construction of vehicle simulators for different platforms (i.e., the programming language needed to be platform independent). As performance was not a critical factor, the main interest was in analyzing movement

traces, not the time vehicles take to traverse predefined paths. Given these requirements, Java was selected as programming language.

3.2 Static Parameterization in Java

The analysis presented in the previous section justifies the selection of the Java programming language as a good candidate to satisfy simulator family requirements. However, our approach for wizlet implementation presented in Section 2.3.4 is restricted to programming languages supporting class-like parameterization mechanisms (e.g., C++ templates), and Java doesn't directly supports class parameterization. Still, a close analysis to wizlet parameterization and Java capabilities reveals that the only extra support necessitated in Java is to allow classes as parameters[1]. In the following paragraphs we show how Java can be extended to support parameterization for class composition by inheritance, when super classes are unknown at class implementation time.

There are several proposals for extending the Java programming language with template-like parameterization [BOSW98, AFM97, MBL97]. However, to keep consistency in the way we express component compositions across programming languages, we preferred an ad-hoc and simple extension to Java, instead of using one of these extension proposals. Our proposed extension has similar syntax to that of GJ [BOSW98], however, the similarity ends there, since we don't offer full support for generic types, as GJ does.

We extend Java syntax to implement wizlets as parameterized classes, thus the syntax in Java is similar to that explained in the previous chapter for C++. In

1. Java interfaces can be extended to implement parameterization [BOSW98]; however, we extend Java to support class parameterization, to be consistent with how C++ implements class parameterization.

the extended syntax, Java classes can be parameterized by declaring its super wizlet as a parameter, as follows[2]:

```
class Wizlet extends <<WizletSuper>> {
  public class Inner1 extends <<WizletSuper>>.Inner1
  { ... }
  public class Inner2 extends <<WizletSuper>>.Inner2
  { ... }
    ...
}
```

Similarly to what we did in C++[3], class `Wizlet` is an abstract subclass, and `WizletSuper` is a parameter defining `Wizlet`'s superclass. Java's class encapsulation is similar to C++'s, thus the similarities between C++ templates and our proposed Java extension. In Java, wizlets encapsulate several inner classes, and Java's inner classes are inherited, just like they are in C++. Thus Java's wizlet implementation is performed using parameterized inheritance and nested classes, just like we can do in C++. Let's see how this mechanism works in a simple wizlet implementation example.

Suppose type declarations in Figure 3.1(a) describing type *Device* whose specialization components are *Window* and *Printer*, and type *R* whose only compo-

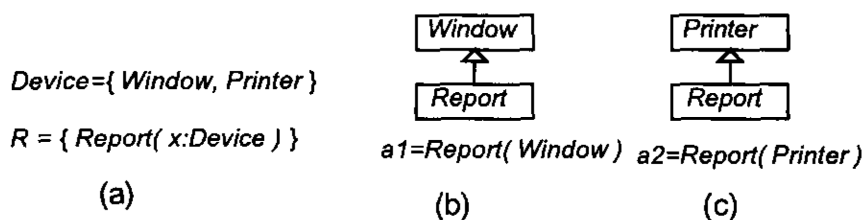

Figure 3.1: Type declarations and equations.

2. Note our use of <<..>> to represent class parameters, which is not native Java syntax.
3. We strongly suggest reading Section 2.3.4 to compare C++ syntax with our proposed Java extension.

nent is *Report*. From declarations in Figure 3.1(a), one can construct two compositions. One is shown in Figure 3.1(b) describing an application where *Report* sends its output to *Window*. The other component composition is shown in Figure 3.1(c) which describes an application where *Report* component sends its output to a *Printer* component. In all cases that follows, we assume that wizlet components contain two inner classes Inner1 and Inner2, which they extend. The following is a partial Java implementation of the *Report* component using our proposed extension to Java:

```
class Report extends <<WizletSuper>> {
  public class Inner1 extends <<WizletSuper>>.Inner1
  { ... }
  public class Inner2 extends <<WizletSuper>>.Inner2
  { ... }
...
}
```

For simplicity, we maintained the internal definition of *Report* similar to our implementation code shown earlier for a parameterized Java class, changing the wizlet name only (i.e., from *Wizlet* to *Report*).

To show how proposed Java's extension mechanism works, lets implement *Window* and *Printer* components. The code for *Window* is:

```
class Window {
  public class Inner1 { ... }
  public class Inner2 { ... }
  ...
}
```

where *Inner1* and *Inner2* represent classes internal to *Window*.

Note that *Window* implementation is raw Java code. This is because *Window* is a base class, thus no parameterization is needed. Similarly, we can implement *Printer* as follows:

```
class Printer {
  public class Inner1 { ... }
  public class Inner2 { ... }
  ...
}
```

From these implementations, we can specify composition equations defining systems that display a report inside a window (see Figure 3.1(b)) on the monitor screen, or that print the report on paper (see Figure 3.1(c)). Note, however, that the implementation of *Report* cannot be directly compiled in Java (i.e. the wizlet from which *Report* inherits has to be concrete in the code implementing *Report*). We can adapt parameterized wizlets using composition equations. From Figure 3.1(b) we substitute[4] the parameter *WizletSuper* with *Window* (as declared by composition equation in Figure 3.1(b)) in the extended Java code. The resulting code for *Report* is:

```
class Report extends Window {
  public class Inner1 extends Window.Inner1
  { ... }
  public class Inner2 extends Window.Inner2
  { ... }
  ...
}
```

Similarly, the concrete code obtained from Figure 3.1(c) using corresponding wizlets (i.e., *Report* and *Printer*) is:

```
class Report extends Printer {
  public class Inner1 extends Printer.Inner1
  { ... }
  public class Inner2 extends Printer.Inner2
```

4. Bracha et al. call wizlet's parent class an *erasure*. The erasure of a parametric type is obtained by deleting the parameter (*WizletSuper* erases to *Window* or *Printer*), the erasure of a non-parametric type is the type itself (so *Window* erases to *Window*, and *Printer* erases to *Printer*) [BOSW98]. Composition equations help to obtain the erasures of every wizlet in a composition, thus wizlets can be translated to Java classes.

```
    { ... }
    ...
}
```

These examples show how, even though Java doesn't offer compiler support to construct component compositions (similar to C++'s template expressions), we still are able to manually adapt components to fit particular compositions. Using this approach, a simple text editor can be used to adapt wizlets, thus they can be compiled into applications. Obviously such manual approach is undesirable and error prone (there is always the risk that incorrect substitutions are made, such as characters deleted by mistake, thus semantically or syntactically incorrect compositions are created).

A better approach is to make use of information about how components need to be adapted (such information is obtained from a wizlet composition expression specifying an application) so a preprocessor can parse equations and perform necessary adaptations to each wizlet. The resulting pre-processed components can now be compiled into an application by the standard Java compiler. Because pre-processing only makes sense after we can guarantee a semantically correct composition expression, configuration predicate checking and wizlet pre-processing should be integrated into a single tool. To operate, such tool needs as input the composition specification (i.e., a composition equation) for an application, and produce the Java source code for that application.

As discussed in section 2.4, integrating pre-processors and configuration-predicate checking as generators produces tools for building product lines. In the following sections, we use the proposed Java extension mechanism to implement a configuration wizard for autonomous vehicle simulators.

3.3 Domain Model for Autonomous Vehicles

As described earlier in this chapter, the domain includes applications to simulate the operation of vehicles. *Vehicles* can be of different types[5] (e.g., Car, Two-wheels motorcycle, Three-wheels motorcycle, Tank, etc.), implement different *movement principle* (normal or differential, which are explained later), and different *control algorithms* (intuitive and fuzzy, explained later). For performing a simulation, users provide initial values for parameters describing initial conditions for a given vehicle (e.g., direction the vehicle is facing, constant speed, initial position, etc.), and define a path the vehicle should follow (such path may consist in a single target position or a sequence of points).

The common operations that vehicle simulators need to implement are:

- *Put vehicle.* A given coordinate point defines vehicle's initial position. An angle states the direction a vehicle is facing.
- *Move to point.* Move towards a predefined point (performing the necessary direction adjustments if destination point is not in front of vehicle),
- *Follow a path.* A path is defined by a set of points. The vehicle must follow the trajectory defined by the path.

The way vehicles operate is determined by their *kinematic models.* The following are canonical kinematic models for different types of vehicles we are interested in simulating.

3.3.1 Kinematic Model of a Car

The kinematic model typical of a car (Figure 3.2) is one in which back wheels are fixed to an axis and front wheels can be turned right or left [Lat93, DJ00].

Nomenclature used in Figure 3.2 is as follows:

5. Italic words represent features in vehicles' domain.

θ = Car's angle from the horizontal axis in the plane.

ϕ = Tires's angle with respect to car axis (car inclination).

L = Distance between front and rear wheels (length).

A = Distance separating wheels in a given car's axis (width).

R = Current turning radius.

A car's *reference point* is the center of rear axis, it represents the car's position. A car's *movement principle* depends on where locomotion power is applied. If locomotion power is applied at the rear wheels (called *differential principle*), the velocity of the wheels need to be controlled. If locomotion power is applied at the front wheels (called *normal principle*), rear wheels' velocity is irrelevant.

A car's kinematic model can be translated to an equivalent model for a two-wheel motorcycle. For a motorcycle, we can imagine both wheels in a given axis (frontal or rear) collapsing into a single wheel at the middle point in that axis. In a two-wheel motorcycle, drive control is in the front wheel, while rear wheel cannot turn. Thus we don't need any special consideration of how a two-wheel motorcycle should be controlled, the only difference is that A (distance between wheels in the same axis) is equal to zero.

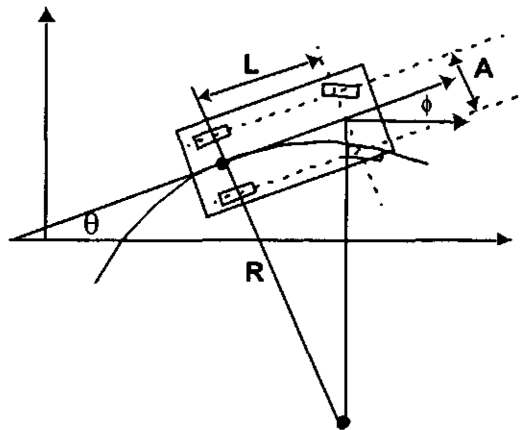

Figure 3.2: Kinematic model for a car.

3.3.2 Kinematic Model of a Tank

The other vehicle type is a tank, in which wheels in an axis are controlled individually. For a tank, wheels in the same axis can be applied different speeds (thus tanks implement a differential movement), but both wheels in the same side (front and rear) have the same speed. Figure 3.3 shows graphically a tank's kinematic model.

Nomenclature used in Figure 3.3 is [Lat93]:

θ = Vehicle's angle with respect to horizontal axis.

v1 = Velocity of wheels in the left side.

v2 = Velocity of wheels in the right side.

L = Length of a wheel.

A = Separation between wheels in the same axis.

R = Radius of the current turn.

Note that there is a difference on how cars and tanks move. Tank's reference point is the middle point between front and rear wheels (which is L/2 in Figure 3.3), while a car's reference point is the center of rear axis. For simulation

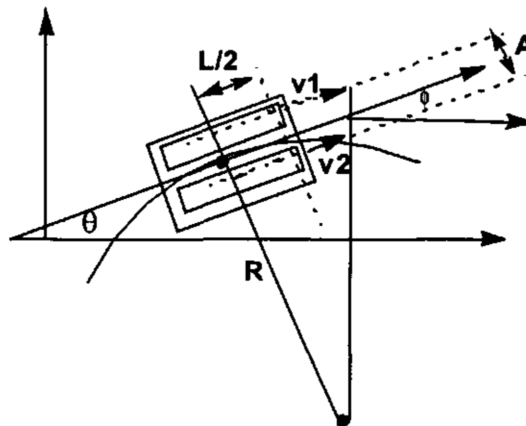

Figure 3.3: Kinematic model for a tank.

purposes, it is important to consider that both cars and tanks support differential movement principle.

3.3.3 Canonical forms of vehicle control

As explained before, we're interested in controlling vehicles to move from their current position and direction to a given target position or follow a path defined by a set of points. Canonical forms of vehicle control to behave in the desired way can be generalized. Consider the model in Figure 3.4; to guide a vehicle to move following a particular trajectory, the movement direction can be adjusted by determining the difference between vehicle's inclination angle θ, and target position's angle, relative to θ. This difference is known as the *error* from the target direction. At each step the control algorithm's goal is to reduce the error by adjusting turning angle ϕ. It should be noted that each vehicle type may have a different maximum turning angle. It is also worth noting that in a differential principle, the angle is produced by adjusting the wheels' speed; in a normal movement principle, the turning angle is adjusted (incremented or decremented) by a fixed value without changing wheels' speed.

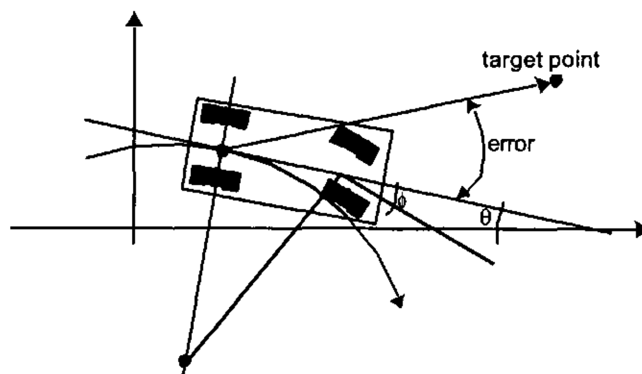

Figure 3.4: Trajectory control.

Such type of control algorithm is called *intuitive* (i.e., the error is reduced by adjusting turning thus final direction is eventually reached, from there, vehicles can move straight to their target point). Different controlling algorithms exit [Lat93], which proceed differently to guide vehicles. For instance, a fuzzy control algorithm use more complex techniques for vehicle guidance. For fast turning, a fuzzy algorithm can simultaneously change the speed of both wheels (increasing one and decreasing the speed in the other wheel). This produces a faster turning thus the vehicle will face in the direction of the target point sooner.

3.3.4 Feature Model

The information collected up to this point for vehicle simulation is summarized in a domain dictionary, shown in TABLE 1.

x,y	Cartesian coordinates representing vehicle position (middle point of rear axis in a car, middle point in a tank, etc., also known as the reference point)
theta(θ)	Vehicle orientation with respect to horizontal axis
phi(φ)	Vehicle turning angle with respect to X axis.
radius	Vehicle's turing radius
trajectory	Set of points defining a path
kinematic model	Model describing the movement of a vehicle type. Each vehicle type has its own kinematic model.
normal drive	The locomotive power is applied at the front wheels.
differential drive	The locomotive power is applied at the rear wheels.
vehicle type	Vehicles we are interested in simulating (examples are car, tank, etc.)
controller	Algorithm to guide vehicles to a specified target point.

TABLE 1. Dictionary for vehicle simulator family

Different vehicle types can be guided according to a similar movement principle (normal or differential). For instance, both Car and Tank can use differential movement principles; however, Tank cannot be guided by normal movement. Control algorithms behave differently to drive vehicles toward specified target points (i.e. vehicle's behavior depends on kinematic vehicle properties).

Note that kinematic models presented thus far are limited in several ways. Real-world vehicles are subject to physical forces and constraints (i.e. inertial forces impede vehicles instantaneously reaching maximum speeds, in the real world vehicles accelerate slowly from stand-still to maximum speed). Other factors are the use of breaks, and environmental factors like wind, different surface types, and obstacles. We don't include these other factors here, which can be considered as opportunities for the simulator's family evolution.

A feature diagram for the vehicle family is shown in Figure 3.5. The diagram shows several types of features:

- Mandatory features: Type, Controller, Principle
- Optional features: Trailer
- Generalization features: Type (2-3wheelMotorcycle, Car, Tank), Controller (Intuitive, Fuzzy), Principle (Normal, Differential).

Several constraints restricting feature combinations are: a Vehicle with a Trailer cannot be a Tank, a Vehicle with Differential drive cannot be a 2WheelMotorcycle.

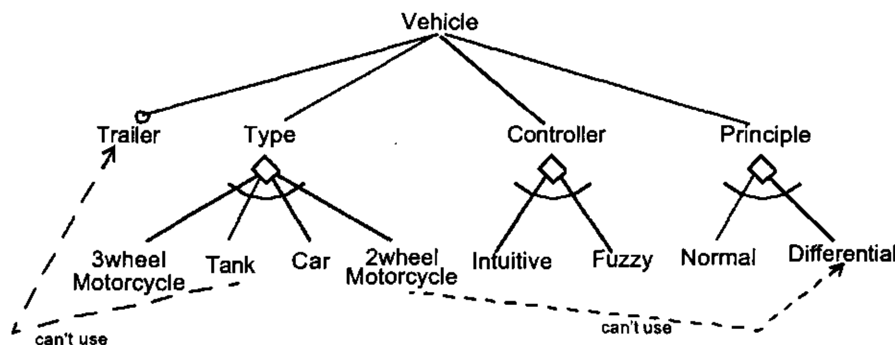

Figure 3.5: Feature diagram of a vehicle simulator domain

3.4 Domain design

As was described in Chapter 2, design modeling consists on devising a hierarchical product-line architecture (i.e. a GenVoca model). Our architectural representations are hierarchical representations of parameterized components. From the domain diagram, we construct a GenVoca model [BO92] which consists in a hierarchical diagram and declaration of layers (types) and their implementations Figure 3.6(a) and Figure 3.6(b), respectively. First of all, note that the simulators' hierarchical model has two layers (i.e., Path and Params) whose corresponding features are missing in the feature model. Path stores information about the trajectory being followed by a vehicle (this is important thus a graphic representation can be produced). Params define general parameters for a vehicle (i.e., coordinate position, facing angle, speed, etc.). Note in Figure 3.6(b) that a component notrailer has been included. This component exports a Trailer interface

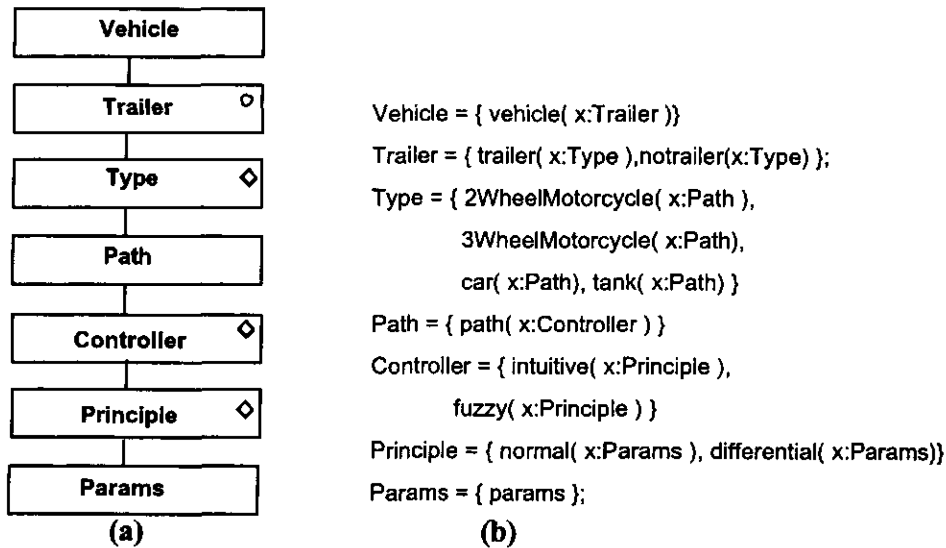

Figure 3.6: Hierarchical model of a vehicle simulators family

but its functionality is empty. Its purpose is to keep consistency in the composition.

Design model in Figure 3.6 can be analyzed compositionally (i.e., by detailing functionality each layer adds to the composition). This extensional perspective is important because it serves as a guideline of how to proceed in translating feature diagrams to hierarchical domain-specific architectures. First, in Figure 3.6 Vehicle stands as an interface to the application. When used, Trailer adds trailer properties to a vehicle (an empty implementation of trailer is used when no trailer is specified by the developer). Layer Type adds the behavior of a particular vehicle type (e.g., Car, Tank, etc.). Controller is the general algorithm for guiding the vehicle towards a point in a plane (e.g., Intuitive, Fuzzy[6], etc.). Finally, Principle defines how vehicle's wheels move (e.g., same or different speed).

Layers Vehicle, Path, and Params should always be part of a composition. This is due to the fact that every vehicle and its trajectory must be displayed in the screen (Path's job); similarly, every vehicle needs to be defined by a parameter set (contained in Params); and every vehicle has to define a user interface (Vehicle does this).

It is straightforward to derive different composition equations from the design model. For instance, two valid equations are:

```
S1=vehicle ( trailer ( car ( path ( intuitive ( normal ( params ))))))
S2=vehicle( notrailer( tank( path( intuitive(differential(params))))))
```

Composition **S1** defines a vehicle simulator for an autonomous car having a trailer and moving according to an intuitive algorithm and whose wheels move in a normal (i.e., constant) speed. Composition **S2** defines a simulator for a tank

6. *Fuzzy control* consists in that before performing every step, a rule-base is evaluated to find the best choice [PY98].

moving according to an intuitive algorithm and using differential speed; `notrailer` is an empty implementation of a trailer component.

3.5 Domain implementation

Our approach for domain implementation consists of implementing design models as wizlets and a configuration wizard as a specification tool for application construction and verification. This section describes these two steps for the vehicle simulators product-line.

3.5.1 Wizlet implementation

As mentioned in the introduction to this chapter, requirements can be satisfied by implementing the simulators family using the Java programming language. As mentioned, currently Java doesn't provide a standard mechanism for component parameterization. We discussed in Section 3.2 how parameterized Java components can be created by extending Java with a construct to express class parameterization similar to C++'s templates.

The following is the implementation of the simulator's family using our wizlet approach. The functionality required by vehicle simulators is rather simple, thus class nesting is not actually necessary. Instead, every layer consists of a single parameterized Java class. In Figure 3.6, wizlet Vehicle defines a wrapper for simulators and makes one call to its super wizlet, its implementation is:

```
public class Vehicle extends <<WizletSuper>> {
public Vehicle(float x, float y,double theta, double phi,
    double speed1,double speed2, double width, double length)
{
  super(x,y,theta,phi,speed1,speed2,width,length);
     ...
 }
```

```
public void performStep() {
    super.performStep();
    ...
  }

  public GeneralPath createVehicle(){
    super.createVehicle();
    ...
  }
 ...
}
```

A partial implementation of wizlet Trailer is:

```
public class Trailer extends <<WizletSuper>> {
  double theta2=0;
  GeneralPath bodyPath;
  Frame bodyShape, leftTireShape, rightTireShape;

  public Trailer(float x, float y, double teta, double phi,
         double speed1, double speed2, double width, double length) {
    super(x,y,theta,phi,spped1,speed2,width,length);
    ...
  }

  public void performStep() {
    super.performStep();
    . . .
  }

  public GeneralPath createVehicle(){
    super.createVehicle();
    ...
  }
  ...
}
```

Wizlet Trailer defines variables and functions for a trailer's shape and path. For a trailer, function `performStep()` re-calculates next position and re-draws tailer's shape. Function `createVehicle()` defines trailer's shape.

Most of the wizlets are similar to Trailer, thus we don't show them here. The top-most wizlet in the hierarchy is Params, which declares variables defining a vehicle's body and its orientation (direction). Auxiliary functions can be used by several layers in the hierarchy:

```
public class Params {
  float x,y,xf,yf;
  double theta,phi,speed1,speed2,radius,length,width;
  double maxPhi;

  public Params(float x, float y,double theta, double phi, double
    speed1,double speed2, double width, double length) {
    this.x = x;
    this.y = y;
    ...
    this.maxPhi=Math.toRadians(40); //max turning angle
    }

    // "set" functions for speeds, phi and final point
  . . .
  double angleLessThan180(double angle){
    while (!((angle > -Math.PI)&&(angle <= Math.PI))) {
      if (angle > Math.PI)
        angle=angle-(2*Math.PI);
      else
        if (angle < -Math.PI)
          angle=angle+(2*Math.PI);
    }
    return angule;
  }
}
```

3.5.2 Implementing Composition Verification

Limited composition consistency verification can be implemented by type declarations in the class hierarchy [Sma99]. However, industrial level application systems require more complex configuration rules than can be implemented in one direction (top to bottom). For instance, in an inheritance hierarchy, we cannot verify at the Principle level that there is a Vehicle component of the correct type, although we can verify the presence of the correct Principle (i.e., Principle is always a super wizlet of Vehicle).

To solve these problems, configuration predicate validation should be performed by an automatic tool implementing configuration predicates equivalent to constraints included on feature and GenVoca diagrams, and in GenVoca type declarations and instances. The tool can receive one string representing the composi-

tion equation, parse the equation and construct a parse tree, which can be traversed up and down to validate component consistency.

We implement a *validation* component for each component in the domain (wizlet repository); each component implements configuration predicates and explanation capabilities. Validation components are implemented as classes that implement two methods: `validate()` which performs configuration predicate evaluation for that component, and an `explain()` method, which defines a string explaining the role of the component in a composition. We implemented an abstract base class for components that are members of the parse tree[7]. The declaration for that abstract base class is:

```
abstract class Component {
  public Component next;
  public abstract void validate(State state);
  public abstract void explain(State state);
};
```

For instance, the declaration of a validation component for Car component is:

```
class Car extends Component {
  Car(); //details of constructor not shown here

  public void validate(State state) {
    if (!state.carSet){
       state.carSet=true;
    } else {
      state.addMessage("Error: Only one instance of Car is allowed\n");
      state.incErrors();
    }

    if (!state.pointSet){
       state.addMessage("Error: Car requires Point on top of it.\n");
       state.incErrors();
```

7. Note that equivalent results can be obtained if we declare a Java interface instead of an abstract base class. The selection of an abstract base class facilitates implementation consistency in programming languages not supporting class interfaces (both C++ and OO Pascal lack the support of class interfaces).

```
    }
    if (next!=null)
      next.validate(state);

  }

  void explain(State state){
    state.addMessage("The simulator contains an Car component.");
    next.explain(state);
  }

};
```

Each component asserts its presence by setting a boolean flag. A precondition for most of the components is that no duplicates are allowed. By checking the flag, we might know if a component of the same type appears before (is on top of) in the composition equation. A pre-restriction may be the existence of a particular component beneath. Again, we can check the flag to test if the property was asserted (as a post-restriction). The previous code segment shows the implementation of the `validate()` method for `Car`.

The previous code segment also shows the implementation of an `explain()` method, which is called if no configuration errors are found. Each `explain()` method appends one string to construct a full description of the composition.

Configuration predicate validation is an ad hoc process. The implementation of configuration predicates may vary widely from component to component. The explanation capability provides convenient support for application construction, both when compositions are valid or invalid. Messages for inconsistent composition equations help to construct correct compositions. On the other hand, explanations of correct composition equations helps to understand if semantics of the system defined by the composition are the same that we had in mind at specification time.

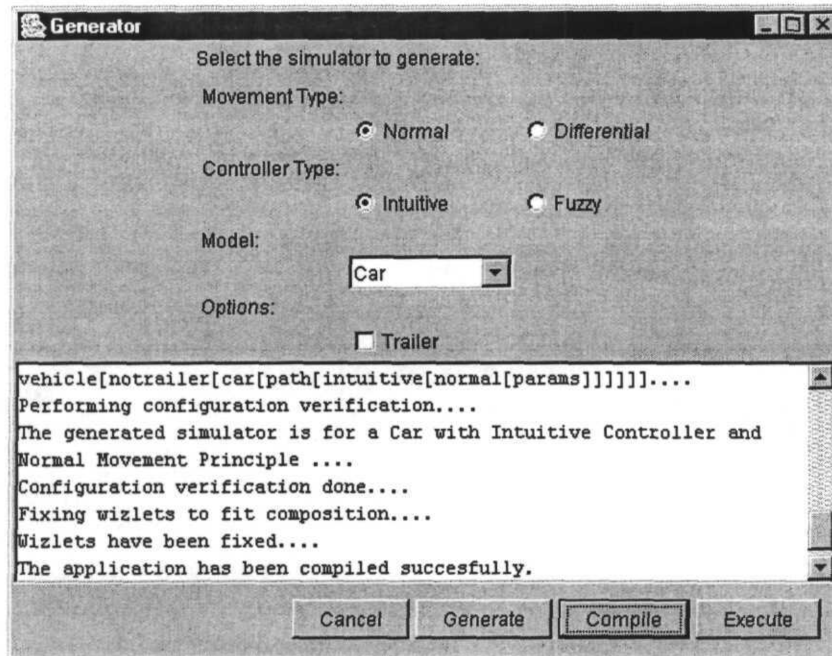

Figure 3.7: Specification interface for vehicle simulators family

3.5.3 Configuration Wizard Implementation and Examples

Figure 3.7 shows a possible specification interface for the vehicle simulators product-line. In the example, variants (i.e., different implementations) are grouped using different grouping facilities in the graphical user interface. For instance, the types of movements are in a group called "Movement type" where choices are "Normal" and "Differential"; "Controller type" grouping includes variants "Intuitive" and "Fuzzy"; "Model" define a list of the vehicle types the developer can choose from; finally, there is a single "Option" to specify if the vehicle is towing a "Trailer" or not.

The specification interface includes a visualization area which displays all messages informing the application developer about the process' status. Several

buttons allow the user to initiate different actions. A "Generate" button is used after a set of requirements has been specified. After generating, the button labeled "Compile" compiles the application. After compilation, the resulting simulator can be executed directly from the configuration wizard when the user presses the "Execute" button.

In Figure 3.7, several options are selected (a normal movement type, an intuitive controller, and a car vehicle type). From these selections, pressing the "Generate" button produces a component composition expression, performs composition validation, and generates the application by adapting wizlets using the composition expression. Pressing the "Compile" button, the application is compiled. Finally, pressing the "Execute" button, the generated application is started. The application produced is shown in Figure 3.8, which corresponds to a vehicle simulator for a car. Vehicles being simulated need a pre-defined initial position, direction, velocities, and a target point. The autonomous control guides the vehicle to the specified point, showing the path followed by displaying a point trace. Figure 3.8 shows the car and a trace after the car has moved several steps towards its specified target point.

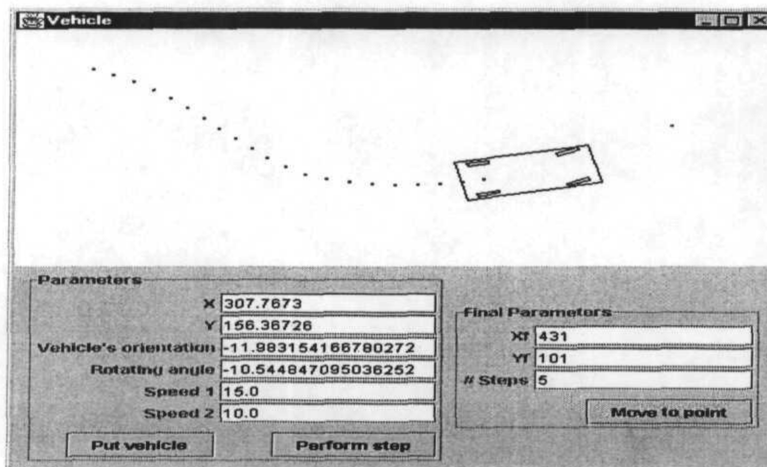

Figure 3.8: Simple car.

ration expression is that a trailer component was added to the composition of a car without a trailer. The description of the composition also includes an explanation for the trailer as part of the generated simulator.

To execute the generated simulator, initial position and velocities should be provided. The results after these parameters are provided are shown in Figure 3.10. As can be seen in Figure 3.10, a car and its trailer follow slightly different paths.

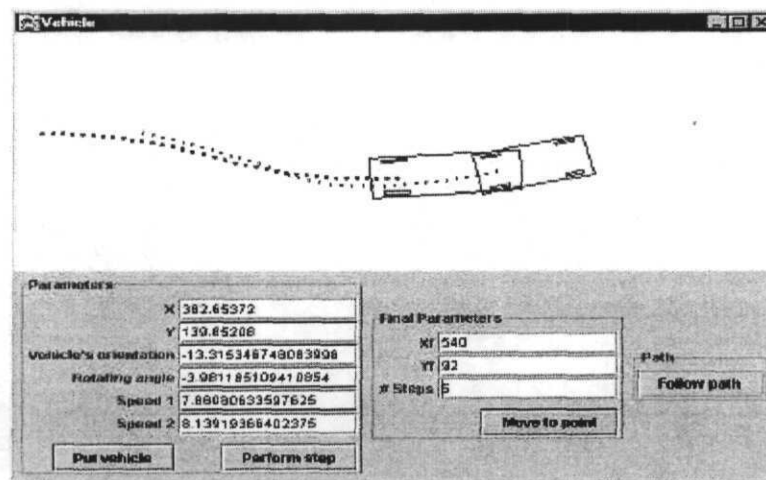

Figure 3.10 Car with trailer.

As explained in analysis and design, other simulator that can be generated is a simulator for a tank. The options selected to generate a tank are shown in Figure 3.11. Again, differences to previous examples can be seen in the composition expression and in the description of the application.

Tanks have different inherent behavior as that of a car. Tires in a tank allow to change direction in smaller spaces thus they can perform maneuvers using less space and face the target direction faster. Given this characteristic operation, tanks can not tow trailers, as trailers need more space to change direction, thus the flexibility of a tank could be limited if it has to tow a trailer. Figure 3.12 shows the tank simulator after several trajectories have been followed by the tank. Comparing the

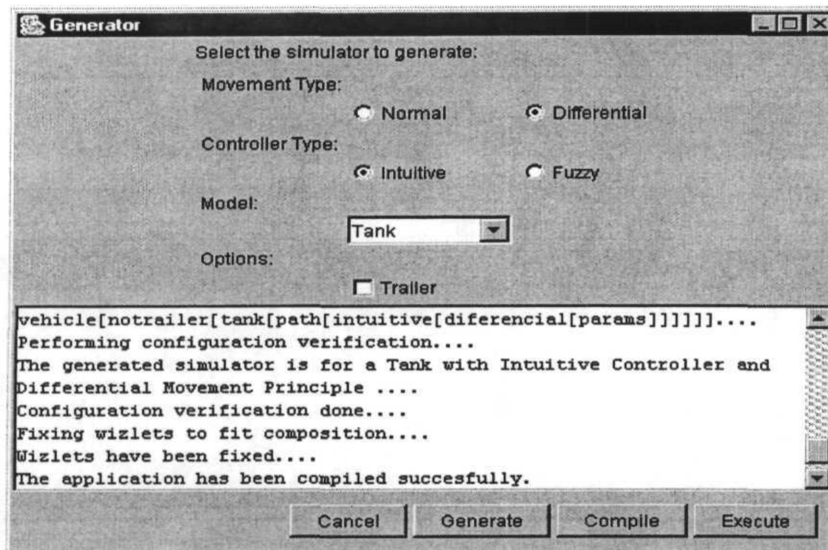

Figure 3.11: Specification of a tank simulator.

paths shown on Figure 3.10 and Figure 3.12 we can observe that in effect, a tank changes direction in a smaller space than is necessary by a car.

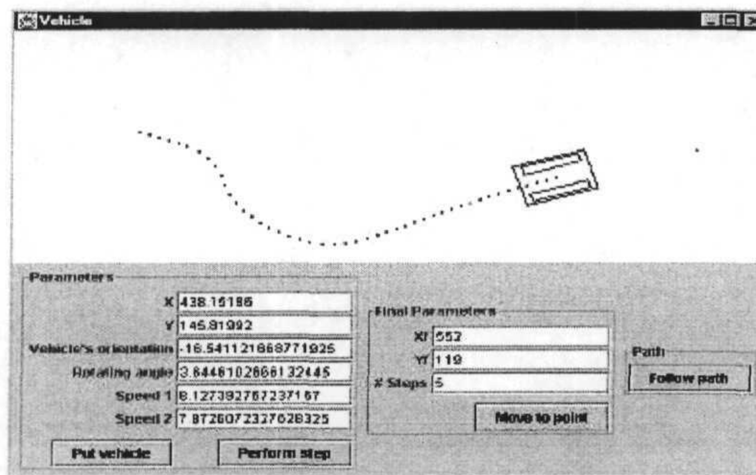

Figure 3.12: Simulator of a tank.

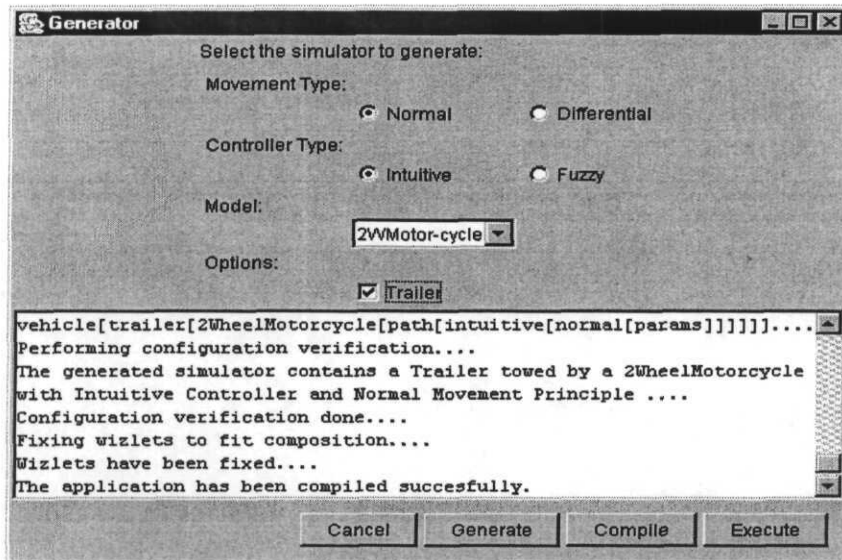

Figure 3.13: Specification of a motorcycle simulator.

The last example we show of a simulator produced by the configuration wizard is a simulator of a two-wheel motorcycle towing a trailer. The configura-

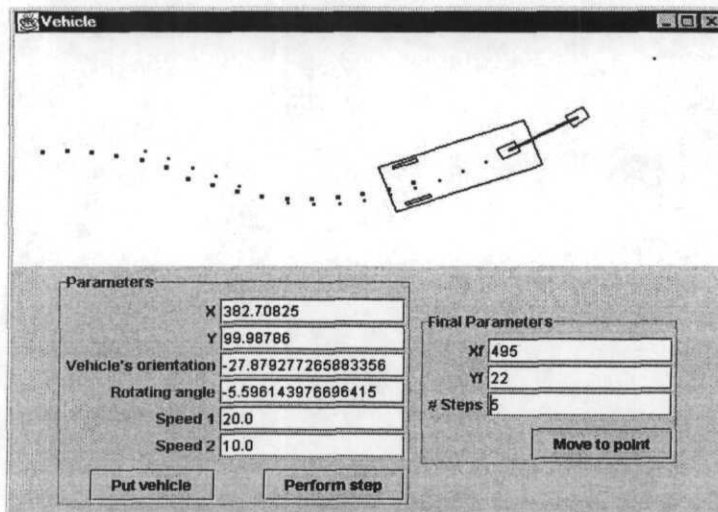

Figure 3.14: Two-wheel motorcycle with trailer.

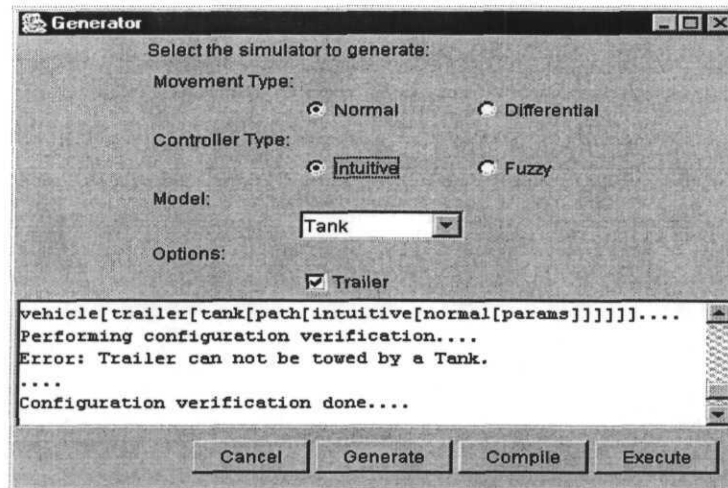

Figure 3.15: Tank towing a trailer is illegal.

tion is shown in Figure 3.13, and the simulator is shown in Figure 3.14, after position and speed have been specified and a target point marked in the simulation area.

Finally, Figure 3.15 shows that towing a trailer with a tank is illegal.

3.6 Discussion

Examples in Figure 3.8-Figure 3.14 show members of a simulators family that is possible to produce from the product-line infrastructure represented by wizlets and their accompanying configuration wizard. The autonomous vehicle simulators product-line is illustrative to show the concept of configuration wizards.

To make possible the implementation of wizlets in Java, the language was extended to support class parameterization. Although type parameterization capabilities in C++ are much more sophisticated than those we needed to add to Java, the combined approach of wizlets and configuration wizards does not required such sophistication. As can be seen in the example described in this chapter, con-

figuration wizards can implement simple specification interfaces allowing non-expert developers to specify and produce applications by selecting features from the specification interface, which are used to generate the application implementing specified features.

To analyze the scalability of our approach, following chapters show the approach applied in other more complex domains, and using other programming languages.

Chapter 4

Computer Numerical Control Systems

A *machine tool* is any machine that uses a tool to cut material to produce items with predefined geometric shapes. Machine tools are essential elements of modern manufacturing and a starting point of every operation intended to precisely cut material (metal mainly), to transform work pieces into useful products. A *computer numerical control system* (CNC) is an application that coordinates the motions in a machine tool. Geometric shapes are communicated to a CNC using a *numeric program.* A machine tool can be considered the processor for a numeric program and the numerical controller as the numeric program interpreter.

Our second example of a configuration wizard for software product lines is in the numerical control domain. In order to better understand the requirements of numerical controllers, the first section includes a general description of tool machines and their programming, and numerical control characteristics and requirements. This establishes a framework for defining a domain model for a computer numerical control system, which we present in the second part.

Even though the model presented in this chapter is applicable to a broad class of two- and three-axes tool machines, our discussion centers more on an engine lathe (a.k.a. turning machine). This selection is based on two facts: turning machines are the most common type of metalworking lathe used in industry, it is the most versatile machine tool (with proper setups and accessories it can perform

such machining operations as facing, straight and taper turning, drilling, threading, milling, grinding, boring, forming, and polishing). Other machine types are presented with fewer details, in order to keep description short.

4.1 Motivation

The implementation of computer numerical control systems is a difficult endeavor [GH91, MBY+00, OSA, OSE97]. Real-time characteristics and communication coordination are complex to implement. Developers must be familiar with three areas: knowledge of real-time characteristics in operating systems, knowledge of how servomechanisms operate, and the functionality that a particular type of machine tool can perform [Ell94]. Activity synchronization is essential to get the most out of control elements in a machine tool. The precision required by manufactured items and performance have to be carefully balanced to achieve the better results at the higher production ratios.

To meet performance and precision requirements simultaneously once, controlling software has to be carefully tuned for each intended task to be performed. Machine tools cost is high [GH91]. In order to lower the cost, the machine must operate with the cheaper servomechanisms available for the intended operations. Control software must adequately exploit servomechanisms to perform complex machining tasks with the cheaper components.

Trouble increases when several similar CNC systems have to be constructed [Ram98]. Cost-effective solutions may dictate that a reuse approach should be applied, but constraints seem to require the application of ad-hoc techniques to meet performance requirements. To meet these requirements, there is the need of a product-line which should flexibly support necessary characteristics. A domain architecture for CNC systems must emphasize *modularity* so main parts

can be easily identifiable, helping to evolve the product line to adapt it to future requirements. It should be *adaptable* to control different types of tool machines. It ought to be *scalable*, thus the addition of new modules (components) should facilitate multiple new valid compositions.

The goal of the CNC project at Monterrey Tech was to build the infrastructure (mechanical, electronic, and computational) to facilitate machine tool retrofitting. This project was aimed at underdeveloped countries which have thousands of old manual machines, and cannot afford the cost of acquiring new automated machines. Old manual machine tool automatization is an option to compete in the global economies, if such a solution can be implemented at relatively low cost [Ram98].

The machine tool domain is complex. The following section is a lengthy introduction to machine tools, their controllers and programming.

4.2 Numerical Control

Numerical control (NC) of machine tools is simply the control of machine tool functions by means of coded instructions. Examples of machine tool functions are: moving the table, turning the spindle on or off, changing the cutting tool, indexing a part, or turning the cutting fluid on or off [GH91].

Almost all industries utilize or are affected by NC, and there appears to be no end to its application. Industrial users of NC equipment are: aerospace, electronics, automotive, etc. [BK94]. Lathes are machine tools useful in many cutting operations. There are two major types of NC lathes: the engine and the turret. A lathe engine is a two-axis machine having longitudinal and traverse motions. Most NC engine lathes have the ability to turn, face, bore, machine external or internal tapers, and machine threads.

4.2.1 NC Machine Tool Elements

NC machine tool elements consist of axis nomenclature, dimensioning systems, control systems, servomechanisms, and open- or closed-loop systems. It is important to understand each element prior to actual programming of a numerically controlled part.

a. Axis Nomenclature

NC machine tools base their motions on the standard Cartesian coordinate system. Standard definitions for X, Y, and Z axes relating to most machines are as follows:

X axis:

1. Must be horizontal.
2. Must be perpendicular to the Z axis.
3. Is generally the longest axis of movement.

Y axis:

1. Perpendicular to X and Z.

Z axis:

1. Is always parallel to the spindle and perpendicular to a plane established by X and Y.

In addition to the X, Y, and Z axes, several rotation movements can be accomplished on NC machine tools around the axes. We will not describe these axes here, because they are related to machine tools more complex than those we are interested in (i.e., machines having four or more axes).

b. NC Measuring System

The term *measuring system* in NC refers to the method a machine tool uses to move a part from a reference point to a target point. A target point may be a certain

location for drilling a hole, milling a slot, or other machining operation. The two measuring systems used on NC machines are *absolute* and *incremental*.

- ***Absolute system***. The absolute measuring system uses a fixed reference point (origin). It is on this point that all positional information is based. In other words, all the locations to which a part will be moved will be given dimensions relating to that original fixed reference point.
- ***Incremental system***. The incremental measuring system has a floating coordinate system. With the incremental system, the machine establishes a new origin or reference point each time the part is moved. That is, each new location bases its values relative to the preceding location. One disadvantage of this system is that any error made will be repeated throughout the entire program, if not detected and corrected.

c. NC Control Systems

There are two types of control systems commonly used on NC equipment: point-to-point and continuous path.

- ***Point-to-point Systems***. A point-to-point controlled NC machine tool, sometimes referred to as a positioning control type, has the capability of moving only along a straight line. Point-to-point systems are generally found on drilling and simple milling machines where hole location and straight milling jobs are performed. Point-to-point systems can be utilized to generate arcs and angles by programming the machine to move in a series of small steps. Using this technique, however, the actual path machined is slightly different from the cutting path specified.

- ***Continuous-Path Systems***. Machine tools that have the capability of moving simultaneously in two or more axes are classified as continuous-path or contouring. These machines are used for machining arcs, radii, circles, and angles of any size in two or three dimensions.

d. NC Servomechanisms

NC servomechanisms are devices used for producing accurate movement of a table or slide along an axis. One common type of servo is an electric *stepping motor*. Stepping motor servos are frequently used on less expensive NC equipment. These motors are generally high-torque power servos and mounted directly to a lead screw of a table or tool slide. Stepping motors are actuated by magnetic pulses from the stator and rotor assemblies.

e. Open- and Closed-Loop Systems

Closed-Loop. A closed-loop system compares the actual output with the input signal and compensates for any errors. A *feedback* unit actually compares the amount the table has been moved with the input signal. Some units used on closed-loop systems are transducers, electrical or magnetic scales, and synchros (e.g., motors). Closed-loop systems greatly increase the reliability of NC machines.

Open-Loop. NC machines that use an open-loop system contain no *feedback* signal to ensure that a machine axis has traveled the required distance. That is, if the input received was to move a table axis 1.000 in., the servo unit generally moves the table 1.000 in. There is no means for comparing the actual table movement with the input signal, however. The only confidence one can have that the table actually moved 1.000 in. is the reliability of the servo system used. Open-loop systems are, of course, less expensive than closed-loop system.

Control of a machine tool can be conceptualized as a canonical form of feedback system (see Figure 4.1) [Ram98]. Five components are present:

- ***Input program.*** Is the NC program describing movements to be performed by the machine.
- ***Control unit.*** Interprets program operations and converts them to control signals for machine actuators. It is the checkpoint where program's reference position and real-position are compared.
- ***Actuators.*** Devices that execute control signals and convert them to mechanical actions to move machine's mechanisms.
- ***Feedback devices.*** Measurement instruments that supply the control unit with the real position of the machine. Feedback devices can sense actuators or sense axis linear movement.
- ***Machine tool.*** Is the final element to be controlled.

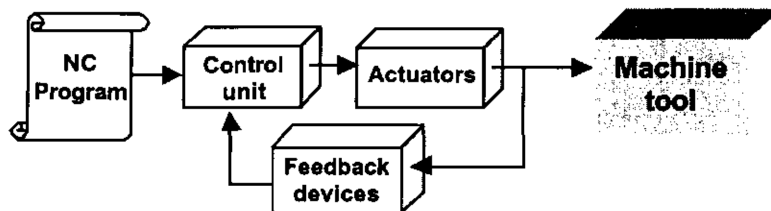

Figure 4.1 Canonical form of a NC system.

4.2.2 NC Programming

Numerical control *instructions* (also known as *blocks*) in a program appear as words made of individual codes. A common format is the word *address format* (also known as EIA-274), standardized by EIA (Electronic Industries Association). EIA-274 assigns an alphabetical code (in upper case) to each function word. The purpose of a *letter address* is for word identification. Words do not have to appear in a rigid format. Different machines may have different sets of words that are rec-

ognizable (e.g., a lathe may not recognize instructions aimed to a milling machine). Functions may consist of:

- ***sequence numbers***: give an ordering to instructions
- ***preparatory and miscellaneous functions***: define units (inches or millimeters), measuring system (absolute or incremental), etc.
- ***X-, Y-, and Z-coordinate information***: with values expressed as absolute or incremental values
- ***spindle speeds:*** adjust speed to accommodate different materials
- ***feed rate:*** is defined as the linear displacement of the tool relative to the workpiece, in the direction of the feed motion, per stroke or per revolution of the workpiece or tool.

4.2.3 Program codes (letter address)

Letter addresses are single letter identifiers used as prefixes to both instructions and values. Sets of instructions or function words use the same letter address, and a different number (e.g., M and G). Other letter address identify parameters (e.g., X, Y, I, K, etc.).

N Block sequence number address. Followed by a number (e.g., N010, N020). The program will be executed from the lowest block number to the highest.

X,Y,Z These addresses signify axis motion in accordance with the designated axis motions of the machine tool. The dimension address will be followed by a signed number.

I,K,R These addresses are used when employing circular interpolation to specify the center of the programmed arc. The commands for X, Y, and Z coordinate addresses equally apply.

T Tool function code identifying the tool to be used, or loaded if at a tool change. Normally followed by 2 digits. Accompanying each tool will be a

corresponding *tool length offset* (represented by two extra digits) which is accessed, during machining, by the tool code.

S Spindle speed letter address. The digits following the address represent the desired speed.

F Feed rate letter address. The digits following the address represent the desired feed rate.

M Miscellaneous function letter address. M-functions are a family of instructions that cause the starting, stopping or setting of a variety of machine functions. The address letter is followed by 2-digits. Following is a table of common standardized M-functions (EIA-274):

M00	Program stop (AC)
M01	Optional stop (AC)
M02	End of program (AC)
M03	Spindle on (clockwise) (W)*
M04	Spindle on (counterclock) (W)*
M05	Spindle off (AC)
M06	Tool change
M08	Coolant on (W)*
M09	Coolant off (AC)*
M30	Program end (AC)

TABLE 2. EIA-274 standard M-functions.

In the table, letters inside parentheses denote the timing of the particular function within the block in which they appear. (AC) indicates that the M-function will be executed after completion of any commanded axis motion, and (W) indicates that the function will be executed with any commanded motion. An asterisk denotes that the function is retained until it is cancelled or superseded; such functions are known as *modal functions*.

G Preparatory function letter address. G-functions are a family of instructions that change the control's mode of operation. For example, changing from

metric to inch units or from absolute to incremental coordinates. Following is a table of common standardized G-functions (EIA-274). In the table,

G00	Rapid movement
G01	Linear interpolation
G02	Circular interpolation (clockwise)
G03	Circular interpolation (counter clock)
G04*	Pause
G20	Inch units
G21	Metric units
G28*	Go to tool home (use an intermediate position)
G29*	Return from tool home
G40	Cancel cutter compensation
G41	Cutter compensation left
G42	Cutter compensation right
G70	Finishing cycle
G71	Longitudinal roughing cycle (turning cycle)
G72	Face contour roughing cycle
G73	Contour parallel roughing cycle
G74	Peck drilling
G75	Grooving
G76	Multi-pass threading
G90	Absolute coordinates
G91	Incremental coordinates
G94	Facing cycle
G95	Cylindrical roughing cycle

TABLE 3. EIA-274 standard G-functions.

4.2.4 NC programming procedures

This section presents examples of NC program instructions portions. In these examples, all axes are assumed to be under single or contouring control. Machine datums will be specified assuming a zero offset facility. Only one preparatory (G) function and one Miscellaneous (M) function per block of information. All dimen-

sional information is given in millimeters. Speed codes are rev/min values and feed codes are mm/min values.

NC programs contain several identifiable parts: setup, move to commanded position, perform interpolation moves, resume commanded position, and end.

a) Starting a NC program

The starting point in a NC program informs the control system of the various setup conditions for the machine task that follows:

- Coordinate values (either absolute or incremental).
- Dimensional units (either metric or inch).
- Tool number.
- Spindle speed.
- Start spindle rotation.

The start of a NC program may thus take the following form:

```
N010  G90          ..Absolute coordinate
N015  G21          ..Metric units
N020  M06 T01      ..Tool change
N025  M03 S2000    ..Spindle on at specified spindle speed
```

b) Programming positional moves

A *positional move* causes the tool to move to a commanded position without any cutting taking place. Such moves are normally performed at rapid traverse. Rapid positioning mode is activated by issuing G00 within the NC program. A positional move may take the following form:

```
N035    G00 Z0             ..Set rapid traverse, retract tool
N040    X159.75 Z250.25    ..Move in X-Z
N045    G01 X160           ..Set linear interpolation mode
```

c) Machining using interpolation

Machining using interpolation simply means machining in straight lines or describing circular arcs.

- **Linear interpolation.** Linear moves may be horizontal, vertical, or at an angle, in any direction. All machining is done under control of feed. A program segment to make a 200 mm horizontal cut, followed by a 300 mm vertical cut, 5 mm into the surface in the work piece may take the form:

```
N050   G01 Z-55     ..Set interpolation, feed down
N055   M08          ..coolant on
N060   G01 X200     ..Horizontal cut
N065   G01 Y300     ..Vertical cut
N070   G00 Z-5      ..Retract spindle -rapid
N075   M09          ..coolant off
```

- **Circular interpolation.** A single circular interpolation command block is capable of producing a circular arc spanning up to 90°. Circular interpolation is limited to contouring in a single plane (i.e., in two dimensions only). When milling, this plane should be selectable (X-Y, Y-Z, and X-Z). When machining an arc, four pieces of information need to be specified:

1) The ***start*** position is assumed to be the current tool, or cutter, coordinate position.
2) The ***end*** position of the arc is specified by X, Y, and/or Z coordinates measured from the ***start*** position.
3) The radius is dealt with by specifying the coordinate position of the center of the required arc. Letter addresses I, J, and K are used for this purpose. I is used to specify the center of the arc in the X-direction; K is used to specify the center of the arc in the Z-direction.
4) The direction of cut is specified by a unique G-code. G02 is used to specify clockwise circular interpolation and G03 is used to specify counter-clockwise circular interpolation.

A program segment for illustrating circular interpolation programming for a turning machine follows (absolute coordinates):

```
N030   G00 X0 Z86        ..Rapid to tool start point 1
N035   M03               ..Spindle ON
N035   G01 Z85 M08       ..Feed to start of rad, point 2, Coolant ON.
N040   G02 X15 Z70 K-15 ..CW circular arc to point 3..
N045   G03 X23 Z62 K8    ..C/CW circular arc to point 4..
N050   G00 X24 M02       ..Retract tool, rapid, spindle OFF..
N055   X100 Z150         ..Rapid back to initial point X100, Z150
```

Figure 4.2 shows the programmed move for the previous program segment.

d) Machining using canned cycles

A *canned cycle* is a (user defined) fixed sequence of operations, that can be brought into action by a single command. Such cycles considerably reduce programming time and effort, on repetitive and commonly used machine operations. Canned cycles form part of the family of preparatory G-functions. G70 to G76 and G92-G95 are reserved for the various cycles.

Fixed cycles automatically perform a number of discrete operations as designated by the appropriate G-code. Common fixed cycles for turning operations are: straight turning, taper turning, face turning, and taper face turning cycles; area clearance (stock removal), grooving and peck drilling cycles; thread turning, taped tread turning, and multi-start thread turning. Most cycles must be accompanied by additional information in the command block. Thread turning, for example, may

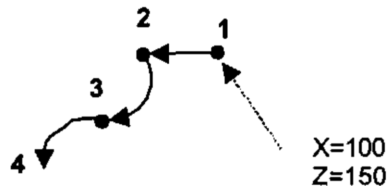

Figure 4.2. Machine tool motion path.

require the lead, depth of thread, and number of passes to be specified. The cycle contour-parallel is used for work pieces with a shape that is equivalent to the contour of the finished part.

A stock removal, or area clearance cycle, is simply a roughing cycle whereby a number of passes are made to clear large amounts of material. The tool will traverse a similar tool path, automatically increasing its depth of cut at each pass.

An example of a canned cycle is shown in Figure4.3. The resulting profile is manufactured by the code segment:

```
N025   M03 S800            ..Spindle ON at 800 rpm
N030   G00 X2 Z0           ..Rapid to tool start (1)
N035   G72 P40 Q60 F0.25   ..Perform facing cycle
N040   G01 X0 Z0           ..Take workpiece as home point (2)
N045   G01 X1              ..First position of profile (3)
N050   G01 Z-0.5           ..Cut horizontal part (4)
N055   G01 X1.5 Z-1.5      ..Cut diagonally (5)
N060   G02 X2 Z-2 R0.5     ..Cut circular segment (6)
N065   G70 P40 Q60 F0.12   ..Finishing cycle
```

Figure 4.4 is a sketch for the profile produced by a G73 canned cycle. G73 is used when the work piece almost has the final profile and a repetitive parallel

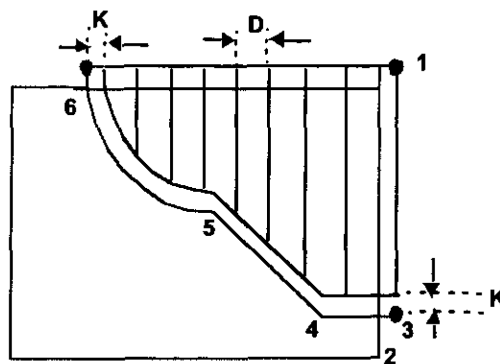

Figure 4.3 Facing cycle (G72).

cutting is enough to produce the final profile. The instruction segment following obtain the work piece shown on Figure4.4:

```
N040   M03 S800             ..Spindle ON at 800 rpm
N045   G00 X2.5 Z0          ..Rapid to tool start (1)
N050   G73 P55 Q85 D0.02    ..Contour parallel
N055   G01 X0 Z0            ..Move to home point (2)
N060   G01 X0.5             ..First position of profile (3)
N065   G01 Z-0.75           ..Cut horizontal part (4)
N070   G01 X1 Z-1.25        ..Cut diagonally (5)
N075   G01 Z-1.5            ..Cut diagonally (6)
N080   G01 X1.25 Z-1.75     ..Cut horizontal part (7)
N085   G01 X2               ..Cut diagonally (8)
N090   G70 P55 Q85 F0.12    ..Finishing cycle
```

Note that blocks (lines) 55-85 define the final desired work piece profile. Instruction in line 50 cut the material at a fast rate. The instruction in line 90 commands the cutting tool to perform a final pass along the profile thus a finer and polished surface is obtained in the work piece.

4.3 Project's Goals

Previous description of NC programming yields the foundation to better understand our second proof-of-concept product line. As explained, a project underway

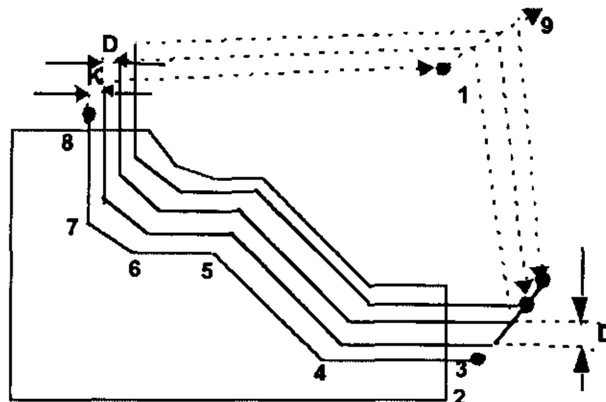

Figure 4.4 Contour parallel (G73)

at Monterrey Tech is aimed at constructing CNC systems for retrofitted two- and three-axes machine tools, with the ability to perform EIA programming codes, handle either incremental or absolute positioning systems, work with either inch or metric units, continuous-path, operate on closed-loop, and incorporate several canned cycles. The project is aimed at implementing a product-line of CNC systems for controlling retrofitted two- and three-axes tool machines [Ram98].

A goal in the project is to provide for maximum flexibility in the construction of different CNC systems thus a developer can select among different motor types, data acquisition cards, drivers, interpolation algorithms, canned cycles, etc. Several of the more important characteristics in which projected tool machines can differ are:

- ***Motor type.*** There are at least four different motor types: stepper, AC, DC, and linear. From these, stepper motors are the most easily to operate; to perform a "step" the motor has to be sent an electrical pulse (a different wire determines if the step has to be performed forward or backwards). AC, DC, and linear motors are more difficult to control, and will not be discussed here.
- ***Motor precision.*** Movement instructions in NC programs are expressed in linear values; these values have to be translated to angular values and then to steps. For each step, the motor has to receive a pulse. Thus, motors with different precision require different pulse amounts to move cutting tools equivalent distances. Thus algorithms translating linear motion values specified in NC programs have to be adjusted to perform correct translations from linear values to pulses (steps). Each motor has to have at least a value of its precision to translate linear to angular values.
- ***Spindle's controller.*** Instructions for spindle are expressed in RPM (revolutions per minute). A spindle is controlled by a hardware driver, and drivers

are controlled by voltages typically in the range between 0 and 10 volts. From these voltage values, drivers generate high-voltage output at variable frequencies. Using different spindle motors requires changing the driver configuration (i.e., there is no need to change software in any way for different motors). However, changing the driver requires an equivalent change in the algorithm performing the translation from RPM to voltage values.

- ***Data acquisition card*** Data cards perform two tasks: send pulses and voltage values to tool machine elements, and detect responses from tool machine elements. These responses are used to feedback control software so informed actions can be performed according to received values. Different data acquisition cards have different configurations; usually configurations depend on pins in card's connectors, which are mapped to registers inside the acquisition card. In different cards, values are sent through different wires and received in different wires. Although cards from different manufacturers or different cards from the same manufacturer may have similar functionality, the hardware interface is different. In such cases the controlling software has to be configured according to a card's configuration to which the machine tool being controlled is attached. Also, different cards can control two, three or more motors of one or more types. The correct control algorithm should correspond to the capabilities and characteristics of the acquisition card.
- ***Machine type***. Machine tools may differ in axis number, instructions they can perform, dimension of the working space, and on all other factors mentioned above (i.e., have different data acquisition cards, motors of different precision, etc.). Most of these differences can be understood as using different acquisition cards (to control different axis numbers), motors having different precision, etc. Dimension of the working space is the parameter that is not determined by other elements in the machine tool. What this means is

that characteristics of a particular machine tool can be attributed to each of its constituting elements, but that is not the case with working space dimensions.

These differences help to understand the variability that the product-line infrastructure should support, and give hints on how variations can be implemented to support diverse configurations.

4.4 A FODA Model for CNC Systems

From the previous lengthy introduction to the CNC systems domain, we can derive the feature diagrams on Figure4.5 and Figure4.6.

Feature diagram in Figure 4.5 shows that there are four subsystems (features at level 1): UserInterface, Translator, MotionGenerator, and MotionControl. UserInterface handles windowing and related events. Translator produces an intermediate representation of NC programs. MotionGenerator produces detailed traces for tool motion. MotionControl controls the tool machine to perform commanded instructions. Figure 4.5 shows that the only optional features in MotionGenerator subsystem are canned cycles. Figure 4.6 shows that only one motor type can be used in a single machine tool (spindle's motor is the exception, because it is controlled independently of the other motors).

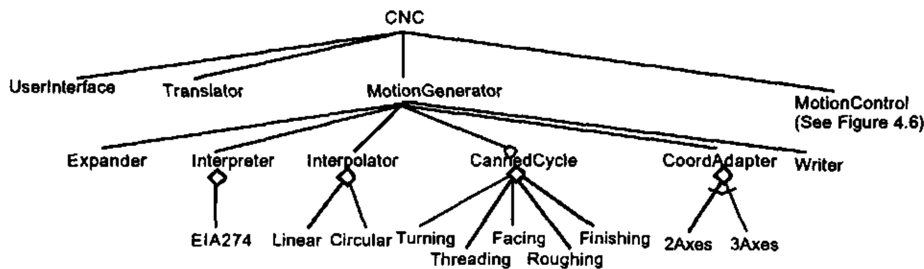

Figure 4.5 Feature model of CNC domain

Note also that all features in MotionControl are mandatory, and "third level" sub-features are exclusive (e.g., only one AxesController, Motor, or Card can be used at a time). It is also worth noting that at this moment there is no information about how features will be organized as components in an architecture for CNC systems.

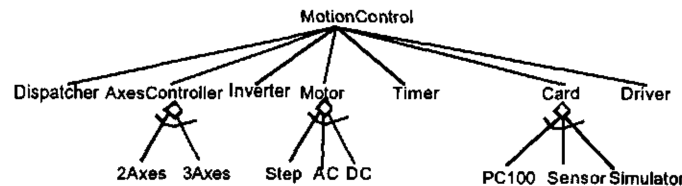

Figure 4.6 Sub-feature model of MotionControl

4.5 Hierarchical Models for CNC Systems

This section includes a more detailed explanation of subsystems and hierarchical models for the feature model described in previous section.

Figure 4.7 shows a high-level architecture of a CNC system (at subsystem level). For performance reasons, we chose not to use a GenVoca layered architecture for the full CNC system. Most of the subsystems run in separate threads and different tools besides component composition are used, as we describe latter in following sections. The system is organized as follows. User interaction is managed by the UserInterface subsystem. Here users issue commands for editing, compiling, and running NC programs. Users can edit configuration parameters for

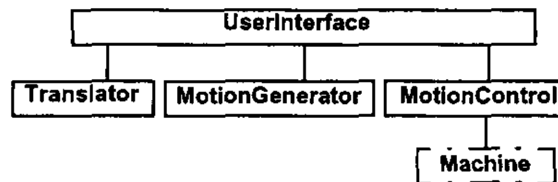

Figure 4.7 CNC's high-level architecture

a particular machine tool (i.e., maximum instruction feed-rate, working-area dimensions, etc.). A special kind of interaction is available to users: there are controls to allow the user define/set preferred reference position from which machining tasks should start (called the *user home*).

After editing a NC program, a Translator performs parsing and syntax analysis. If correct, symbolic representations (in G-and-M codes) are translated to an intermediate notation. Translator removes modal instructions by inserting appropriate instruction codes to all its output. We use lexical and parser generators (Lex and Yacc) to implement Translator subsystem.

Once a NC program has been "translated", motion instructions it defines can be prepared to be feed to a tool machine. The MotionGenerator subsystem shown in Figure 4.7 performs such preparatory actions, as follows. Users can write NC programs using different measurement units (inches or millimeters) and coordinate systems (absolute or incremental). Machining operations on work pieces typically consist on cutting-tool motions in different directions. Axis motors can perform discrete and small steps. Moving from the current position to a target position commonly requires that axis motors perform a number of steps. Each cutting tool motion instruction is then translated to the number of steps necessary depending on motor precision (i.e., steps needed to move a given linear distance). Diagonal and circular movements are more complex; in this case instructions are translated to step sequences that activate successively the appropriate axis for the cutting tool to move following the required path. MotionGenerator produces motion paths and keeps consistent measurement units (inches) and measurement system (relative coordinates) for all tasks it performs.

Another subsystem in Figure 4.7 is MotionControl, which controls the machine tool. Machine tool elements are governed by inertial laws (i.e., motions cannot be constantly performed at high speed, machine elements has to accelerate/

decelerate from current speed up/down until desired speed is reached). In general, physical components in machine tools respond to instructions at lower speeds than the electronic controlling equipment that feeds them. There may be times when commands are sent to a higher speed than that a machine tool can respond; in such cases these commands will be ignored (missed). The control must correct such events and re-send missed commands whenever necessary. Feedback mechanisms keep track of such events. The machine tool operator must be informed at real-time of the machine status (i.e., coolant on/off, spindle's speed, cutting tool position, etc.). MotionControl controls all these tasks.

The following sections preset detailed descriptions of both MotionGenerator and MotionControl.

4.5.1 MotionGenerator subsystem

The MotionGenerator subsystem is an intermediate module in a CNC system. Its input is a file (the output produced by the Translator subsystem) containing symbolic equivalents of the original NC program entered by the user. In general, the task performed by MotionGenerator is similar to that of a code generator in a programming language compiler. In a traditional compiler, high-level instructions are translated to assembly or machine language instruction sequences (i.e., high-level instructions are abstractions of lower level tasks performed by the computer's processor). In a machine tool, high-level instructions represent commands for moving from an initial position to a target position. However, at low level, machine elements can perform more simple tasks (i.e., move a fraction of an inch for each instruction). Thus, high-level instructions have to be translated to several more detailed instructions, according to machine-tool's specific characteristics and particular instruction (i.e., depending on the movement instruction different algorithms are used to produce the detailed output).

Other high-level tasks performed by MotionGenerator are intended to increase CNC system's performance. For instance, a consistent measurement unit is kept (i.e., the output from MotionGenerator is always expressed in inches, regardless of the measurement units set by the user); thus, no time is spent on measurement unit translation when instructions are fed to machine tool elements.

Figure 4.8 shows an expanded layered architecture of the MotionGenerator subsystem, which consists of a stacking of six components. The component at the top is Expander, who controls the translation (or expansion) process performed by MotionGenerator. For reasons that will be evident later (when we discuss CannedCycle component), Expander stores the input program in a local data structure; then the input program is sent as a stream to the next lower-level layer. Note that Expander is independent of instruction format or machine tool's specifics, it just "blindly" reads the input program and streams it down. Expander handles all interactions with UserInterface subsystem, which consists of starting the translation process for a given input file, and sending back the translation results.

The next component in the hierarchy is Interpreter. Interpreter distinguishes among instruction codes, and keeps the translation process' status. Process status is determined by current measurement unit mode (inches or millimeters) and reference position (absolute or incremental). Interpreter knows instruction formats, that is, it knows what parameters are needed for each instruction type, and the

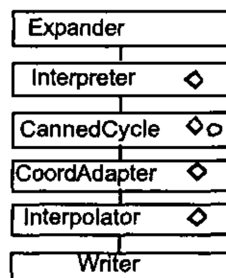

Figure 4.8 MotionGenerator subsystem

order in which these parameters are expected. According to instruction codes, the appropriate transformation is requested to a lower level layer, sending the necessary parameters. In this way, Interpreter receives an instruction stream, decodes every instruction and complements them with information of which transformation function should be applied. When necessary, Interpreter translates measurement units from millimeters to inches so that lower level layers operate consistently on a uniform unit mode (inches) basis. Machining precision is in thousandths of an inch; thus Interpreter also translates distance values to thousandths. In summary, Interpreter's task is to transform instructions by converting measurement parameters to thousands of an inch, regardless of the current measurement unit defined by the user; then requests particular additional transformations from lower level layers, according to each instruction type.

CannedCycle layer is below Interpreter. CannedCycle receives a profile that should be the result of a machining task; such profile must be obtained by repetitively executing patterns of movement. Profiles are defined by groups of linear and circular instructions (i.e., G01, G02, and G03). Canned cycle instructions include parameters declaring how the particular profile is to be obtained; these parameters define cutting depth and cutting-tool retraction distance for each cycle. Each particular canned cycle operation is performed very differently from the others, thus components implementing canned cycles must determine operations sequences that need to be performed to obtain the specified profiles. A CannedCycle component translates work piece profiles defined by lines and arcs to patterns of linear movements whose result will be to produce the specified profile in the work piece.

At this point, instructions are expressed in thousandths and all constitute simple preparatory (Gxx) or miscellaneous (Mxx) instructions (that is, canned cycles have been expanded to linear movements. What remains to be done is to fix

the measuring system. The CoordAdapter layer translates absolute values to incremental representations, thus lower level layers consistently deal with incremental measurements. CoordAdapter supports both absolute and incremental systems; thus swapping between them would be relatively easy. For instructions requiring additional transformation (i.e., linear and circular movement), CoordAdapter requests additional transformations from its lower level layer, all other instructions are directly send to output.

Interpolator is the last layer performing instruction transformations. As already mentioned in this and previous sections, initial and target positions are part of movement instructions. However, target positions typically are farther than it is possible to advance in a single motor step. Additionally, it is rarely the case that movements will be along a single coordinate axis (i.e., horizontal or vertical). Thus, for most of the cases, the cutting tool will need to move diagonally to reach target positions. In such cases, both motors controlling positioning in the plane of movement should be activated simultaneously for moving the cutting tool along the trajectory defined by the movement instruction (linear or circular). As was previously mentioned, at this point all measurements are expressed in thousandths of an inch and values are incremental. In order to move the cutting tool along the appropriate trajectory, we need to know the direction at each intermediate position. Depending on the trajectory (linear or circular), we can select an appropriate algorithm to produce all intermediate movements that motors should perform. Note that, at this point, movement precision does not depend on motor precision, but only on the minimum precision we are interested to obtain on the finished work piece. Thus the same output can be useful to command motors of different precision and type. Finally, Interpolator doesn't know how to communicate its results; all intermediate positions are sent to the next lower layer.

Subsequent stages on the CNC system need to know how many expanded instructions are produced for every input instruction, before performing the operations. Thus the protocol implemented in Writer is to store all expanded instructions it receives and counting them until a flush method call is received. Writer sends first the counter value and then the expanded instructions.

It is now time to see how different MotionGenerator subsystems can be constructed for machines incorporating different elements. In the previous section we described possible changes in a machine tool, which require equivalent changes in the controlling software. Here is how MotionGenerator can be adapted for every change:

- ***Motor type***. Instructions that have to do with motors controlling cutting tool positioning are translated to traces whose units are always expressed as thousandths of an inch. That precision is independent of a particular motor type and precision. Thus a different motor type does not require any change in MotionGenerator.
- ***Motor precision***. See discussion for motor type in previous paragraph.
- ***Spindle's controller***. In what concerns MotionGenerator, spindle's motor is just turned on/off. No change is necessary to control a machine tool having different spindle controller (driver).
- ***Data acquisition card***. The output produced by MotionGenerator does not depend on the data acquisition card configuration to which the machine tool is connected. Again, no changes are needed in the subsystem when different cards are used.
- ***Machine type***. Machines can be different in a number of characteristics. For instance, different machine types may have different axis amount, may be able to perform different canned cycles (if any), and interpret different instruction sets. Cutting tools can move only on perpendicular planes, as our

algorithms are two-dimensional, it is only necessary to inform an algorithm (via a parameter) about the axis that will intervene on the motion instruction. We need to have a different implementation of CannedCycle for each canned cycle type. It seems that we will need to have a different interpreter for each machine type. We circumvent this need by implementing an Interpreter that has the capability of handling instructions for most of the machines we have identified as amenable to be automated. Thus, what will really need to be changed is the Translator subsystem.

Our previous discussion shows how the architecture we suggest for MotionGenerator supports CNC systems' evolution. All needed changes can be localized inside one or more components, thus instantiating the proper subsystem will require to use the appropriate component implementation. In Figure 4.8, numbers on components' upper-right corners represent the amount of potentially different implementations that will be needed. For instance, we may need two different interpreters (one for EIA and other for ISO standards), six canned cycles, etc.

Note that a single MotionGenerator can have more than one implementation of some components. For instance, one machine can require performing both linear and circular trajectories, thus a corresponding interpolator for each type of interpolator need to be present in the CNC system.

4.5.2 MotionControl Subsystem

The MotionControl subsystem feeds instructions to machine elements and receives feedback from actuators to monitor machine operation. The input to MotionControl is the output produced by MotionGenerator (i.e., instructions to turn on/off spindle, coolant, etc.; instructions to change cutting tool; sequences of instructions defining a trace the cutting tool should follow, etc.). MotionControl uses these instructions to guide the tool machine to perform a specific machining operation.

Most of the complexity in MotionControl is that it must operate at real-time. There are inertial forces preset in the machining process. As a result, physical machine elements cannot respond to instructions at the speed the controlling computer can send the movement instructions. Once in motion, physical elements can respond to instructions in shorter time (i.e., response time is related to motion speed), thus the controlling software must consider such behavior.

This basic operation is complicated by the need to deal with user interaction and process monitoring. Feedback received from machine tool devices is sent to the user interface to keep users informed of the process' status. Besides the described operation, MotionControl should respond to user events such as pausing the process, canceling the process, and resuming the process.

Figure 4.9 shows MotionControl subsystem's internal hierarchical structure. The first layer is InstructionDispatcher, which handles user interaction and NC program execution and controls instruction dispatching to lower level layers. For instruction dispatching, the (expanded) input program is read one record (instruction) at a time; instructions are pre-processed and lower-level operations are requested, according to instruction's type. When the input file is exhausted,

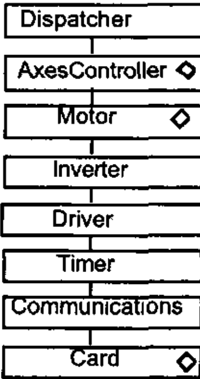

Figure 4.9 MotionControl subsystem

InstructionDispatcher moves the cutting tool back to its original position and then the system is ready to process a new NC program.

The next layer is AxesController; it has information of machine tool's configuration, which is determined by how many axis the machine tool contains. Upon request, it is able to move cutting tool to a pre-specified position (called the "tool home"). Movement instructions are routed to the corresponding motor, so the correct motor performs the needed steps in the expected direction.

Motor is the most complex layer. There exists a Motor component for each machine tool axis. Motor issues movement actions to the electric motor attached to the machine axis it controls, according to motor type (DC, AC, step, etc.). Movement distances come expressed in thousandths of an inch, thus Motor adjusts them according to motor type characteristics (e.g., for a stepper motor, distances are translated to steps, according to motor precision). Motor also encloses axis-positioning control; it keeps information of each movement instruction sent to a motor. Feedback on the physical motor's response to a movement request is received by Motor. If a motor was unable to perform the requested action, its controlling Motor component detects such fault and the movement instruction is re-sent to motor (two tries are attempted only). Additionally, each Motor component keeps track of cutting tool positioning along the axis it controls. Using this information and information on the working space dimensions, a Motor can know when a movement instruction will move the cutting tool away of the working area, and thus can refuse issuing the instruction and inform the user of the situation. To keep performance high, positioning information is displayed proportional to motor precision and instruction feed rate. When moving (i.e., feeding instructions) at high speed, rate of positioning display is slowed down, thus control can be devoted more to coordinate machine tool operations. When moving at slow speed, position is displayed for every instruction. Such behavior takes into account that low-speed oper-

ations are executed when complex movements or finishing operations are being performed, and the user needs to be informed of operation's progress. On the contrary, high-speed movements occur at the middle of a "long distance" movement instruction, thus intermediate positioning is not so important for the user. Whenever instruction-feed rate is changed, the display rate is adjusted accordingly. Finally, we remark the fact that Motor requests from a lower layer that the corresponding motor perform a step, but does not care about how long motor will take to execute the commanded action.

Inverter performs a simple task; it just computes equivalent voltage value (in the range between 0 and 10 volts) for the incoming rpm value. Voltage value calculation depends on the characteristics of the driver attached to the machine's spindle (i.e., the same driver can be used in different motors or the same motor with different drivers, but Inverter's calculations depends only on the driver).

Up to this point, no layer has taken care of the fact that signals (requests) are being sent to machine tool's physical components whose response time is much slower than the controlling-computer processor's speed. In practice, we need to implement a waiting mechanism to block execution while motors perform requested operations. This blocking mechanism should be flexible enough so the waiting time can be adjusted according to machining conditions. That is, when a motor is asked to move from a stationary position, it takes more time to react than when it is already in movement. Timer layer implements the waiting mechanism. When movement instructions are sent to a motor, Timer retains control (i.e., blocks) until the appropriate time elapses. The waiting mechanism has to be synchronized to motor reaction time. To work under such strict timing requirements, Timer synchronizes with motors using hardware interrupts. Typically hardware interrupts occur within tiny error limits (enough for the precision required by a

motor). Note that changes in instruction feed rate to motors work by changing the waiting time (i.e., the frequency at which interruptions are being generated).

Users are interested in monitoring the machining process; thus values showing machine tool status have to be dynamically displayed. These values have to be updated at each status' change (i.e., a motor is moved, the spindle is turned on/off, coolant is turned on/of, etc.). The user interface and controlling subsystems run on different processes, thus all data interchange uses an interprocess communication mechanism. The Communications layer performs all communications with the user interface. It decides when data has to be sent to the user interface process (according to motor speed, or when actuators are turned on/off). Note that communications are in two directions, from the user interface to the motion control subsystem to issue commands, and from motion control to the user interface to inform of executed tasks (users request the execution of a program, pause, resume, or cancel the execution of the current program; the motion control informs of actions started or new position of the cutting tool. At the user interface, users edit machine configuration (i.e., working area dimension, motor precision, etc.); such information has to be supplied to components in MotionControl. The Communications layer handles all communications associated with user interaction and machine configuration. The protocol used here is that before executing any operation, upper-level components check if Communications component has an incoming message from the user interface and process them.

The bottom layer is Card. By providing a standardized interface to higher level layers, Card hides hardware data acquisition card details. Each hardware engineer can decide at which pins control wires are attached, and then Card has to be programmed accordingly. Card is an abstraction of a hardware data acquisition card to which machine tool's elements are attached to the controlling computer.

Again, we need to analyze how the model can be adapted to implement domain variations. Here we consider the same variations that previously identified for CNC systems:

- ***Motor type.*** Previously we identified that there are at least four different motor types we would need to control: stepper, AC, DC, and linear. Each motor operates very differently, and our previous descriptions have been limited to stepper motors. For instance, stepper motors operate at discrete steps for each electrical pulse they receive. DC motors operate on voltage levels; according to the distance the motor has to advance, thus for DC motors it is necessary a translation to volts. AC motors operate on voltage values too, but the behavior (i.e., the computing algorithm) is different from that of DC motors. As a consequence, we will need a different Motor component whenever we use a different motor type.
- ***Motor precision.*** There is a direct relationship between the linear distance instructions in the NC program, and the angular distance the motor in the corresponding axis has to rotate. Then it is only necessary to know motor's precision (i.e., distance a motor moves at each step) to compute the total distance in terms of motion units. The precision value is normally fixed for long time periods (probably the whole life span of a machine tool), thus we decided to store it in a configuration file. If motors in a machine are substituted by others of different precision, it is only necessary to edit the configuration file to put the appropriate precision value.
- ***Spindle's controller.*** Spindle motors are AC motors, which can rotate at different speeds and support different workloads. Inverters change motor speed according to their input voltage values. Input voltages to inverters are typically in the range from 0 to 10, but there is not a constant linear among different inverters. So it will be necessary to use different algorithms to convert

from rpm to volts for different inverters. As a consequence, whenever we change one inverter by another, it will be necessary to use a different Inverter.

- *Data acquisition card.* All layers in MotionControl are independent of one another. It is clear that a mapping from domain components to model abstractions (components) exists for most of the cases (exceptions are Timer and Communications layers). The idea is that each software component implements the abstraction that its type defines. This way, when a different data acquisition card is to be used in a machine tool, we only need to use the Card component implementing that card's functionality. It should be noted that wiring in a particular data acquisition card is not fixed. Data acquisition cards are flexible and similar functionality can be implemented using very different wiring configurations. However, the implementation of a Card component is restricted to a particular card's wiring. This way, if different wiring configurations are desired for a single card type, it will be necessary to have a different implementation of Card component for each wiring configuration.
- *Machine type.* Main variabilities that machine tools can have are: axes' number, instructions they can perform, dimensions of the working space, and on all other factors mentioned above (i.e., have different data acquisition cards, motors of different precision, etc.). Note that most of these changes are similar to above-mentioned changes; distinct, however, is the use of a different number of axes. A change in the number of axes can be implemented easily by substituting AxesController and/or Card. We use an AxesController appropriate to the number of axes in the machine tool. It may be necessary to use a Card component that can control the number of axes in the new machine tool.

Again, changes in machine tools can be implemented by substituting a component by another implementing appropriate algorithms. Note that changes in machine tools impact more to MotionControl than they do to MotionGenerator. Certain changes don't have any influence on MotionGenerator (i.e., spindle's control and data acquisition card). However, all identified domain changes may influence at least one component in MotionControl. That is a consequence of how MotionControl is modeled; almost all MotionControl components / features have a direct counterpart in the machine tool domain. Then, it is natural that changing an element in the domain makes it necessary to perform the corresponding change in the composition.

We now turn to analyze constraints governing how components can be used in the domain model. First of all, layers have to keep the ordering shown in Figure 4.8.a and Figure 4.8.b. However, not all layers are mandatory for all machine tool configurations. Which compositions are mandatory and which aren't, is determined by configuration predicates. Table 4 and Table 5 show constraints

Component	Constraints
Expander	Interpreter defined below
Interpreter	Axis number consistent with machine type Canned cycle (if any) defined below
CannedCycle	Allow duplicates Not mandatory CoordAdapter defined below
CoordAdapter	Interpolator (if any) defined below
Interpolator	Allow duplicates Not mandatory Axis number consistent with machine type
Writer	Axis number consistent with machine type

TABLE 4. Constraints for components in MotionGenerator subsystem

for models in Figure 4.8 and Figure 4.9, respectively in a descriptive manner (rather than using logic predicates). Note the presence of constraints enforcing

component ordering: AxesController must be on top of Motor, Motor on top of Timer, etc.; other enforce implementation consistency: AxesController has to be consistent with machine's axis number (2 or 3); Motor should be consistent with motor type (AC, DC, step, etc.); Card should be able to support the number of axes in the tool machine.

Component	Constraints
Dispatcher	AxesController defined below
AxesController	Axes number consistent with MotionGenerator Motor component defined below
Motor	Driver component defined below Timer component defined below
Inverter	Card component defined below
Driver	Motor component consistent with driver
Timer	Communications component defined below
Communications	Card component defined below
Card	Supports motor type and axis number

TABLE 5. Constraints for components in MotionControl subsystem

4.6 GenVoca Models

In Section 4.5 we identified hierarchical models for CNC's systems. We presented descriptions of each layer in the models and how domain features are mapped to components. We described how the models can be adapted to different changes that are required to implement variations in CNC systems controlling machine tools with different characteristics. In this section, we show hierarchical models for the corresponding models presented in Section 4.5. We present type instances and show different CNC systems instances/compositions.

Straight from Figure 4.5 and Figure 4.6, we can derive corresponding hierarchical models shown in Figure 4.10 and Figure 4.11, respectively. Capitalized names represent types; lower case names are components. Note in Figure 4.10 that

```
Expander    = expander( x:Interpreter) }
Interpreter  = {interpreter( x:CannedCycle) }
CannedCycle = {cannedCycle( x:CannedCycle),
                finalCycle( X:CoordAdapter) }
CoordAdapter = {coordAdapter( x:Interpolator) }
Interpolator  = {interpolator( x:Interpolator), finalInt( x:Writer) }
Writer = {writer}
```

Figure 4.10 MotionGenerator's types and implementations.

the only types describing symmetric components are CannedCycle and Interpolator.

Figure 4.12 shows concrete instances for type declarations from Figure 4.10. Note in Figure 4.12 that we do not specify concrete parameter types, they are subject to compositions described in the corresponding model (see Figure 4.10).

We can now have instances of concrete MotionGenerator subsystems by composing components from Figure 4.13. For example, in the following equations, **mg1** and **mg2** are valid equations.

```
mg1 = expander( interpreterEIA274( roughCut(
       codeExpansion( linearInt( writer )))))

mg2 = expander( interpreterEIA274( roughCut( finishing(
   codeExpansion( linearInt( circularInt( finalInt( writer)))))))
```

```
InstructionDispatcher = { instrDisp( x:AxesController) }
AxesController = { axesController( x:Motor) }
Motor        = { motor( x:Inverter) }
Inverter      = { inverter( x:Timer) }
Timer        = { timer( x:Card) }
Card         = { card }
```

Figure 4.11 MotionControl's types and implementations.

```
Expander    = {expander (x:Interpreter) }
Interpreter  = {interpreterEIA274 (x:CannedCycle) }
CannedCycle  = {roughCut(x:CannedCycle), finishing(x:CannedCycle),
         peckDrilling(x:CannedCycle), finalCycle (x:CodeExpansion) }
CodeExpansion = {codeExpansion (x:Interpolator) }
Interpolator  = {linearInt(x:Interpolator), circularInt(x:Interpolator),
         finalInt(x:Writer) }
Writer  = {writer}
```

Figure 4.12 Component instances for MotionGenerator.

In the first example, **mg1** is a MotionGenerator containing an Expander component that sends instruction streams to an Interpreter implementing the NC programming language EIA274; there is a component implementing a rough-cut canned cycle; then a component to perform CodeExpansion; a component to perform linear interpolation, linearInt; finally comes a component implementing input/output operations to a file. In the second example, **mg2** implements a functionality similar to **mg1**. However, **mg2** implements a finishing canned cycle, and adds circular interpolation capabilities.

The following is an example of an invalid MotionGenerator subsystem:

```
mg3 = expander( roughCut( interpreterEIA274(
                linearInt( circularInt( finalInt(writer))))))
```

Note that an Interpreter component cannot be below a CannedCycle component (roughCut in the example), and that a CoordAdapter component (which is missing in the example) is mandatory.

```
InstructionDispatcher = {dispatcher (x:AxesController)}
AxesController    = {2AxesController(x:Motor), 3AxesController(x:Motor)}
Motor   = {stepMotor(x:Inverter), DCMotor(x:Inverter), ACMotor(x:Inverter)}
Inverter = {inverter(x:Timer)}
Timer   = {timer(x:Card)}
Card    = {pcCard, dcCard, simCard}
```

Figure 4.13 Component instances for MotionControl.

Figure 4.13 shows concrete instances of realm declarations for MotionControl model from Figure 4.11. Instances of MotionControl subsystems can be constructed composing components from Figure 4.13. In the following composition equation, mc1 defines a valid composition:

```
mc1 = dispatcher( 2AxesController( stepMotor(
                              inverter( timer( pcCard)))))
```

Composition equation **mc1** defines a MotionControl subsystem for a machine having two axes and whose axes are controlled by stepper motors, and uses a National Instruments' PC100 data acquisition card. Conversely, the composition **mc2** defined by the composition equation:

```
mc2 = dispatcher( stepMotor( inverter( timer(pcCard))))
```

is invalid, since an AxesController instance needs to be present on top of a Motor instance.

4.7 Compositional Implementation

In Chapter 2 we presented in detail our approach of component implementation in C++, it consists in defining a C++ parameterized class for each component, then compositions are expressed as template instantiation expressions.

Even thought every layer can contain many classes in its implementation, a model for CNC systems is better implemented by defining each layer as a single parameterized class. The GenVoca model presented in this chapter is similar to a model for avionics systems (see [BGCS95]) in that software components represent physical entities. Models for systems implementing physical entities seem to be better expressed by mapping domain entities in the model to features and directly to components in the implementation. This is an advantage, since it makes easy to swap physical components and perform the corresponding swapping on the software components.

In the previous section we described how each subsystem can be produced from a component composition. Still, these subsystems should be put together as described by Figure 4.7. A system that run on a single thread of control would include the two corresponding component compositions (an perhaps the systems itself would consists in a composition of the subsystems). However, a CNC system has to run in several threads in order to function properly (machining has real time characteristics). Given such restriction, the compositions defininig every subsystem are defined in a separate run time unit and executed in a separate thread. The main thread is the user interface, which is used to interact back and forth among the subsystems.

4.8 Design Wizard for CNC Systems

Previous section describes a model for CNC systems and its implementation. As noted, CNC systems developers have to write two different but highly related component compositions (i.e., composition equations), for each CNC system they want to construct (one composition for MotionGenerator and other for MotionControl subsystems).

Our prototype implementation of a design wizard for CNC systems is called CNCgen. Figure 4.14 shows CNCgen's graphical specification interface. In CNCgen, each domain feature consists in a developer selectable option in the graphical interface. Developers specify features by selecting from features supported (i.e., components already implemented), those he/she wants the target CNC system to implement. Once a feature set has been chosen (the configuration of an application specified), pressing a button generates composition equations for both

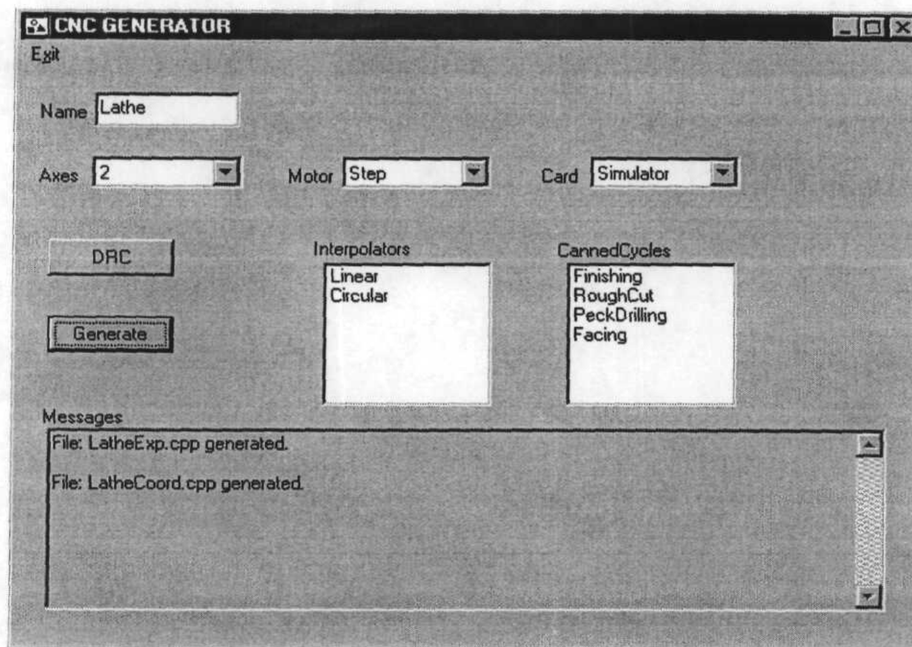

Figure 4.14 CNCgen's user interface

subsystems (MotionGenerator and MotionControl), and then performs composition equation validation; results are shown in a Messages window. If everything was correct, the developer may press a button to generate the CNC system implementing specified features[1]. It is only necessary to compile the application for having a CNC system.

CNCgen uses the approaches for component implementation and configuration verification already described in Chapter 2. The only new elements are the user interface and the generative engine. It is worth noting that such simplicity is the result of our model's carefully planned design and implementation. Arriving at a so clean design requires several iterations and the instantiation of a number of the potential application systems. Our prototype has been tested in two ways. First, we have retrofitted a turning machine. Second, we implemented a simulator for different machine types.

It was fairly easy to implement a machine tool simulator. The only element in direct contact with the machine tool, sending signals and receiving feedback from it, is the Card component (see Figure 4.9). Thus we could emulate a machine tool by implementing a simulator Card, which was able to control two- and three-axes machine tools. Our simulated machines may have different Motor type components, Inverters, Timers, etc. The user interface for CNC systems is reconfigurable at run-time. At the initialization step (when the system is loaded and executed), the user interface exchanges configuration information with the MotionGenerator and MotionControl subsystems to reconfigure itself to display data relevant to the machine tool (e.g., positioning in two or three axis depending on the machine tool axis number, etc.).

---

1. In C++, the weaving code consists in specifying includes for files implementing components, and writing the necessary template compositions.

4.9 Discussion

A CNC system is a hard real-time application (i.e., clock's frequency has to be constant so no jamming is produced in the tool machine thus the working piece has a terse finishing [KS97]). That aspect didn't represent a problem, mainly because motion speed of physical components is rather slow as compared to current PC clock speeds. Implementing efficient code was enough to meet timing requirements.

For this example, it is remarkable to note that simplicity of the configuration wizard in Figure 4.14 is apparent, the code of a CNC application is about ten thousand source code lines. Because of its complexity and size, we consider this our most complex example of a configuration wizard. From a single specification, two subsystems (equations) are produced. These subsystems run on separate processes and communicate using operating system communication primitives (pipes and triggers[2]).

The implementation language already supports the parameterization mechanisms our approach uses, thus no extensions were necessary.

2. A pipe contains messages of varying sizes. A trigger is a signal sent to a process to inform it of a certain event.

Chapter 5

Credit Unions Product-Line

This chapter describes the third example of applying our proposed approach to implement a software product line. The programming language used in this example is Object Pascal, which does not directly provides a template-like parameterization mechanism, thus as we did for Java, here we propose Pascal extensions to support wizlet implementation.

5.1 Credit Union Management

A credit union is a cooperative, non-for-profit financial institution organized to promote thrift and provide credit to its members. Credit union members are provided with a safe, convenient place to save and borrow at reasonable rates.

A credit union is member-owned and controlled through the election of a board of directors drawn from membership. Membership is not open to the general public. Instead, it is limited to persons sharing a common bond of occupation, community, or association. To join a credit union, potential members must be first eligible under the common bond provisions, and submit a membership application [Umh01].

Credit unions are not typical financial institutions and thus are managed differently and are subject of special control by the federal government[1]. In this

Chapter we present a product-line approach for administrative financial reports that credit unions require.

Credit unions are similar to banks (i.e., both offer financial products and services to consumers). As cooperative organizations, credit unions exist solely to meet their members' financial needs, not to make a profit off of them. Credit unions were created to enable small business to pool their financial resources to help themselves and each other. As a consequence of its success in low cost loans, membership continues to increase[2].

Credit unions are non-for-profit, member-owned, democratically controlled financial cooperatives. In particular, in this Chapter we concentrate on credit unions whose members are small-business owners.

Accounting information in credit unions can be used for two purposes: managerial or financial. *Management accounting* is concerned primarily with financial reporting for internal users, in particular for managers. *Financial accounting* usually is addressed to external users, mainly for funding purposes or government information. Accounting application systems usually are designed to produce information for both internal and external reporting.

The accounting process consists of two phases (see Figure 5.1): the recording phase and the reporting phase. The *recording phase* is concerned with collecting information about economic transactions and events and distilling that information into a useful form. In the *reporting phase*, the recorded information is organized and summarized, using various formats for a variety of decision-making purposes [SSS00].

1. Rules describing how credit unions are managed are limited to Mexican credit unions.
2. Just to have an idea of the number of credit unions after 67 years after the first was established, in the United Sates of America -as of January 31, 2002- there are more than 10,850 serving more than 76.7 million members [CUO02].

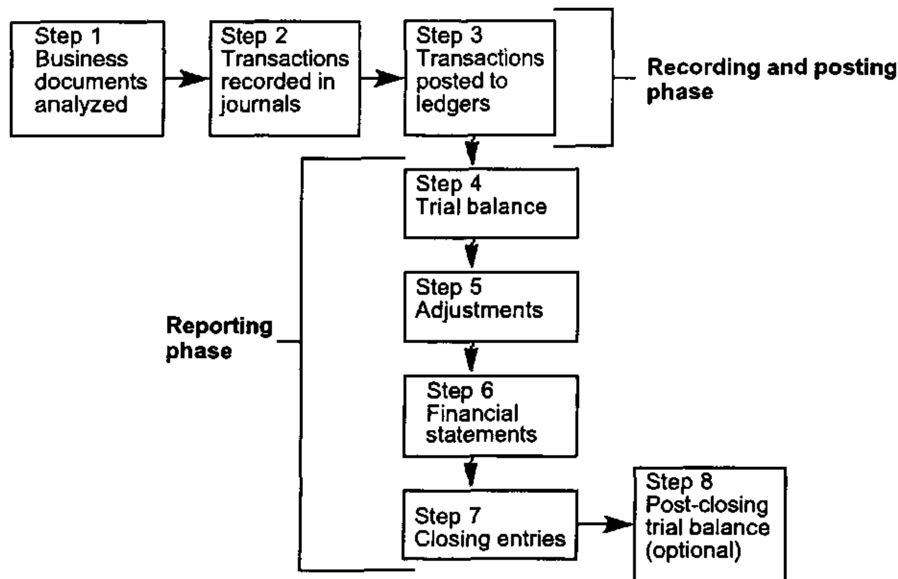

Figure 5.1 : The accounting process.

Several terms commonly used in the accounting domain are [WKK93]:

- ***Account***: individual accounting record representing an increase or decrease in a specific asset, liability, and equity item.
- ***Ledger***: keeps the entire group of accounts that a financial entity maintains. If the financial entity has more than one ledger, a general ledger concentrates all the assets, liabilities, and owner's equity accounts appearing in the financial statement.
- ***Journal***: records transactions in chronological order before being translated to accounts. For each event accounts involved are identified and affected accordingly (with an increase or decrease).
- ***Posting***: is the operation of transferring (summarizing) information in journals to appropriate accounts in the ledger.
- ***Transaction***: events involving the transfer of goods or services between two or more entities[3]. A transaction produces several records in a journal.

- ***Accounting rules***: specify how accounts are integrated to build the general ledger.

5.2 Static parameterization in Object Pascal

As can be observed in the previous section, major requirements in a credit union's accounting product line are information recording and reporting. Borland's Delphi was chosen as development environment because it provides facilities for graphical user interaction, fast report preparation, and a set of database drivers.

Delphi's underlying programming language is Object Pascal which fully supports object orientation, thus our component-based product line approach is applicable. As is the case with Java, Pascal[4] does not support a template-like parameterization mechanism, similar to that of C++. This section shows our proposal for extending Pascal[5] to support parameterization required by wizlets. A few concepts on Pascal programming are presented to better explain our extension mechanism for wizlet parameterization in Pascal.

The equivalent to a software module in Pascal is a *unit*. A *unit* can implement (and export) several classes and other programmer defined types. A unit is divided in two parts: an interface section and an implementation section. The unit's interface section contains declarations that the unit can export to (be accessed from) other units. The *implementation* section contains a unit's internal details (local or private class declarations and method implementation). A *uses*

3. Note that here *transaction* is a business transaction, not a database transaction.
4. We limit our discussion to the Pascal implementation provided by the Delphi environment from Borland International. However, at the time of this writing there is not an ANSI standard for template-like extensions to Pascal.
5. In this chapter we will refer to Object Pascal simply as Pascal, as we are not interested in describing similarities or differences between Object Pascal and other Pascal dialects.

statement followed by a list of unit names, allows accessing the elements exported by other units. To inherit from a class defined in a different unit, the name of the super class is specified inside parenthesis in the line declaring a subclass, with the scope operator (a point) indicating the unit where that class is defined. The following code segment declares ThisUnit as a new unit, which imports the elements exported by an already existing unit OtherUnit:

```
unit ThisUnit;
interface
uses OtherUnit;
type
  Inner1 = class(OtherUnit.Inner1)
    ....
  end;
  Inner2 = class(OtherUnit.Inner2)
    ....
  end;
  Inner3 = class(OtherUnit.Inner3)
    ....
  end;
implementation
  // component's local class declarations and
  // method implementation
end.
```

The code shows the declaration of three inner classes (Inner1-3), as extensions of classes that already exist in the OtherUnit unit).

As was pointed out earlier, Object Pascal does not provide direct support for template-like class parameterization. However, a similar approach to that we used to extend Java in Chapter 3 can be implemented to extend Pascal, and thus parameterized wizlets can be implemented. As was explained and shown in the previous code segment, the encapsulation entity in Pascal is the `unit`, thus we simply need to parameterize units. Using similar tags to those we used in Java, a generic wizlet declaring an unknown (super) wizlet from which a specific wizlet inherits can be implemented using Pascal units, as the following code shows:

```
unit UWizlet;
interface
uses U<<WizletSuper>>;
type
Inner1 = class(U<<WizletSuper>>.Inner1)
  ....
end;
Inner2 = class(U<<WizletSuper>>.Inner2)
  ....
end;
Inner3 = class(U<<WizletSuper>>.Inner3)
  ....
end;
implementation
// component's local class declarations and
// method implementation
end.
```

To emphasize the parts in which an identifier refers to a unit name, in the code we have used a U prefix[6]. The code can be easily translated to our previous example of a Pascal unit. To describe how this extension mechanism works, we turn to implement the same code we implemented to show how Java extensions work. If necessary, refer to Section 3.2 for a more detailed discussion of the example and to compare the similarities between Java and Pascal extensions.

In the example, we want to be able to show information (a report) in both the screen and in print. Figure 5.2 shows how software modules used in the example are related in a module hierarchy.

With the proposed Pascal extension, a Report component that will inherit from a device component containing three inner classes, can be implemented as:

```
unit UReport;
 interface
 uses U<<WizletSuper>>;
 type
```

6. In Pascal's programming realm it is common to prefix a unit's name with a U, to characterize the name as a unit's identifier. The code we implement here follows such approach, just to keep consistency with what is considered the "norm" in Pascal programming.

```
 Inner1 = class(U<<WizletSuper>>.Inner1)
  .... //class declarations
 end;
 Inner2 = class(U<<WizletSuper>>.Inner2)
  .... //class declarations
 end;
 Inner3 = class(U<<WizletSuper>>.Inner3)
  .... //class declarations
 end;
 implementation
 // Report's local class declarations and
 // method implementation
end.
```

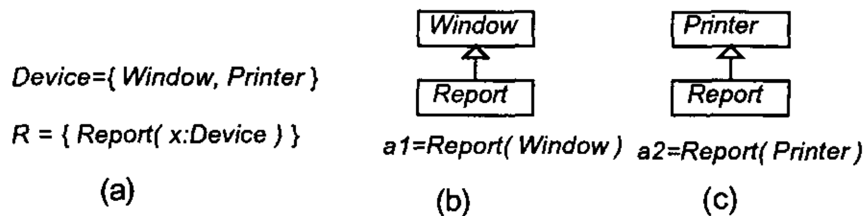

Figure 5.2: Type declarations and composition equations.

The implementation of Window as a final (top-most) component in a component inheritance hierarchy is:

```
unit UWindow;
interface
type
Inner1 = class
 ....
end;
Inner2 = class
 ....
end;
Inner3 = class
 ....
end;
implementation
```

```
// Window's local class declarations and
// method implementation.
end.
```

Suppose we have an implementation of a Printer component, which is similar to Window but sends report's results to a printer. From these implementations, we can build composition equations for systems that display reports in a window in the screen or print them on paper, as follows:

Sys1 = Report (UWindow)

Sys2 = Report (UPrinter)

Note that Report cannot be directly compiled; first it should be adapted to every composition in which it is used, by instantiating its super wizlet. For this we use the corresponding composition equation. For example, a Report's version that can be compiled to obtain Sys1 is shown in the following code, in which the parameter has being substituted (by UWindow component):

```
unit UReport;
interface
uses UWindow;
type
Inner1 = class(UWindow.Inner1)
 ....
end;
Inner2 = class(UWindow.Inner2)
 ....
end;
Inner3 = class(UWindow.Inner3)
 ....
end;
implementation
// Report's local class declarations and
// method implementation
end.
```

As we did in Java, we can use a composition equation to find the necessary substitutions. In the previous example, the only necessary substitution was to change every instance of <<WizletSuper>> by the corresponding parameter name from the composition equation; the places where substitutions are to be made are marked with <<WizletSuper>> tags. To emphasize the use of Pascal's units, we decided to put a U prefix before a unit name, we think in this manner it is easier to see where we are referring to a unit instead of a class. However, such naming scheme is not a Pascal's characteristic that needs to be followed.

Such substitutions can be performed manually using a simple text editor, and resulting Pascal units can be compiled into an application. As discussed, we don't want to perform component adaptation by hand, using text editors. We suggested the use of preprocessors[7] that parse the composition equation and perform necessary adaptations to each wizlet. To operate, such tool will need as input the application specification from which a composition equation is generated, and the corresponding Pascal source code for the application produced. In the following sections we describe accounting systems for the credit union domain, domain models, and implement the infrastructure for a corresponding product line.

5.3 Domain Model of Accounting Systems

Previously we explained that financial statements are reports that need to be produced by an accounting system. The three major financial statements that accounting systems produce, are [LC96]:

7. In fact, we implemented the same preprocessor for Pascal that we used for Java.

- The *balance sheet*: reports, as of a certain point in time, the resources of a company (the assets), the company's obligations (the liabilities), and the net difference between assets and liabilities, which represents the owners' equity. Other name given to this statement is *general balance*, which is the name we use in the following discussions.
- The *income statement*: reports, for a certain time interval, the net assets generated through business operations (revenues), the net assets consumed (expenses), and the difference, which is called net income. The income statement is the accountant's best effort at measuring the economic performance of a company.
- The *statement of cash flows*: reports, for a certain time interval, the amount of cash generated and consumed by a company. Credit unions don't use this type of statement.

Other statements and operations particular to credit unions are:

- *Federal reports*: reports required by federal government.
- *Optional reports*: responsibilities, financial information, etc.
- *Posting*: particularly for credit unions is a monthly and yearly operation which leaves accounts prepared for the next accounting period.

As previously stated, to describe how financial statements and reports are built from journal entries, accounting rules are used. These rules specify if a recorded amount in a transaction is to be added to or subtracted from a group account. Other accounting rules specify which grouping accounts appear in every financial statement.

In general, requirements in the accounting domain for credit unions can be grouped into mandatory and optional operations. *Mandatory operations* are those related to information recording, all accounting systems have similar recording necessities. *Optional operations* mainly consist in reporting operations, different

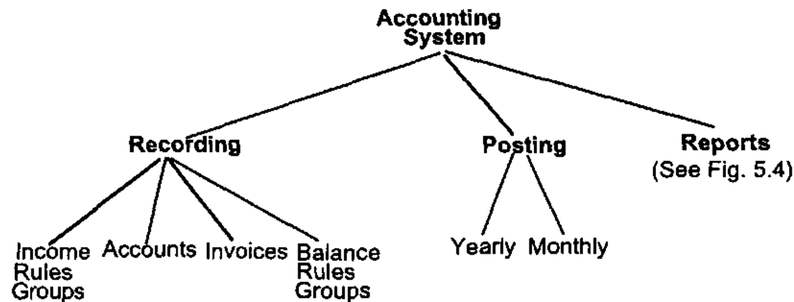

Figure 5.3 : Feature diagram for general ledgers

unions may need different reports, and a particular union may be interested in a special report that is not used by others.

Operations that fall inside each classification of accounting operations are:

- *Mandatory operations*: journal recording, accounting rules and groups recording, invoice recording, monthly and yearly posting.
- *Optional operations*: official reports, auxiliary reports, special reports, and optional reports.

A feature diagram for accounting systems in the credit unions domain is depicted in Figure 5.3 and Figure 5.4. Figure 5.3 shows features associated to recording and posting, Figure 5.4 shows features involved in reporting operations.

Figure 5.3 shows operations associated to recording and posting activities. Groups of accounts are defined to integrate financial statements, with rules specifying how transactions in accounts will be posted (summarized) to accounts. Invoices contain information about one or more accounts. Posting is performed in a monthly or yearly basis, thus (monthly and yearly) statements can be prepared and reported.

Figure 5.4 shows reports the six possible classifications of accounting reports: analytic, auxiliary, cross reference, special, optional, and official. Analytic

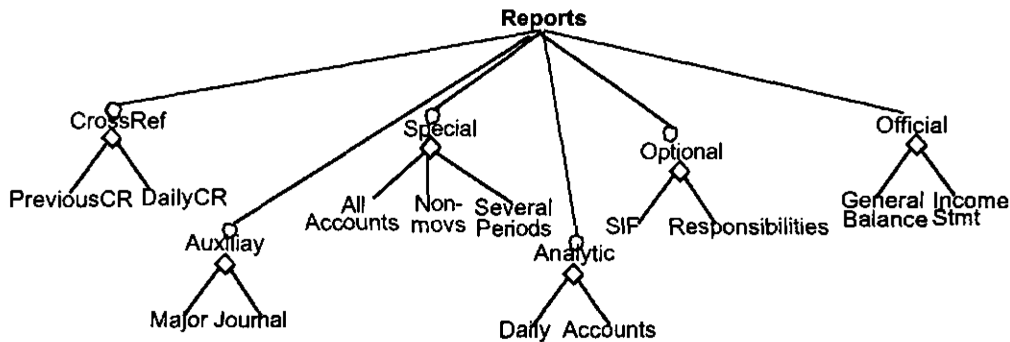

Figure 5.4 : Feature diagram for reports

reports show the how business operations relate to specific accounts. Cross-reference reports show the balance previous to posting or how daily transactions for every account were recorded. Special reports can include all the accounts, accounts for which transactions weren't recorded, or balance for several periods (months). Optional reports include reports that are required by the federal government. Official reports include a general ledger and an income statement.

The feature diagram makes evident the constraints. All constraints refer to the possible presence or absence of a component in an accounting system.

- All recording features should be in an accounting system.
- All posting methods are necessary in an accounting system.
- All official reports are required in an accounting system.

5.4 Domain design

It is impractical to describe the diversity of operations in an accounting system for credit unions in a single hierarchical structure. As we did for the example of CNC systems presented in Chapter 4, separate subsystems are constructed for every major subsystem in an accounting system using different wizlet compositions. Figure 5.5 shows the general architecture of an accounting system. An

accounting system is constructed from a user interface and a subsystem for every major set of operations. Each subsystem is constructed by a different wizlet composition, and interconnected to a user interface for the accounting system: a recording subsystem, which encloses recording activities, a posting subsystem for operations associated to posting, and a reporting subsystem which contains all reporting operations.

Each one of the three subsystems is defined by a separate hierarchy, the corresponding hierarchies are shown in Figure 5.6.

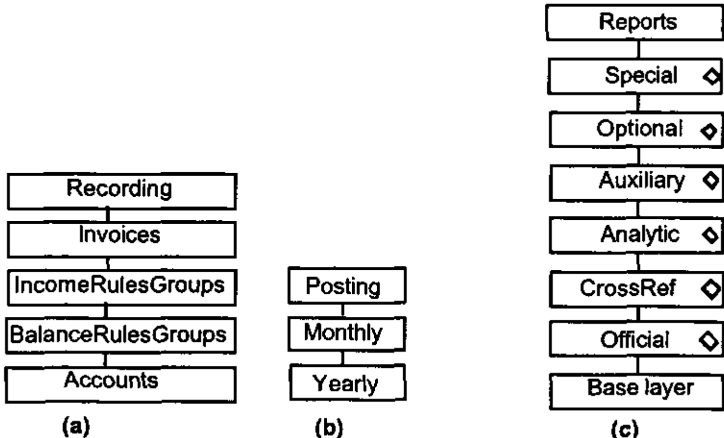

Figure 5.6 : Hierarchical model for general ledgers

The recording hierarchy in Figure 5.6 includes layers implementing the recording operations for the different accounting concepts. The Recording layer offers a general interface for recording operations, depending on accounting transactions being recorded, it communicates directly with the corresponding layer at a lower level, a user may want to add a record for a single account, or a group of

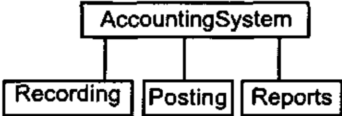

Figure 5.5 : Accounting systems architecture.

accounts, or define rules which involve several accounts, or recording an invoice, which includes at least two accounts (e.g., the account from which money was borrowed, and the client's account to who the money was lend). When an invoice needs to be recorded, the is used Invoices to record the associated data. When rules or groups are to be defined, Recording communicates with IncomeRulesGroups or BalanceRulesGroups component, depending on if rules or groups for the income statement or the general balance are to be defined, respectively. A group of accounts define accounts that are associated, thus related accounts can be summarized together (for instance, all accounts related with expenses can be grouped even thought they can refer to different types of expenses).

Another component hierarchy shown in Figure 5.6(b) is Posting. Posting is performed by adding or subtracting the amounts in an accounting transaction to accounts involved in the transaction. Posting can be performed in a monthly or yearly basis, and the appropriate posting component used in every case.

A third component hierarchy shown in Figure 5.6(c) include all reporting components. The layer at the bottom implements the interaction with a database to extract accounting information necessary to produce the different reports. Most of the reports correspond to presenting the recorded information in different ways, by using different account groups, and totalizing accounts involved in every report. A report is defined by the set of accounts involved.

Following is the type and instance declaration of accounting systems for credit unions. This information was produced using both the feature and hierarchical diagrams from Figure 5.3 and Figure 5.4.

```
Recording = {accounts( x:Recording ),invoices( x:Recording ),
            balanceRulesGroups( x:Recording ),
            statusRulesGroups( x:Recording ), finalRecording}
Posting = { monthlyPost( x:Posting ), yearlyPosting( x:Posting ), finalPosting }
Reports = {reports( x:Special )}
```

```
Special = { allAccounts( x:Special), nonMovements( x:Special ),
            severalPeriods( x:Special), finalSpecial( x:Optional) }
Optional = { sif( x:Optional), responsibilities( x:Optional ), finalOpt( x:Auxilary)}
Auxiliary = { major( x:Auxiliary ), journal( x:Auxiliary ), finalAux( x:Analytic)}
Analytic = { accountsA( x:Analytic ), dailyA( x:Analytic ), finalA( x:CrossRef)}
CrossRef = { previousCR( x:CrossRef ), dailyCR( x:CrossRef ),
            finalCR( x:GeneralBalance)}
Official = { generalBalance( x:Official ), incomeStatement( x:Official ),
            finalO(x:BaseLayer)}
BaseLayer = {baseLayer }
```

The following section describes in more detail the internal structure of each component and how the hierarchical dependencies are translated to class inheritance dependencies.

5.5 Domain implementation

In Section 5.2 we presented the extension notation to describe component parameterization using Pascal's units. The implementation is tightly related to an internal view of the component, Figure 5.7 shows an inheritance hierarchy describing the inner classes from every components and how they extend other classes in the hierarchy. Class Report in every layer handles the interactions necessary to produce every particular report. Classes Accounts, Groups, Rules and Invoices implement the operations necessary to extract the information every report should produce. The base layer implements the interaction to the database storing accounting information.

Contrary to the examples presented in previous chapters of this dissertation, most of the components in Reporting have several (inner) classes. One of the more simple layers is Journal, which is of type auxiliary and produces a report containing all accounts with values posted for the time period being reported. Jour-

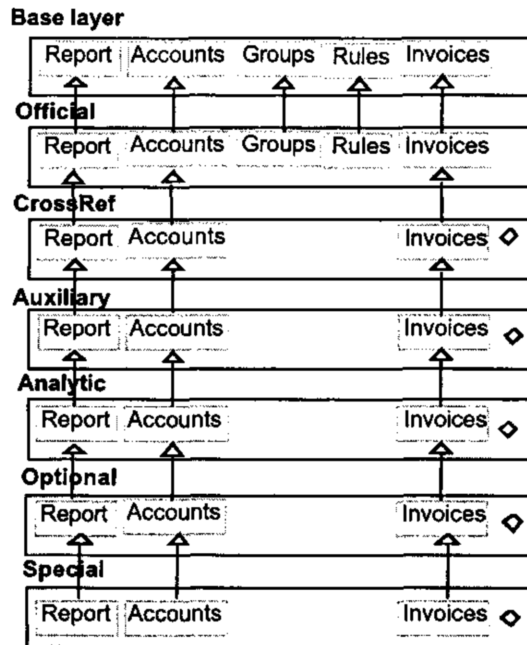

Figure 5.7 : Class hierarchy of accounting systems

nal has the structure described in the code that follows. Note that a U (which stands for 'unit' is added as a prefix to the unit's name) and an internal class has the wizlet's name (e.g., UJournal is a unit containing Journal's implementation); in this way, all references to a unit (such as in the uses clause) is prefixed by 'U'.

```
unit UJournal;//Auxiliary report
interface
uses U<<WizletSuper>>; //bring inner classes into context
type //new declarations go from here until 'implementation' section
Report = class (U<<WizletSuper>>.Report)
    procedure init; override; //init() is being overloaded
    procedure journalReport(Date date1, Date date2);
    procedure finish; override;
end;
Accounts = class(U<<WizletSuper>>.Accounts) //Extend supper class
   procedure init; override;
   Account function retrieve(string accountID); override;
```

```
  Invoice = class(U<<WizletSuper>>.Invoice) //Extend supper class
     procedure init; override;
     procedure first; override;
     procedure next; override;
  end;
  implementation
//Implementation uses syntax: class.function_name
  procedure Report.init
  begin
      inherited init
      ....
  end;
  procedure Report.journalReport(Date date1, Date date2)
  begin
   .... //here goes report's implementation
  end;
  procedure Report.finish
      inherited finish; //close all upper wizlets
      ...
  end;
  procedure Accounts.init
  begin
       inherited init
      ....
  end;
  Account function Accounts.retrieve( String accountID)
  begin
   .... //here goes retrieve's implementation
  end;
  end.
```

In the previous code, we can see that there are several references to a upper wizlet (enclosed inside <<...>> pairs), all these references will be substituted by the corresponding wizlet name at composition / generation time. Also note the use of declarations *override* and *inherited*. An *override* declaration informs the compiler that the subclass is substituting a method of the same name that exists in its superclass (sometimes called function overloading). To invoke the overloaded method, the *inherited* declaration is used.

Following is a partial implementation of the wizlet for Daily, a report of type CrossRef-erence.

```
unit UDailyCR; // CrossReferece report
interface
uses U<<WizletSuper>>; //bring unit's classes into context
type //new declarations go from here until 'implementation' section
Report= class (U<<WizletSuper>>.Report) //Declare class
    procedure init; override;
    procedure dailyReport(Date date1, Date date2);
    procedure finish; override;
end;
Accounts = class(U<<WizletSuper>>.Accounts) //Extend supper class
   procedure init; override;
   Account function retrieve(string accountID); override;
end;
Invoice = class(U<<WizletSuper>>.Invoice) //Extend supper class
   procedure init; override;
   procedure first; override;
   procedure next; override;
end;
 implementation
//Implementation uses syntax: class.function_name
procedure Report.init
begin
    inherited init //execute init in parent class
    ....
end;
procedure Report.dailyReport(Date date1, Date date2)
begin
 .... //here goes report's implementation
end;
procedure Report.finish
   inherited finish ; //finish all upper wizlets and...
    ...//perform other local finishing tasks
end;
procedure Invoice.init
begin
    inherited init
    ....
end;
procedure Invoice.first
begin
 .... //retrieve first invoice
end;
procedure Invoice.next
begin
 .... //retrieve next invoice
end;
procedure Accounts.init
begin
    inherited init
    ....
end;
```

```
Account function Accounts.retrieve( String accountID)
begin
 .... //here goes retrieve's implementation
end;
end.
```

The similarity of implementation in these two examples helps to clarify how our extensions to support unit parameterization work. As was the case with Java, the only substitution required is the name of the component that is being imported. These substitution can be performed using the composition equation as a guide. Again, such substitutions can be performed by hand, but to simplify this task and ensure that semantic consistency is preserved in the composition, we prefer to use a configuration wizard. A graphical user interface for a configuration wizard of accounting systems for credit unions is shown in Figure 5.8. An account-

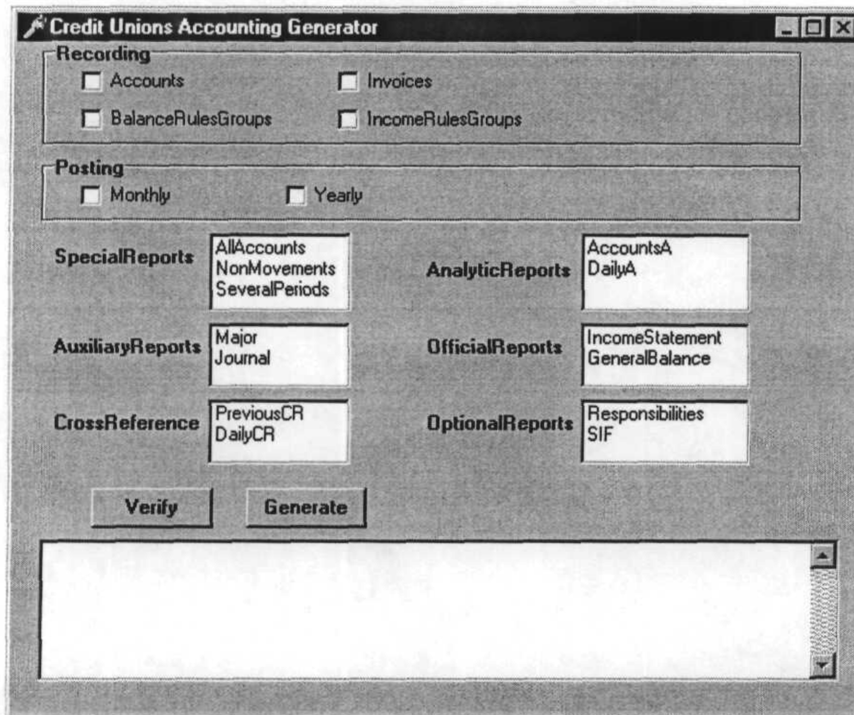

Figure 5.8: Specification interface for accounting systems

ing system consists of several windows (i.e. for data input and reporting), as our purpose is to describe how systems are generated, we do not show the windows that define the different accounting systems that can be generated.

As described at the end of section 5.3, the only composition constraints we need to specify are the necessity of particular components in a composition. For instance, all components in the Recording category are mandatory; however, most of the reports are optional. We described in previous chapters how these kind of constraints can be implemented and verified. Pascal does not impose any limitation on verification, thus we do not consider necessary to show specific implementations of the verification code in Pascal.

5.6 Discussion

In this chapter we presented an example of a product line of accounting systems for credit unions implemented as a configuration wizard. Comparing extensibility approach for Object Pascal as explained in this chapter with that presented for Java in Chapter 3, we see little differences. As discussed in Chapter 2 and Chapter 3, the similarities between the extensibility approach used in Java and Object Pascal is not accidental, but instead we planned the mechanism to be useful across programming languages.

The example presented in this chapter is important in two ways: first it helps to analyze the applicability of our approach in the informations systems area, second it allows to analyze how our extensibility approach works for different programming languages which include additional encapsulation mechanisms besides classes. We show how units encapsulating several classes and providing facilities for interface declaration are well suited to be extended for supporting wizlet imple-

mentation. Such finding is important because units are similar to modules in another programming languages (e.g., Modula, Ada, and VisualBasic).

Chapter 6

Product-Line Evolution

Applications evolve to adapt to changes in their requirements, incorporate new requirements, or eliminate unnecessary requirements[1]. A similar case occurs for product lines, changing requirements for a product line makes necessary that equivalent changes be made to infrastructures used to produce these product lines. In this chapter we discuss product line evolution in general, and infrastructures to gracefully evolve configuration wizards implementing product lines in particular.

6.1 Evolution in product-lines

The relationship between product line evolution and their product line infrastructures is depicted in Figure 6.1, which shows how as product line infrastructures evolve, they are capable of producing different product lines. Applications incorporating the new requirements that can be satisfied by a product line, can be synthesized from the product line infrastructure. For instance, Figure 6.1 shows that product line 1 can be produced from the product line infrastructure 1. To implement the evolution required by product line 1, thus product line 2 can be

1. To differentiate software evolution from software maintenance, we consider that software maintenance includes activities for fixing errors. In contrast, *software evolution* consists in improving the existing algorithms (e.g., for performance), adding new features, or removal of existing feature implementations.

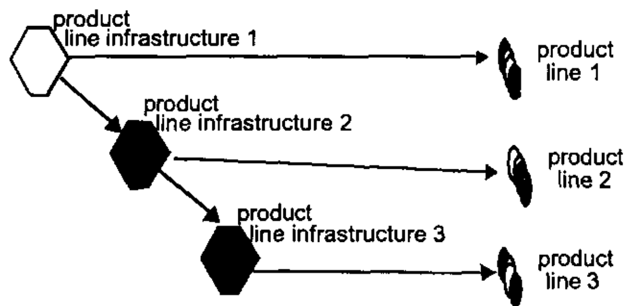

Figure 6.1 Product-lines can evolve into new product-lines

produced, the product line infrastructure 1 is evolved to product line infrastructure 2. Subsequent evolution in the product line will require corresponding evolution in the product line infrastructure.

Product line evolution has been investigated and different categories identified [SGB2001]. These categories can be characterized using features as follows:

1. New implementations of existing features: new feature implementations (i.e., specializations or sub-features) are added to a feature.
2. New features: completely new categories (e.g., non-root feature nodes in a feature diagram) are added.
3. Delete existing features: one or more sub-features are removed.
4. Feature diagram restructuring: relationships among components change. This type of evolution can be the most complex to implement, as it may require many changes in the feature diagram.

The way in which these variations are incorporated into product line infrastructures depends on how domain analysis and their corresponding designs are represented. In previous chapters we explained how our approach for configuration wizards use feature diagrams and hierarchical architectures to represent a domain analysis and its design, respectively. The discussion presented in the fol-

lowing sections is limited to show how product line evolution can be represented in the diagrams on which our approach is based[2].

Note that for an analysis model (i.e., a feature diagram), evolution category 1 (with respect to the previous list of evolution categories) can be represented by adding the corresponding (specialization or sub-) feature to the generic feature. An evolution category of type 2 can be represented by adding a new non-root node to the feature diagram. An evolution category 3 requires eliminating the corresponding feature from the feature diagram. Finally, an evolution category 4 would need to reorganize the feature diagram. How the corresponding architecture (i.e., the hierarchical) diagram has to be modified for every case depends on the semantics of affected features. The following sections describe how evolution can be implemented in configuration wizards, which are tools resulting from feature and hierarchical diagrams.

6.2 Specifying product-line evolution

To automate the implementation of the evolution categories described in the previous section, it is necessary to design a way to easily describe the changes we want to perform in the product lines, thus their product line infrastructures can be adapted. Along this dissertation our unifying concept had been that of "feature". We use feature diagrams as the result of domain analysis, and GenVoca hierarchical diagrams to represent domain designs. In or use of GenVoca, every component has a one-to-one correspondence to a feature, thus using features as units of specification would simplify the description of composition equations. In order to propose a specification notation, we need to consider how features are used in our

2. Part of the contributions of this work is to show how variability can be dealt with when software product lines are implemented by the combination of feature diagrams, GenVoca diagrams, and configuration wizards.

approach of configuration wizards. For every feature, we want to be able to display (or hide) the feature name in the developer GUI so it can be included in applications. For complex applications, we also may need to be able to generate more than one composition equation (sub-system) from a single feature diagram. Thus a specification needs to emphasize the selectability property of a feature: if it should be displayed in the developer's GUI, if one or several sub-features from one feature could be selected, etc., and it also needs to specify the composition equation in which the corresponding component can be included. Concretely, a *feature specification expression* consists of three elements describing a feature: a composition equation name, a property indicating the selectability characteristics, and a list of the possible sub-features for the feature. The general form of a feature specification expression is as follows:

feature = (composition, selectability, {sub-feature1, sub-feature2,...})

In this feature specification expression, feature is the name of the feature being described, composition specifies the name of the composition equation (e.g., system or sub-system) in which the sub-features of feature can be used, selectability is an attribute describing how the feature or its specializations (or sub-features) will be presented in a GUI to the application developer, and sub-feature(i) represents every sub-feature for the feature being described.

For example, the specification expression

f1 = (c0, s1, {f11, f12, f13 })

specifies that feature f1 can be used to construct composition equations named c0, will be presented as described by selectability property s1, and has sub-features f11, f12, and f13.

As a simple example, consider the case of the feature diagram in Figure 6.2. Applying the defined syntax for feature specification to the feature diagram in Figure 6.2, produces the following specification:

f0 = (c0,s0,{f0}) [3]

f1 = (c0, s1, {f11, f12, f13})

f2 = (c0, s2, {f21, f22, f23}) (6.1)

f3 = (c0, s3, {f31, f32})

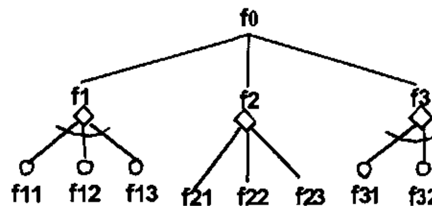

Figure 6.2 Initial feature diagram.

According to this specification, all components implementing features from Figure 6.2 will participate in compositions whose name is always c0. Although a different selectability property is specified for every feature, more concrete examples described lately in this chapter will show that the possibilities are limited.

Using specification (6.1), evolution categories discussed above can be represented by adding or removing sub-features from the corresponding specification expressions, adding a completely new specification expression for a new feature, etc. For example, the inclusion of a new specialization to feature f1 (lets say f14) can be done by adding its name to f1, thus the specification expression for f1 is now:

f1 = (c0, s1, {f11, f12, f13, f14})

3. Note that the specification does not correspond to the structure of the feature diagram. Instead, what we are implicitly describing in the specification is the layered architecture of the application family, not the feature tree. For every layer in the architecture, we are specifying the possible implementation components we can choose from at application generation time. However, the feature diagram is helpful because it clearly shows the sub-features of every feature and the selectability property of every non-root feature (i.e., if it is mandatory, single selection, etc.)

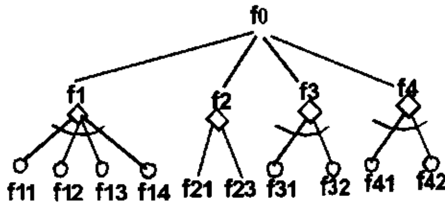

Figure 6.3 Simple feature diagram.

Similarly, the deletion of a specialization, lets say f22 from feature f2, would give a specification for f2 as

f2 = (c0, s2, {f21, f23})

The addition of a completely new feature, lets say f4 whose sub-features are f41 and f42, is represented by creating a new feature specification expression describing the new feature and its sub-features as follows:

f4 = (c0, s4, {f41, f42})

Which describes a new feature f4 with selectability property s4 and having sub-features f41 and f42.

Figure 6.3 shows the feature diagram after these changes have been done to f1, and f2, and the new feature f4 added. The following is the complete resulting specification after these changes:

f0 = (c0, s0, {f0})
f1 = (c0, s1, {f11, f12, f13, f14})
f2 = (c0, s2, {f21, f23, f24}) (6.2)
f3 = (c0, s3, {f31, f32})
f4 = (c0, s4, {f41, f42})

Specification 6.2 shows how to represent a relatively simple feature diagram. A more complex feature diagram is shown in Figure 6.4. For product lines whose feature diagrams contain many features, it would be impractical to implement applications using a single component composition. One reason may be that

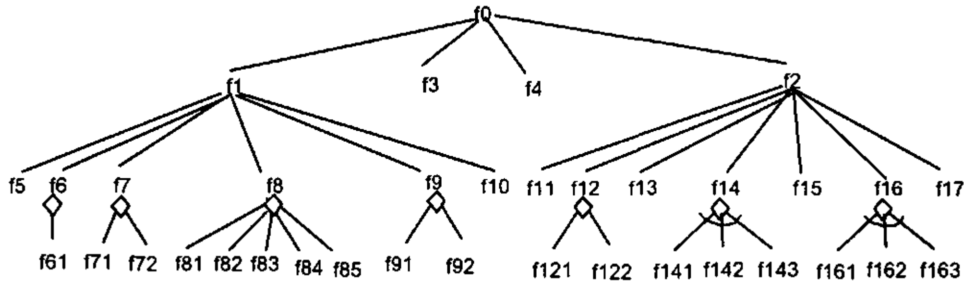

Figure 6.4 Complex feature diagram.

resulting hierarchical compositions would have many layers. Other reason may be that there is necessary to distribute part of the functionality in several sub-systems. For instance, the feature diagram in Figure 6.4 has 34 features and compositions constructed using the components implementing these features may have up to 30 components / layers.

Different reasons to divide complex compositions into several more simple composition equations (each one representing a subsystem) may exist (e.g., distributed subsystems should run on different threads or processors). The following specification describes how the feature diagram from Figure 6.4 can be used to construct two composition equations, called mg and mc:

```
f0 = (cnc, s0, {0})
f1 = (mg, s1, {f1})
f5 = (mg, s5, {f5})
f6 = (mg, s6, {f61})
f7 = (mg, s7, {f71,f72})
f8 = (mg, s8, {f81,f82,f83,f84,f85})
f9 = (mg, s9, {f91, f92})
f2  = (mc, s2, {f2})
f10 = (mc, s10, {f10})
f11 = (mc, s11, {f11})
```

f12 = (mc, s12, {f121, f122})

f13 = (mc, s13, {f13})

f14 = (mc, s14, {f141, f142, f143})

f15 = (mc, s15, f{15})

f16 = (mc, s16, {f161, f162, f163})

f17 = (mc, s17, {f17})

Note that the feature diagram in Figure 6.4 is not an arbitrary diagram, in fact, its structure corresponds to the feature diagram of the CNC systems domain described in Chapter 4. In the previous specification, mg and mc stand for Motion-Generator and MotionController, respectively, as was discussed for a CNC system in Chapter 4. The discussion of why several features (i.e., f0, f3, and f4) are not included in the specification was presented in Chapter 4.

Expressions used to specify feature diagrams make evident the difficulty of implementing variability among design elements when relationships change (e.g., evolution category 4). These variations would need to completely redefine the specifications expressions describing the feature diagram.

Note from the previous discussions that we still are not able to produce component compositions, for that we need the rules describing and limiting valid component stackings described by a hierarchical GenVoca component diagram. Later in this chapter we describe how the GenVoca model is used to help evolve product lines.

6.3 Evolving product-lines with meta-generators

It is more difficult to implement product line infrastructures than it is implementing single applications. Similarly, evolving product line infrastructures is more difficult than evolving single applications. However, the approach of implementing product lines by product line infrastructures (e.g., configuration wizards), can be

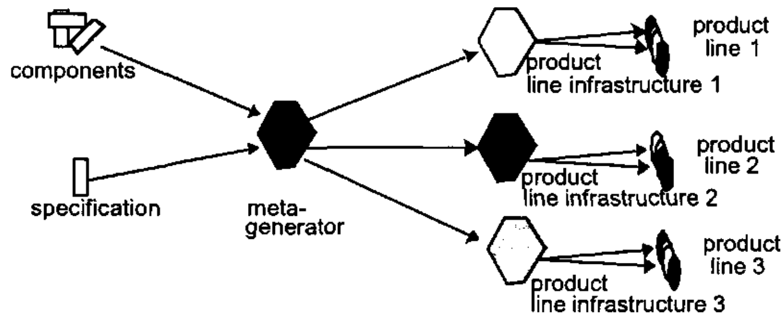

Figure 6.5 Meta-product-lines can produce application families

extended to implement meta-infrastructures to produce the product lines infrastructures. In our approach, this consists of implementing a generator of configuration wizards. Because product line infrastructures can be produced by software generators, a product line infrastructure generator can be itself a meta-generator. Such meta generator can obtain a specification of a product line infrastructure as input and produce the corresponding product line infrastructure.

Such meta-generator does not need to be completely produced by a generative approach, thus some software components may need to be provided besides the product line infrastructure specification. This approach is depicted in Figure 6.5; the similarities in symbols used to represent specific product line infrastructures and meta-generators is due to the fact that a meta-generator itself represents a product line infrastructure that produces a family of product line infrastructures.

To evolve a product line, the specification is rewritten accordingly and the meta-generator produces the particular product line infrastructure. In our notation to represent feature diagrams as sets of feature specification expressions, a specification defines a product line that the product line infrastructure should be able to produce. The complete specification of a feature diagram is processed by a meta-generator, from which a particular product line infrastructure is produced.

Discussion on how a product line infrastructure can be produced by meta-generator tools have been kept at an abstract level in order to describe the approach. Still something that has not been reviewed in detail is what exactly "components" (see Figure 6.5) represent. Information in a "component" could include descriptions or templates of how component compositions can be produced and verified, etc. The specific models could be specific to a meta-generation approach, the following section describes how such descriptive and partially implemented components are used to generate configuration wizards.

6.4 Evolving configuration wizards

Up to this point in this dissertation, product-line evolution has been dealt with in a general, abstract manner. Our approach of configuration wizards has not been related to our discussion. This section puts configuration wizards in context with our previous discussion of product line evolution, to show how configuration wizards can evolve to meet evolving domain requirements.

We defined configuration wizards as tools that present developers a visual specification interface for application specification. To limit or extend the possibilities of specifying different applications, the specification interface has to present application developers with the options available to specify applications. How underlying wizlets and configuration rules change for different sets of applications has to be un-noticed by developers. Application developers just have to be provided with the necessary facilities to specify applications, thus application developers are constrained to the specification facilities presented by the configuration wizard. Developers do not have to be concerned with how configuration rules and wizlets are added or removed from their development environment.

According to our proposed architecture for configuration wizards (see Figure 2.9 and Figure 2.8), the following events are necessary when a new feature is added to a configuration wizard:

- The corresponding wizlet is implemented and added to the wizlet repository.
- Domain knowledge (configuration rules) for the wizlet are implemented and added to the knowledge repository. Note that this may require changes to configuration rules in several wizlets.
- User interface is modified to display the specification for the new wizlet.
- Parser is modified to recognize the new wizlet.
- eqBuilder is modified to be able to add the new wizlet specification to a composition equation.

The first two events consist in implementing the new wizlet and its composition rules; these events have to be performed by hand. The remaining three events consist in modifying already existing code to handle specification and instantiation of the new wizlet. These three last events can be performed by a meta-generator as was described in Figure 6.5.

Before we discuss the architecture of a meta-generator, let us first analyze in an abstract manner the process involved in generating a specification interface

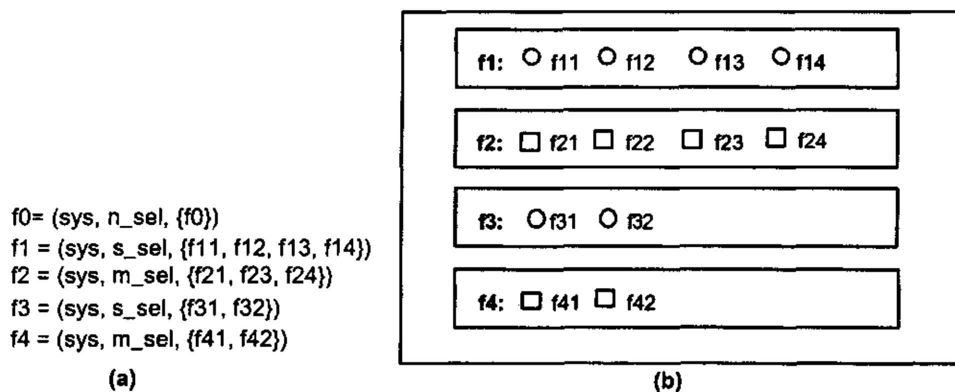

Figure 6.6 Specification and developer interface

(developer GUI) and the rules constraining component compositions, using a specification. Figure 6.6(a) shows the specification we previously developed for the feature diagram in Figure 6.3. Note that specific feature selectability properties are specified, thus the GUI can be generated. The selectability properties specified are interpreted as follows: s_sel specifies single selection from the sub-feature list, m_sel specifies that multiple sub-features from the list can be selected, n_sel specifies that features will not be presented in the GUI, but are still necessary for other activities. Figure 6.6(b) shows the GUI produced from the specification in Figure 6.6(a). In the GUI, we can observe how features are grouped in containers (rectangular areas) that allow single selection (circles called radio buttons), and multiple selection (small squares called choice buttons), according to constraints defined in the feature diagram (and in the specification).

Some properties from the feature diagram are not declared in the specification. These properties and constraints are implemented inside verification components, as was described in Chapter 2. Our approach to verification validation uses a Status class which contains attributes/variables whose values are used in the verification of a component composition. In the following paragraphs we show how the Status class is generated.

We start with the verification code for the component implementing feature f0 from the diagram in Figure 6.3.

```
class f0 {
/*<
   boolean f1_set;
   boolean f2_set;
   boolean f3_set;
   boolean f4_set;
>*/

void verify(Status status) {
  lower->verify(status);
  if (!status.f1_set)
     status.addMessage( "at least one f1 feature must be specified" );
```

```
   else if (!status.f2_set)
      status.addMessage( "at least one f2 feature must be specified" );
   else if (!status.f3_set)
      status.addMessage( "at least one f3 feature must be specified" );
   else if (!status.f4_set)
      status.addMessage( "at least one f4 feature must be specified" );
}
...
}
```

Note how attributes/variables to register the presence of each component (i.e., feature) in a composition are declared as comments (i.e., inside /*...*/ pairs). In the verification function, we assume that the Status component will have the necessary variables and that every verification component receives Status as a parameter. In the previous code, Status is used to check if at least one component implementing every mandatory feature was found. If not, an error message is reported.

The following code shows a partial implementation of the verification code for other components / features from Figure 6.3.

```
class f11 {
/*<
  boolean f11_set;
>*/
void verify(Status status) {
  if (status.f1_set)
     status.addMessage( "only one instance of f1 can be specified" );
  status.f11_set = true;
  status.f1_set = true; //assert an instance of generic feature f1
  lower->verify(status);
}
}
class f21 {
/*<
  boolean f21_set;
>*/
void verify(Status status) {
  status.f21_set = true;
  status.f2_set = true; //assert an instance of generic feature f2
  lower->verify( status );
}
}
```

Note the similarity in the implementation code segments for features f11 and f21. First every component declares its presence in the composition, and may be check that no other component implementing an alternative feature is present. Then the presence of a component implementing their generic feature is declared (i.e., sets the flag indicating this).

Other constraints not shown in a feature diagram may exist, and they should be implemented in the verification code. However, to keep discussion simple, we do not describe other constraints.

The previous code segments declared in a comment (/*...*/ declarations) the variable(s) they use internally to assert its presence to other components. To share the variable(s) with other components below or above it, a copy of the variable is put in a Status class, and one instance of the class is made available to every verification component.

Other code that can be derived from the product line specification is the code that generates a composition equation. Composition equations are constructed using the specification from the developer GUI, and for every selected feature its corresponding component is added to the composition. The following code sketches the code of a composition equation generator.

```
//Equation generator
equation.addComponent("f10("); //mandatory component
if (f11.selected() )
   equation.addComponent(" f11 ");
if (f12.selected() )
   equation.addComponent(" f12 ");
if (f13.selected() )
   equation.addComponent(" f13 ");
if (f21.selected() )
   equation.addComponent(" f21 ");
if (f22.selected() )
   equation.addComponent(" f22 ");
if (f23.selected() )
```

```
   equation.addComponent(" f23 ");
if (f31.selected() )
   equation.addComponent(" f31 ");
else If (f32.selected() )
   equation.addComponent(" f32 ");
else If (f33.selected() )
   equation.addComponent(" f33 ");
```

Other code that can be generated from the product line specification and the specification from a developer GUI, is the code to instantiate verification components. Assuming that the composition equation has been tokenized and a token representing a component's name is received as parameter, the following code shows how verification components are instantiated.

```
Component createInstance(String token) {
//Parser -instantiate verification components
if ( token.equals("f11") )
   return new f11();  //create instance
if ( token.equals("f12") )
   return new f12();  //create instance
if ( token.equals("f13") )
   return new f13();  //create instance
if ( token.equals("f21") )
   return new f11();  //create instance
if ( token.equals("f22") )
   return new f22();  //create instance
if ( token.equals("f23") )
   return new f23();  //create instance
if ( token.equals("f24") )
   return new f24();  //create instance
...
}
```

Finally, the code of the Status class will be something like the following code segment:

```
Class Status {
  String messages;
  boolean f1_set;
  boolean f2_set;
  boolean f3_set;
  boolean f4_set;
```

```
  boolean f11_set;
    ...
  boolean f21_set;
    ...

Status(){
   Messages=""; //clear messages
}

public void addMessage(String msg) {
   message= message + msg;
}
   ...
}
```

Our previous discussion described how different parts from a configuration wizard can be derived from a product line specification. Figure 6.7 shows an architecture describing how specific configuration wizards can be produced by a meta-generator, proceeding in a similar way to that described above. Figure 6.7 summarizes our previous discussion on how feature diagrams can be represented by specification expressions and the use of a product line meta-generator, combined with our notion of configuration wizards. The central role in Figure 6.7 is played by the meta-generator (MGenerator)[4]. A product line specification (PLSpec) is provided

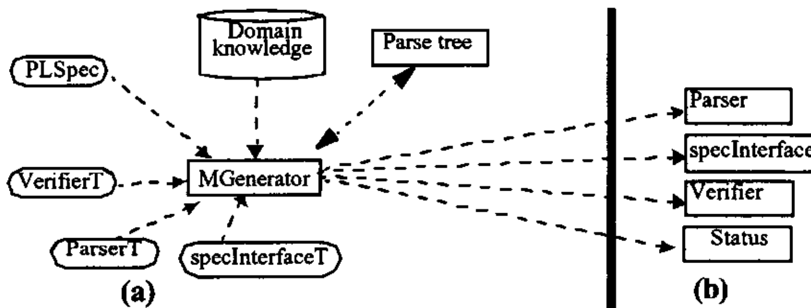

Figure 6.7 Configuration wizards derivation from product line specifications

4. Note in the figure that rounded rectangles represent source files (text files) while squared rectangles represent compilable code.

to the meta-generator from which it produces an internal representation, called a parse tree (data structure) in Figure 6.7, which will be used in most of the tasks performed by the meta-generator. Other input provided to the meta-generator consists in several template files (i.e., files containing markups specifying places where code should be inserted), that will be used to generate some of the components used by the configuration wizard. In Figure 6.7 all template source files are at the left, and the generated files are at the right. One template file is ParserT (parser template), which is used to generate the Parser component, that will parse composition equations and instantiate validation components in the configuration wizard (remember that we decided to split wizlets in two parts: validation code and application code). Another template file is specInterfaceT, a template used to generate the specification interface (specInterface component) for the configuration wizard. A third template file is VerifierT, which is used to generate the Verifier component that will be responsible for conducting the validation process in a composition equation. Finally, another component generated by the meta-generator, and which does not have a template counterpart is Status; Status is directly generated from the provided product line specification (PLSpec).

To understand how Status is generated, remember that the validation process in a composition equation is performed in two directions (top to bottom and bottom to top in the hierarchy), thus we need to keep information of the presence of each component participating in an application composition. For that, a boolean value is enough (true if the component is in the composition, false if it is not). Other validations require more complex variable types: we may need to check the number of axes (an integer value), a motor type (a string value), etc. Because these values need to be propagated and made available to all components, our solution was to put all validation values associated to every component in the Status component. However, when new components are added or deleted, their corresponding

validation variables need to be added or deleted correspondingly in the Status component, and this should be performed by hand (not a recommended task). One way in which we can put all necessary validation variables inside Status, is to copy them from each validating component to Status, using the product line specification as a guideline. At validation time, all components update and use information directly from Status. Because at application generation time the application developer will be limited to specify only those components available in the GUI, we can never face the case of an unknown verification variable.

To describe how Status is generated, here we use the feature diagram from Figure 6.3 and its specification presented in (6.2). To simplify the description, lets assume the only constraints are the presence or not of components in a composition equation. According to Figure 6.3, Status has the following implementation, written in Java:

```
class Status {
    String message;
  public:
    boolean f1_set; // flag to mark component's presence
    boolean f2_set; // flag to mark component's presence
    boolean f3_set; // flag to mark component's presence
    boolean f4_set; // flag to mark component's presence
    boolean f11_set; // flag to mark component's presence
    boolean f12_set; // flag to mark component's presence
    boolean f13_set; // flag to mark component's presence
    boolean f21_set; // flag to mark component's presence
        ........
  Status( ) { //constructor
      message = ""; //string initialization
  }
  void addMessage( String msg) {
        message = message + msg;
   }
  ........
};
```

Note that verification variables for features in the diagram are included inside Status, these variables were copied from the verification components. A

partial implementation of the verification equation for some of the components in Figure 6.3, is as follows:

```
class f0 extends Verifier{
  /*< //start markup
   boolean f1_set; // flag to mark component's presence
   boolean f2_set; // flag to mark component's presence
   boolean f3_set; // flag to mark component's presence
   boolean f4_set; // flag to mark component's presence
  >*/

  void verify(Status status) {
    lower.verify(status);//verify lower components first
    if(! status.f1_set)
      status.addMessage("At last one f1 feature must be specified");
    if(! status.f2_set)
      status.addMessage("At last one f2 feature must be specified");
    if(! status.f3_set)
      status.addMessage("At last one f3 feature must be specified");
    if(! status.f4_set)
      status.addMessage("At last one f4 feature must be specified");
   }
};
```

Note in this code segment that verification variables are inside a pair of tags /*< ... >*/ (by using Java standard comments, we can leave the code untouched inside the component, so it still is helpful for latter reference and does not interfere with compilation). However, in the verification equation, variables are assumed to be declared inside Status. Note also that verification variables are verified after the verification process has been performed. The following code segment describes a partial implementation of the verification component for a specialization of feature f2, namely f21:

```
class f21 extends Verifier{

   /*< //start markup
    boolean f21_set; // flag to mark component's presence
   >*/

   void verify(Status status) {
    state.f2_set=true; // a specialiaztion of f2 has been found
```

```
    state.f21_set=true;
    lower.verify(status);
    if(status.f3_set)
      status.addMessage("f3 features can't be used with f2 features:
f21");
   }
   .......
};
```

Note in the code the assumption that a sub-feature of f3 is assumed to be in a "lower layer".

The following code shows the implementation of a sub-feature for feature f3, namely f31:

```
class f31 extends Verifier{

   /*< //start markup
    boolean f31_set; // flag to mark component's presence
   >*/

   void verify(Status status) {
    state.f3_set=true; // a specialiaztion of f3 has been found
    state.f31_set=true;
    if(status.f2_set)
      status.addMessage("f2 features can't be used with f3 features:
f31");
   }
   .......
};
```

The meta-generator also produces a Parser component, Parser is produced from a template file. The partial final appearance of the code for the generated Parser component is:

```
class Parser {
     ......// code common to all parsers
     Component newComponent(String token) {
        if( ( token.equals("f21"))
           return new f21();
        if( token.equals("f31"))
           return new f31();
    .....// code common to all parsers
};
```

The code shows that it is necessary to call a constructor for every component we found when a composition equation is being parsed by the configuration wizard. The code that needs to be inserted inside every new component specified in the product line specification is similar, and can be generated using the specified components. To define where the code needs to be inserted, a tag is used. For instance, the template for ParserT is defined as follows.

```
class Parser {
     ......// code common to all parsers
/*< specific code for parser goes here >*/
    .....// code common to all parsers
};
```

Note again that the pair /*< ...>*/ is used as markup of the place where the code to call appropriate constructors should be inserted. That markup is used by the meta-generator to insert the code for instantiating verification components.

The other components (specInterface and Verifier) are generated in a similar way to how Parser is produced, thus we do not describe them here. The only thing that can be worth mentioning is that implementing specInterface would be simple or complex, depending on the programming language constructs available to dynamically produce graphical user interfaces. Programming languages and their graphical environments provide different facilities to modify the graphical specification of compositions. The following sections show how two of the product lines that were described in previous chapters of this dissertation can be generated using the approach described in this chapter.

6.5 Evolving a product-line of vehicle simulators

The previous section described how a meta-generator uses a product line specification to determine which features are to be presented to the application developer in the GUI specification interface by the generated configuration wiz-

ard. That information is used also to specify which components can be included in the equation builder module of a configuration wizard. Our proposed approach of generating configuration wizards from product line specifications and meta-generators is applied here to show how a specific product line can be evolved.

In Chapter 3 we described a configuration wizard for vehicle simulators. The configuration wizard for vehicle simulators can be described using the product line specification notation we introduced in this chapter, as follows:

```
vehicle = (vehicle,n_selectable,{vehicle})
Options = (vehicle, m_selection,{ Trailer})
Type = (vehicle, s_selection, {Car, TwoWheelMotorcycle,
                              ThreeWheelMotorcycle, Tank})
Controller = ( vehicle, s_selection, {Intuitive, Fuzzy})
Movement = ( vehicle, s_selection, {Normal, Differential })
Path = ( vehicle, n_selectable, {Path})
Parameters = (vehicle, n_selectable, {Parameters})
```

The specification GUI interface generated from this product line specification is shown in Figure 6.8. There are several things worth noting in the specification. First is that a single composition equation will be produced by the configuration wizard (vehicle)[5]. The selectability properties specify the functionality necessary in the user interface without committing to a particular programming language: s_selection stands for a single selection component, m_selection specifies a multiple selection component, n_selectable specifies features that will not be presented in the GUI interface, but necessary to produce the composition equa-

5. The necessity of specifying more than a single composition for one application was explained before in this chapter. Complex applications would consist of several modules that should be executed as separate intercommunicating processes. Each module is a subsystem that needs to be compiled and executed separately from the others, and thus needs to be generated as an independent composition.

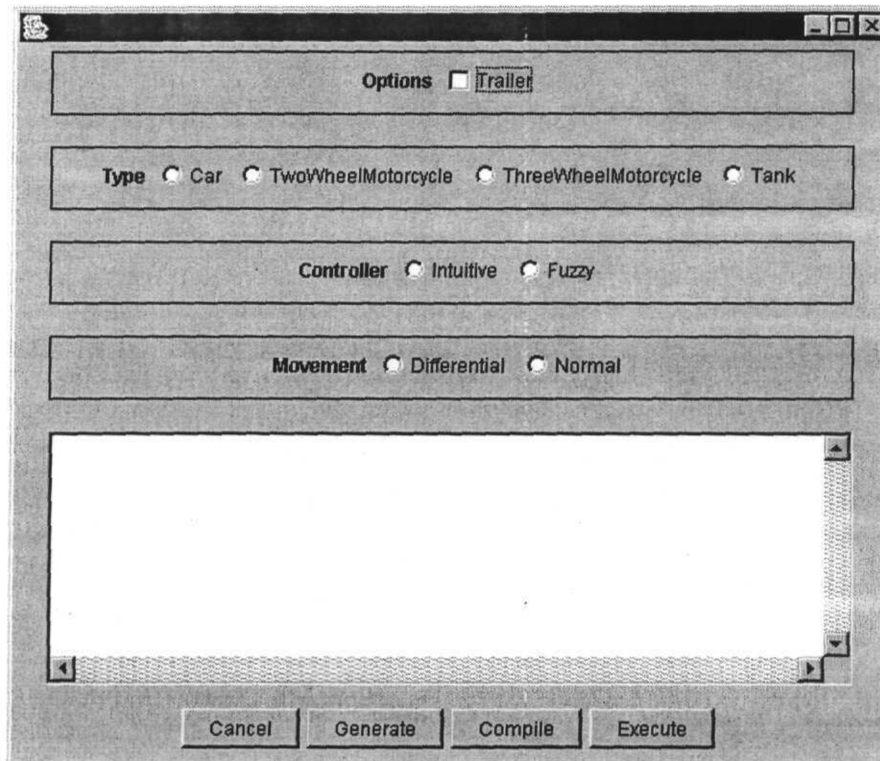

Figure 6.8 Original interface of configuration wizard for vehicle simulators

tion. Adding or removing groups of elements from the GUI interface is simplified by the selectability properties. For instance, s_selection property can be implemented using radio buttons or drop-down single-selection lists; m_selection properties can be implemented by multiple selection combo boxes.

To describe how this particular configuration wizard evolves, we can try several changes. The first change consists in adding a new controller type; this change requires editing the product line specification thus the corresponding feature specification description for controller is

Controller = (vehicle, s_selection, {Intuitive, Fuzzy, Lazy})

showing a Lazy sub-feature added to the Controller feature. Other change that we can make is to remove a feature. For instance, we can remove the Options feature from the product line specification[6]. Figure6.9 shows the specification GUI interface produced after these two changes have been done to the product line specification (i.e., Lazy instance added to Controller feature, and removing Options feature).

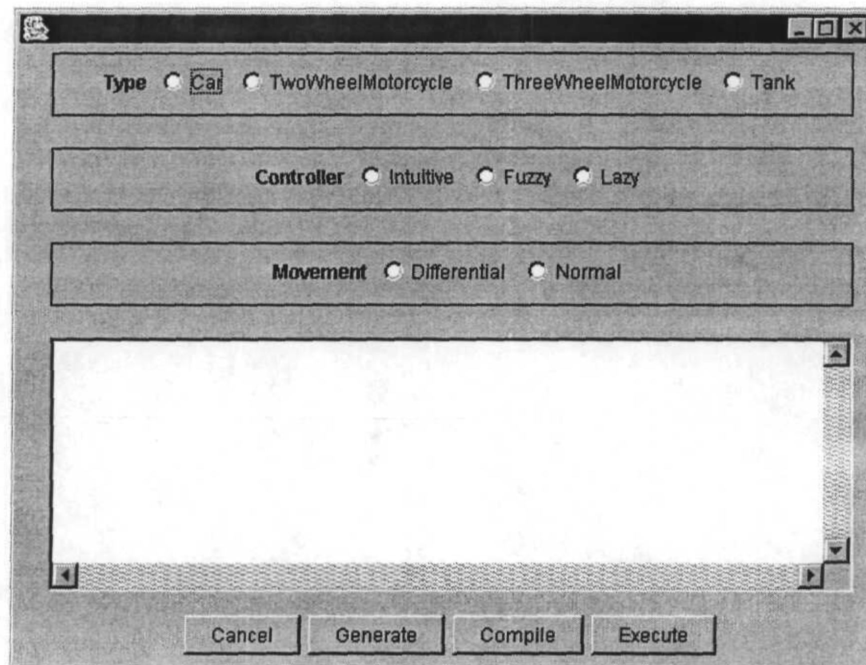

Figure 6.9 Evolved interface of configuration wizard for vehicle simulators

6. Note that completely removing a feature has to be consistent with the feature diagram. Only optional features can be removed from a configuration wizard's specification user interface.

6.6 Evolving a CNC product-line

In Chapter 4 we described the analysis and design domain models of a configuration wizard for CNC applications. As was described in Chapter 4, a CNC system has two subsystems: MotionGenerator and MotionCoordinator. MotionGenerator translates machine instructions to an internal, more detailed representation; MotionCoordinator synchronizes the feeding of these instructions to machine tool's mechanical parts (or more appropriately, to servos attached to mechanical parts).

Following is the product line specification for a CNC configuration wizard:

```
Expander =       (mg, n_selectable, { Expander })
Interpreter =    (mg, n_selectable, {InterpreterEIA274 })
CoordAdapter = (mg, n_selectable, {CoordAdapter })
Canned cycles= (mg, m_selection, {Finishing, RoughCut,
                PeckDrilling, Facing})
Interpolators = (mg, m_selection, {Linear, Circular})
Dispatcher =   (mc, n_selectable,{Dispatcher})
AxesControl = (mc, n_selectable,{AxesController})
Axes =          (mc, s_selection, {2, 3})
Motor =         (mc, s_selection, {StepMotor, DCMotor})
Driver =        (mc, n_selectable,{Driver})
Inverter =      (mc, n_selectable,{Inverter})
Card =          (mc, s_selection, {PC100Card, SimulationCard})
```

In the product line specification, mg and mc stand for MotionGenerator and MotionControler, respectively. As the product line specification shows, there are many non-selectable (and thus mandatory) features. Non-selectable features represent components that are mandatory for different compositions. Figure6.10 shows the specification GUI interface of the configuration wizard for CNC systems, produced by a generator from the previous product line specification. Elements avail-

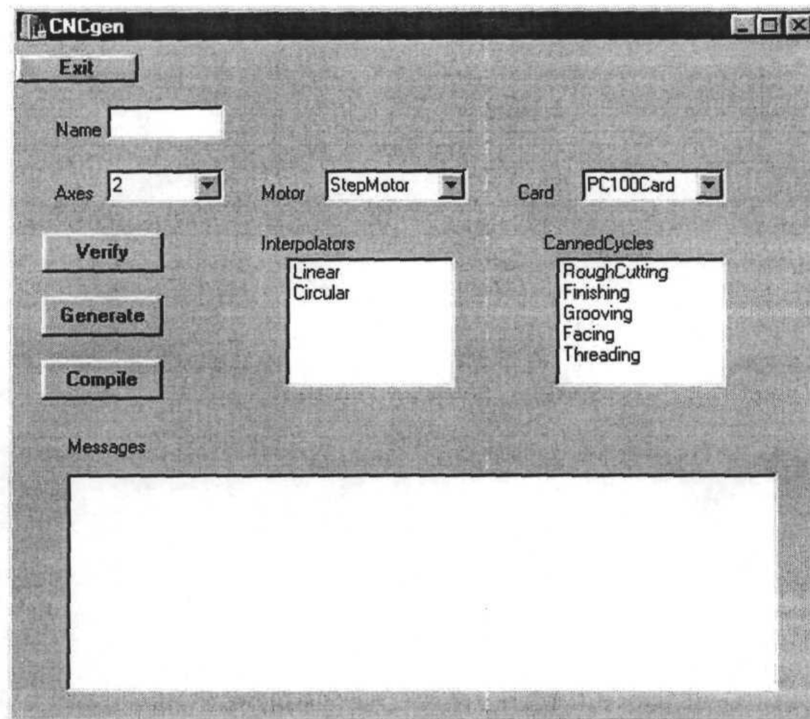

Figure 6.10 Original interface of CNC configuration wizard

able in the programming language, and constraints in the CNC evolution possibilities, allow us to define a constant specification GUI interface. As described below, the evolution possibilities for this domain consists in adding or removing features from already existing generic features, but not adding or removing generic features.

In Chapter 4 we discussed possible variations in a CNC product line. These variations can be made to applications if the configuration wizard is evolved to include/exclude corresponding features. Following are the identified variations and a brief discussion on how each variation can be approached:

- Card type: a new type of card, or a different configuration of a card will be supported. The required component needs to be implemented and its specification (name) added to the Card feature specification expression.
- Interpolators: a new type of interpolator needs to be incorporated. The component implementing the interpolator is added and its name included in the Interpolators feature specification expression.
- Canned cycles: a new type of canned cycle will be used. The new canned cycle is implemented in a component and its name is added to the Canned cycles feature specification expression.
- Motor type: use a new motor type. The corresponding component is implemented and its name added to Motor feature specification expression.

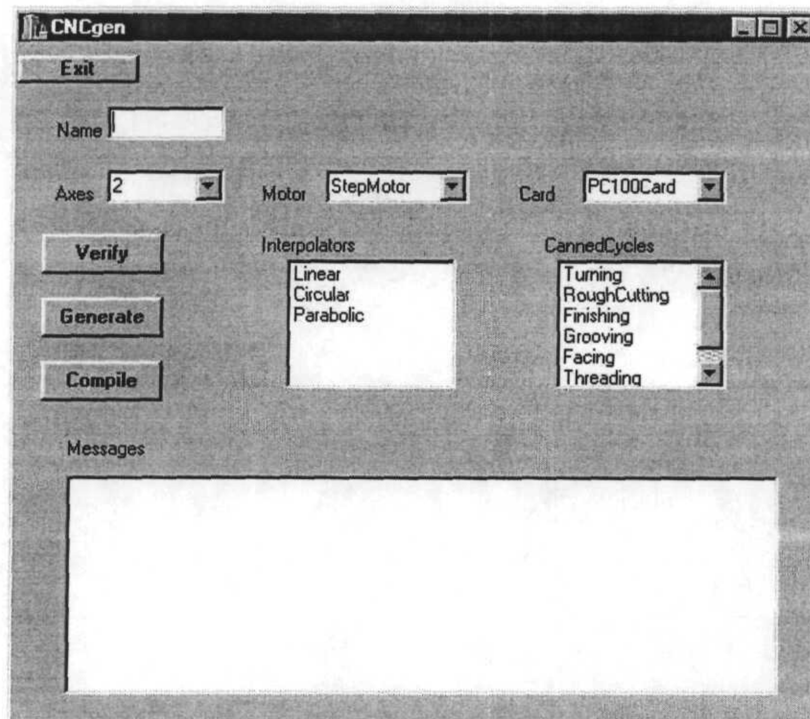

Figure 6.11 Evolved interface of CNC configuration wizard

- Machine type: this is a complex requirement which can influence several features. The number of axis can be different, the machine may require a new type of card (or a new configuration of an existing card), etc. In every case, the appropriate components are added/removed and the corresponding meta-specifications are written.

An example of the user interface for a CNC configuration wizard including a new interpolation algorithm (Parabolic interpolation) and a new canned cycle (Turning), is shown in Figure 6.11.

As was emphasized in our first example, how changes are introduced in a product line specification has to be consistent with the feature diagram.

6.7 Limitations and advantages

In this chapter we discussed why product line evolution is necessary and described how meta-generators can automatically perform required changes. We explained how product line evolution can be implemented by a meta-generator whose products are configuration wizards. In the approach described, configuration wizards can be adapted to support new requirements, represented as features, by modifying their product line specifications and generating configuration wizards.

In the simplicity of the idea is its contribution. Evolving single applications is a paramount problem; evolving product line infrastructures capable of generating sets of similar applications is indeed a more complex task. We demonstrated in this chapter how the use of features along the development cycle of configuration wizard implementation simplifies the addition and removal of features. The combination of feature diagrams with GenVoca hierarchical diagrams helps to simplify the task. The addition of graphical specification interfaces for application configu-

ration simplifies the incorporation of new features to applications (i.e., if a feature is available for its use in applications, it will be part of the graphical interface, if it is not, then it will be absent from the interface). Implementing components as wizlets that are adapted by configuration wizards using application developer specifications simplifies the task of evolving product lines.

The approach used has its limitations and advantages. The following two sections discuss them.

6.7.1 Limitations

Our approach for product line evolution is strictly limited to our implementation of product line infrastructures as configuration wizards. Our interpretation of components as implementing features simplifies both the initial implementation of configuration wizards and their evolution by meta-generators. One limitation is that we did not explore how different ways of modularization and parameterization would give different results.

The implementation of the meta-generator is strictly guided by the architecture of a configuration wizard. Evolution consisting in feature addition or removal to the product line is easy, as shown in the examples we implemented. However, the implementation of meta-generators is highly dependent in the feature diagram and the GenVoca model(s). A big limitation is that in general, it is difficult to implement structural (i.e., architectural) changes after the meta-generator has been implemented. Given the tight dependency of the meta-generator to the models, changing the models can require a lot of hacking in the meta-generator.

Each feature is implemented by a component. It is difficult to construct configuration wizards if features cross-cut components. Such features as transactions and distribution, which cross-cut to several components cannot be dealt with in a simple manner, if possible, using our approach.

Domains different to those we implemented may impose performance requirements. Although performance estimation can be calculated, resulting compositions may not be enhanced to improve performance; that is, components are composed as is.

Compositions are limited to their specification in the GenVoca model, thus even if their semantic properties allow different component orderings in compositions, our approach limits compositions to a single option. More flexibility can be achieved if components can be used in different positions in a composition. Such flexibility would result in applications with a better performance. Configuration wizards and their meta-generators would need a certain level of intelligence to allow such possibilities. We did not explore such possibilities.

We acknowledge that may be the simplicity to evolve our configuration wizards can be attributed to the care taken in their initial development. An original requirement on the examples was that they should make simple feature addition and removal. If configuration wizards and their wizlets are not designed for evolution from the beginning, surely they will be more complex to evolve using a meta-generator (or even by hand).

It may be impractical to use an incremental approach to implement configuration wizards and their meta-generators. This is a restriction for domains in which there are not available applications to analyze, previous to the implementation of a configuration wizard. Incremental development is one of the current approaches of software engineering, but we can not take advantage of it in our approach.

A compositional implementation does not allow the use of more than a single instance of the same component in a composition. This makes difficult to apply the approach in domains such as data structures that would require such possibility. However, such impossibility can be attributed in part to programming lan-

guage mechanisms, not completely to our approach. For instance, programming languages such as Eiffel support a mechanism for property (e.g., method and attribute) cancellation from super classes to subclasses. Such mechanisms can make possible to use several instances of the same component in a composition. We did not explore such possibilities.

It was not our intent to propose a generic approach to parameterization. We did not conduct any analysis of the parameterization extensions, thus other possibilities that are enabled by the implemented extensions were not explored.

6.7.2 Advantages

As was described in the previous section, a number of limitations exist in our approach of configuration wizards and their meta-generators. However, as examples shown demonstrate, their possibilities are still important. The product line specification developed to specify configuration wizards is simple enough to facilitate the evolution of different configuration wizards.

We did not consider ourselves expert programmers in the programming languages used to implement configuration wizards and their meta-generators. However, carefully applying our proposed approach made simple such implementation in different programming languages. We consider the approach is strong enough to be applied in other domains using other programming languages with similar characteristics to the languages we used in our examples.

The configuration wizards that are meta-generated are similar to those we initially implemented by hand. Once constructed, meta-generators highly simplify the evolution of configuration wizards.

We consider a main advantage of our approach that it can be consistently applied across programming languages and domains, as demonstrated by the

examples we described. The parameterization limitations from different programming languages were consistently solved by using a similar extension approach.

6.8 Discussion

The approach to product line implementation based on configuration wizards was extended in this chapter to implement product line evolution by meta-generators. A notation to specify configuration wizards is introduced. The notation is used to specify two of the example configuration wizards described in previous chapters, and describe how these configuration wizards can be evolved by a meta-generator specific to a product line.

Advantages and limitations of the approach to evolve configuration wizards were discussed. These limitations and advantages are in part a consequence of the domain analysis and design representations used. Several of the limitations were attributed to the weak support from programming language mechanisms, mainly related to their limitations in module or class parameterization.

Chapter 7

Related work

Previous chapters sparingly described the work of others related to our work. In this chapter we discuss research being conducted in the area of software product lines, and how we consider our work on configuration wizards extends or concretizes the work of others.

7.1 Product-Line Engineering Methods

Numerous approaches to product-line engineering exist as can be observed from the success of product-line workshops and conferences [Don00, Cha02], and product-line engineering books [Bos00, JRV00, WL99]. In this section we present descriptions of related approaches and what we consider are their most relevant contributions to product-line engineering. For every approach, we discuss why we consider it is limited in scope to be considered a complete and detailed product-line engineering method.

7.1.1 FODA

The Feature-Oriented Domain Analysis Method (FODA) [KCH+90], has been successfully used in different domains. Its distinctive characteristic is that domain concepts, called features, are represented as tree-like relationships describing com-

positions, and-or associations, and generalization/specialization relationships. A FODA model is used as a communication tool between users and developers because concepts are derived from the terminology ordinarily used by domain experts to describe the domain. FODA is limited to domain analysis, no approaches are suggested to how features can be implemented so they can be reused to construct the product line described by a feature model.

Our method considers FODA the most relevant method for domain analysis, and extends its usefulness to design product-line architectures based on parameterized components which implement features, that can be reused to construct application families.

7.1.2 FeatuRSEB

The Reuse-Driven Software Engineering Business method (RSEB) [JGJ97, GFD98] is a domain modeling method aimed at software development enterprises, and is distinctive it that it emphasizes reusability. The unifying concept along the RSEB method is use cases. In FeatuRSEB, FODA's features are integrated with use cases by deriving feature models from use case diagrams.

In the description of FeatuRSEB and its models, no suggestion is presented on how features are individually implemented and later composed into applications. In contrast, our method considers configuration wizards an their wizlets as inseparable elements to implement and produce application families.

7.1.3 Feature Abstract Specification and Translation

The Feature Abstract Specification and Translation (FAST) approach [WL99] proposes general recommendations of a process for product-line engineering. FAST suggests a generator-based approach for product-lines, but does not propose spe-

cific ways that can be used for domain analysis, component implementation, and generator development.

7.1.4 Organization Domain Modeling

The Organization Domain Modeling (ODM) [Sim95] is a systematic domain analysis and design approach. Similarly to FAST, ODM is a general approach thus there is no commitment to any particular methods and technologies. An important aspect in ODM is its emphasis in recommending that features should be used along the process (i.e., from analysis to design to implementation).

The recommendations of ODM are extensively applied in our work to propose a method whose elements are features. Our method show how features can be implemented as components that are concretized into applications by a configuration wizard.

7.1.5 FODAcom

FODAcom [VAM+98] is an extension to FODA for the telecommunications domain. Several FeatuRSEB's elements are also present in FODAcom. For instance, both make use of use-cases at their initial phase, thus feature identification is performed at the use-case level. However, FODAcom employs a UML-like [BRC98] notation for representing feature models. Functional and behavioral FODA models are used to derive domain architectures. No clues are provided over how to implement components and tools necessary for product-lines based on FODAcom.

7.1.6 PuLSE

The Product Line Software Engineering (PuLSE) [DFK98] process is a collection of methods covering the development life cycle, consisting of PuLSE-Eco (eco-

nomic feasibility study), PuLSE-CDA (concept identification, structuring, and documentation), and PuLSE-DSSA (definition of a domain specific software architecture). The PuLSE process has been used in a number of small and medium enterprises [KMS+00]. PuLSE does not define any particular implementation approach, nor specific engineering methods. In contrast, our method consistently uses parameterized components (features) for feature implementation.

7.1.7 FORM

The Feature-Oriented Reuse Method (FORM) [KKL+98] extends FODA to cover the whole engineering process. FORM has been implemented in functional and object-oriented environments. Reported benefits are attributed to the easy translation of feature models to object models because of similarities in concepts used in both modeling approaches (i.e., objects can represent real world entities). Variability is implemented using inheritance and templates, macros are used for selecting among alternatives [LKCC00].

Our work extends FORM by adding configuration predicates to components prescribing constraints that should be met for them to be reused in compositions describing applications.

7.1.8 GenVoca

GenVoca [BST+94] is a design approach that represents an application as a component stacking (hierarchy). Every component modify in some way its immediate top level component. Such modifications, called *refinements*, can consist in adding, removing or changing in any other way the output of the previous component. GenVoca has been implemented in a number of generators for different domains. GenVoca is distinctive in that components are parameterized with respect to their low level component, thus a GenVoca model is a reference architecture which can

be instantiated differently. To guarantee that a component stacking defines a valid application, GenVoca proposes describing component inter-dependencies as predicates, which are to be true for valid compositions and false for invalid compositions.

Our contribution to GenVoca is the addition of feature models (diagrams) to identify component types and represent constraints as relationships in feature diagrams, that will be translated to constraints in GenVoca. By doing so, features are a unifying concept along a product-line method (components implement features that are parameterized by other features that can be instantiated in compositions implementing applications.

In recent implementations of GenVoca models, components represent collaborations, thus responsibility-driven or use-case driven approaches could be used for analysis [Sma99].

7.1.9 Discussion

The methods described in previous section can be categorized in two groups: methods emphasizing a single aspect or part of a product-line method, and generic methods presented at a very general level thus any particular approach fits them.

In previous chapters we presented a method for product-line engineering applicable across multiple domains. Such method does not proposes completely new activities and work products for each life-cycle development phase. Instead, the method uses the most successful approaches currently in use. By combining successful approaches to define the activities and their products, the experience in their use is retained.

7.2 Implementing Product-Lines

Infrastructures being used for implementing product-lines are diverse. Several of the implementation technologies are presented in this section.

7.2.1 Aspects

For some problems, design decisions cannot be captured in a single unit. For instance, in an object-oriented approach some design decision cross-cut several classes. *Aspects* are programming constructs that work by cross-cutting the modularity of classes [KLM+97]. So, for example, a single aspect can affect the implementation of a number of methods in a number of classes. Certain domain independent issues in applications typically cross-cut several classes (e.g., persistence, transactions, distribution).

Aspects can consist in attributes or methods inserted to selected classes in an application. An aspect *weaver* is a tool that processes aspect declarations and performs insertions declared by the aspect declarations. Programming languages can be extended to support aspects, thus no completely new languages need to be designed for aspect-oriented programming [Kic00]. An example of an aspect weaver is AspectJ, which extends Java syntax to declare aspects, as extensions to attributes and methods of classes.

7.2.2 Frameworks

A framework implements the common characteristics of a product-line, and defines a mechanism to add the variable characteristics thus applications can be constructed [Bas97]. Object-oriented programming-language mechanisms, mainly inheritance and polymorphism, are used to construct framework infrastructures [FJ99, DW98, Lew95]. Frameworks have been broadly used to implement soft-

ware product-lines [FJ99]. An interesting approach for implementing frameworks using parameterized components is presented by [BCS00].

7.2.3 Mixins

A mixin is an abstract sub-class whose super-class is unknown at implementation time [Bra92, VN96]. At application construction time, mixins are instantiated by specifying their super classes in a composition expression. In programming language implementations that support the definition of classes inside other classes, a mixin can consist of several inner classes inside a container (component) class [FF98, Sma99].

Bracha [Bra92] discusses how a modular programming language can be extended to support mixin implementation and how parameterized modules are used to build applications. An approach for implementing mixins describing roles in a collaboration is proposed by VanHilst [Bra96a, BR96b]. Smaragdakis [Sma99] shows how such approach requires complex compositions even for simple application systems, and proposes that a single mixin can implement a complete collaboration, these are called mixin layers. Mixin layers require simpler composition expressions to specify whole applications.

7.2.4 Components

What is considered a software component varies considerably according to the implementation approach. A component is a unit of encapsulation and can be implemented as functions, classes, modules, units, packages, and the like [FF98, BBC+00]. The important characteristic is that a component can be handled as a unit at application construction time and tools can help at component integration time [KC99, RFS+00, SMB00].

7.2.5 Generators

A software generator receives as input an application specification expressed in a domain-specific language and produces the corresponding application in a programming language such as C, Pascal, Java, etc.[BO92, Haz96]. There are two types of generators, compositional and transformational [Tho98]. Compositional generators map specifications to parameterized components that are adapted to fit the specification. Transformational generators refine the specification in several steps, applying one algorithm at every step until a fully refined application is produced. Transformational generators can proceed independently at each step or be guided by the developer thus correct transformations (e.g., reduction, optimization, etc.) are applied [CE99, CE00].

7.2.6 Software Kits

A software kit consists in a set of components and tools for application construction as component compositions [GW94]. Tools for component integration can perform verification of the compositions [Sta00, RFS+00]. There is no restriction to what a component can be and to how better represent and implement components and their kits.

7.2.7 Design wizards

Wizards (also known as "experts" and "advisors") are visual tools to walk the user along the steps of a complex process for application specification. An example of a design wizard for construction and critique of data structures was presented by Batory et al. [BCR+00]. Wizards extend software kits by providing expert assistance, such as helping to improve application performance and textually describing component compositions. Design wizards have been suggested as front-ends for

many technologies [BCR+00], such as software generators [BST+94, Kie96, Tho98], frameworks [Joh88], and libraries [Big94].

7.2.8 Configuration environments

Configuration environments [BABR97, KC99, Kot99, RFS+00, SMB00, Sta00] assist users in specifying applications. The developer is provided with specification facilities (e.g., languages, tools, etc.) for application description. The environment verifies consistency and reports any errors, thus development proceeds in a conversational manner. Composition can be performed statically (before the application is compiled) or dynamically (as the application is running). Consistency verification can employ reflection capabilities, be based on interfaces (typing), or use more detailed (semantic) component descriptions.

7.2.9 Discussion

Technologies presented in this section have been used to implement software product-lines. However, significant cost is associated to most of them. For instance, aspect-oriented programming requires extensions to programming languages and sophisticated weavers to be developed; generators are complex tools which are expensive to implement. A low cost alternative is the mixin-layers approach, which requires that class parameterization be supported by the programming language. Class parameterization is supported by a limited number of programming languages (e.g., Ada, C++, Modula-3); other broadly used programming languages do not support class parameterization (e.g., Pascal, Visual Basic, Java). Besides being limited to programming languages supporting class parameterization, mixins lack support for complex composition verification.

In previous chapters dissertation we showed how application engineering in a product-line environment can be simplified by the use of configuration wiz-

ards. A configuration wizard is a software tool containing wizlets as components, predicates expressing knowledge of valid compositions, a specification interface, and a generator to produce software product-lines.

7.3 Product-line evolution

In Chapter 6 we presented and discussed an approach to evolving product-lines implemented by configuration wizards. In that chapter we described how two of our example product-lines are evolved by meta-generator tools, which take product-line meta-specifications expressed as feature sets and produce the configuration wizard implementing a product-line.

Evolution of individual applications has been broadly researched. However, an issue that has not been extensively discussed is how product-lines can evolve to support new requirements, or eliminate requirements that, for any reason, will not be available in a product-line infrastructure. Most of the literature in product-lines refer to methods and tools to construct product lines. How these methods and tools address product-line evolution is not clear.

Riebisch [RP01] describes a process for assisting in product-line evolution by maintaining links among life cycle work products. The links relating requirements to design decisions and implementation is suggested as a means to help evolve product lines. A tool to record and use documented decisions is proposed for changing, refactoring, and reconfiguring product-line architectures.

Katayama [KT01] proposes a formalization of collaboration-based product-lines. Operators for representing specifications, evolution, and differences help trace the validity of changes made to a product line. No tools implementing proposed operators are suggested for evolving a product-line.

Although not presented as means to help evolve product-lines, compositional frameworks [BCS00] can help to implement evolution in a product-line. Adding parameterized components to a framework exponentially increases their ability to construct different members in an application family.

Metaprogramming mechanisms can help to evolve product-lines [CE00]. Implementing features as parameterized components simplify the addition of new requirements to meta-generators based on metaprogramming facilities.

A broad analysis on product-line evolution is presented in [Bos00] and [SGB01]. Similar to our approach, features are identified as a unifying concept to model, implement, and evolve product-lines. Variability is identified as present at different levels of abstraction in product-line work products, from architecture to code. Categories of evolution in a product-line are described in [SGB01]. How the mapping of category changes to implementation can be performed is not discussed.

7.3.1 Discussion

We do not present a formalization of product-line evolution implemented as configuration wizards. However, we consider our approach partially includes concerns and results from work related to product-line evolution. Contrary to others, our work shows how evolution can be performed by adapting the product line infrastructures from meta-specifications describing them.

7.4 Recap

Our approach for software product lines is based on several product-line engineering methods. The implementation we use is similar to mixin layers with the addition that programming languages with encapsulation facilities can be

extended to support component parameterization. As described, configuration wizards are derived from a combination of design wizards, software kits, and configuration wizards. Results in product line evolution are considered to propose an approach that simplifies evolution of software product lines by meta-generators.

Chapter 8

Conclusions

The work described in this dissertation was guided by two goals. The first was to propose a methodology for software product line engineering and demonstrate its general applicability across domains and programming languages. The second was to find out if a methodology for product lines can be extended to include the complexity of automatically evolving the product line infrastructure. The previous chapters show that, to a certain extent, both goals were achieved. In this chapter we recap the results and contributions, discuss lessons learned both in developing and evolving software product lines, and propose research that remains to be conducted in the field of configuration wizards

8.1 Proposed Methodology

The methodology we proposed was based in the abstract reference process for product line engineering in existance since 1996, as reported by F. van der Linden [Van02]. In this dissertation, we propose the use of specific methods to perform the activities from that reference process, as follows:

- *Domain analysis.* We proposed the use of feature diagrams with extensions that we introduced to represent feature variations and constraints. We found feature diagrams to be of great use to communicate with the clients, so they can validate if all important domain aspects have been registered and are related in a correct way.
- *Domain design.* Hierarchical component diagrams are used to represent the design of software product lines. We added adornments to components so that their characteristics could be emphasized --important when modules are to be implemented and composed with others using a visual specification.
- *Domain implementation.* We use modular implementations similar to mixin-layers (i.e., parameterized classes) to implement components. However, to be able to provide more complex verification capabilities than mixin layers do, the composition constraints are coded separated from the application code. Our approach also assumes that components will be composed by a composition tool, according to an application specification. This last assumption provides enough flexibility to include mandatory components in an application without the need to be specified by an application developer, and if necessary, to perform complex adaptations to components in a composition.
- *Application engineering.* Once the infrastructure for a software product line has been created, particular applications can be generated. The specification and generator front-end provides the application developer with facilities to specify and analyze different specifications, validate their conformance, and generate the final application.

The proposed methodology for software product line engineering was validated by applying it to three cases. Even tough the general applicability of a methodology hardly can be conclusively demonstrated by its use in a small number of specific

examples, we have the confidence that the domains and technologies used to demonstrate it are different enough to increase the degree of confidence in the soundness of the methodology. However, as is the case with most of the methodologies, we can not provide a list of the characteristics a domain should posses in order for the methodology to be useful.

Our contribution to existing methods and technologies was limited. For instance, the combination of feature diagrams and hierarchical models to engineer product lines had been suggested before in combination with a technique called *generic programming* [CE99]. The use of graphical environments to construct applications from components that were designed to participate in hierarchical compositions has been also used before [BCRW99]. Our main contribution in this part is the extensions of feature diagrams to represent constraints, and to hierarchical diagrams to emphasize how components are to be displayed in a graphical user interface to be manipulated by application developers. We also introduced a notation to emphasize that a component implements a type, characterized by an interface the component exports. These extensions simplify the process of product line engineering by making more visible the way every model contributes to the final implementation.

As demonstrated by our examples, the combination of these extensions with the approach of implementing product line infrastructures as configuration wizards, consistently produce successful results. These extensions and implementation approach were demonstrated to be useful in implementing three completely different product lines using different programming languages.

A more important contribution of our research is in software product line evolution. Experience in software development has shown that changes in requirements to applications are always necessary, and costly to implement. One difficult

problem is how to evolve a product line infrastructure to adapt it to new requirements.

Our proposed approach to product line evolution arises from the idea that even though every particular domain has its own characteristics, and its requirements change in different aspects, categories of changes at the product line architecture level can be identified [Bos00]. The particular changes can be associated to one category, thus similar architectural changes can be approached in a similar way, independent of the domain and implementation technology. Because our approach is consistently used along different domains and technologies, we are able to use a similar approach to evolve different product lines. This simple concept was demonstrated applying our approach to evolve two of the product lines we implemented.

To be able to describe different product lines in a similar way, we developed a specification notation that is based on feature diagrams and hierarchical models. Every product line is specified by describing components that can be instantiated, how these components are grouped in types, and if it is possible to select more than one component from a single type (if any). In that way, if a new component needs to be added to a product line that requirement is specified in the valid selections of a component type. If a component implementing a particular requirement needs to be removed, it is deleted from the specification to the product line. The specification of the product line is parsed to instantiate the infrastructure that is able to generate members of the product line. The two examples presented describe how such an approach is applied to evolve two of the product lines. Other changes in requirements that are not architectural can be approached in a different way; these changes are implemented in the algorithms, but the architecture does not change, and thus the product line does not change either.

As mentioned before, we can not affirm that just by presenting a finite number of experiments, can be one hundred percent secure of the validity of a methodology for product line engineering across different domains and programming languages. Besides, our research was affected from the fact that in two of the applications we were not able to demonstrate the product line in more than one instance. Due to these reasons, we still consider the methodology, and the way in which it was tested, have limitations. In the following sections we discuss contributions in more detail, and aspects that we consider are important and that can be classified as positive and negative, or as tasks that we did right or wrong for the research described in this dissertation.

8.2 Results and Contributions

Along this dissertation, software components were implemented to be reused in different compositions as units of reuse. To be handled as units, components encapsulate the set of properties (i.e., data and operations) characterizing every component. We showed that software product lines can be constructed from software components called wizlets, using a specification and composition tool called a configuration wizard. Both, wizlets and their configuration wizards have to be implemented for every product line, thus a programming language aimed at implementing configuration wizards needs to provide support for encapsulation and property (functionality) publishing.

One objective in our work was to demonstrate that software product lines could be implemented as configuration wizards, which provide enough support to gracefully evolve product lines by adding or removing requirements described as features. We provided a characterization of configuration wizards, proposed what a wizard can be, and introduced a methodology and technologies for implementing

and evolving software product lines as configuration wizards. Contributions in this dissertation can be summarized as follows:

- *Approach for software product lines as configuration wizards.* We proposed and demonstrated an approach for conducting product-line engineering based on configuration wizards. The approach consists in using feature diagrams for domain analysis, GenVoca for domain design, and wizlets and configuration wizards for implementation. The approach was tested in three different application domains to implement software product-lines.
- *Wizlets are parameterized components.* Modular and object-oriented programming languages not supporting component/class parameterization can be extended thus actual parameters can be specified at composition time. A configuration wizard implements a preprocessor for component adaptation. Semantic information prescribing environmental constraints can be implemented inside or outside a component. As our compositions are performed statically, we suggested that predicates constraining wizlet composition be implemented separately from wizlets and defined inside a knowledge repository. Predicates are evaluated at composition time for composition consistency. If everything is correct, wizlet's parameters are instantiated and the necessary code is generated, and then compiled into an application.
- *Specification interfaces include variant non-mandatory features.* Variants are special values or implementations of a type. As a result of our approach, the application developer needs to specify only those variants characterizing the application. The configuration wizard can insert mandatory features to the specification and propagate values that are used by several components.
- *Product-line evolution by meta-generator tools.* We propose a notation and approach to meta-specify a product line, and how that meta-specification can be used to produce a corresponding configuration wizard. The flexibility of

our approach is due mainly to how features are consistently used along the process, from domain models to design models, and finally to implementation.

To demonstrate that our approach can scale to different domains, we applied it to implement three different product lines in disparate domains, in the areas of simulators, real-time systems, and information management systems, respectively. Many application domains can be characterized as having similar requirements. Even though a formal theory of configuration wizards was not presented, we believe that our approach can be useful for developing product-line infrastructures for other domains.

Our experience gained in the process of implementing three configuration wizards using different programming languages in different domains, can be summarized as follows:

- *The participation of a domain expert is necessary.* Technical skills are not enough to implement product lines, a profound domain knowledge is mandatory to implement infrastructures for software product lines. A technical expert may lack the necessary domain knowledge. Models used in product line engineering should be useful to capture the domain knowledge and use resulting models to communicate with domain experts.
- *It may be difficult to find domain experts.* It is difficult to find all necessary domain knowledge in a single expert, several experts may be necessary. Participation of several domain experts is a plus for obtaining required information.

- *Difficulty in understanding components as parameterized units of reuse.* The necessity of different components implementing features seems natural, but how these features are translated to components, and how parameterize these components to be composed with others to construct applications is not straightforward.
- *Lack of programming languages implementing component parameterization.* Our work required extending programming languages supporting encapsulation but not component parameterization. A uniform extension approach we developed was fundamental to our work, thus hacking is reduced when different programming languages are used.
- *Necessity of tuning components.* Using a single approach for component composition sometimes requires that components have to be carefully planned. Component tuning is necessary to meet both performance and component substitution requirements.

8.3 Done Right

We demonstrated that our methodology can at least be successfully used to engineer three completely different domains using different implementation technologies. It can be argued that we were successful in the selection of domains and implementation technologies. However, it was the case they were the only choices we had at hand at the moment. The idea of product lines is attractive because its perceived benefits and its potentially broad applicability. But unfortunately we did not have at hand the level of domain knowledge on different domains or the possibility to be able to conduct a domain selection. Thus, we were not able to make a domain selection and choose domains we anticipated the approach could be useful.

Given that, we can consider the domains we implemented correspond to a random selection.

Other aspect of the methodology we propose is their architectural and thus component oriented nature. Our initial notion was that the approach should be amenable to be successfully implemented in any programming language that supports modularization and encapsulation facilities similar to those provided by classes of objects, at least. Parameterization capabilities, were absent, could be performed by a preprocessor previous to compilation. In two of the examples we implemented our product lines using programming languages that lack class or module parameterization (e.g., Java and object-oriented Pascal). Our proposed extension to programming languages was consistently used to represent component, class, and module parameterization; parameter instantiation was preformed by a preprocessor using a composition expression. The environment we obtained in this way, was able to completely generate the specified applications.

Our more complex endeavor was to demonstrate that the proposed methodology could be extended to be able to support product line evolution in a seamless manner. That goal was achieved by extrapolating the idea of using a specification to define the components that should be included in a specific application, to represent a product line infrastructure by a product line specification. We devised a notation to write a product line specification, and implemented a meta-generator to adapt components of a product line infrastructure to meet that product line specification. The result was that just by changing a product line specification, we can be able to produce the infrastructure that is able to generate applications that comply with the product line specification. This approach was demonstrated by implementing two meta-generators that were able to evolve two of the example product line infrastructures presented in this dissertation.

8.4 Done Wrong

There are several aspects limiting our proposed approach, in this section we discus how our methodology can not guarantee to be useful for every domain. The main limitations are the pragmatic nature of the approach against a more formal approach, the fact that we did not show any numbers describing how our approach can be compared to others, and lack of support in the study cases to demonstrate that our approach indeed is able to produce infrastructures that can generate members from a product line.

First is the fact that, besides the diagrammatic representations and specification notation, no formal notation was introduced to specify a product line. Our approach is pragmatic in nature; however, a formal notation could have been used to specify how the consistency in a product line is preserved by the introduction or removal of components. In principle, the constraints could be extended thus such validation can be performed.

Another limitation is that we were unable to show numbers to compare our approach to other approaches. Although presenting numbers could be a more conclusive demonstration of a quality, our approach was demonstrated by feasibility. Particularly, the domain of computer numerical control systems required that products comply with strict finishing standards. The manufactured products the generated system produced met these standards, thus demonstrating that generated systems could be able to satisfy the standards. In that domain, only carefully tuned applications had been able to meet such standards.

As a final remark, we observe that our methodology can not be demonstrated to be conclusive (i.e., applicable to every case). However, we wonder if any pragmatic approach can demonstrate that. The lack of a formal method always lacks of some kind of certainty. However, the case studies we conducted demon-

strate that, at least for domains with similar characteristics, and for programming languages with equivalent mechanisms to those we applied our methodology, the application of our approach could produce results similar to those we obtained.

8.5 Future Research

In this dissertation we proposed and explored configuration wizards as tools to implement and evolve product lines and an approach to conduct product-line engineering based on configuration wizards. There are different aspects that could be researched prior to an industrial use of our approach.

- *Further validate the approach for configuration wizards.* Although we presented three configuration wizards in three disparate domains, this by no means is a conclusive demonstration of their possibilities. Other domains may require more complex component interactions than parameterized inheritance allows.
- *Explore other modularization mechanisms.* The main mechanism used for wizlet implementation was encapsulation and inheritance. However, there seems to be no reason why association (e.g., class aggregation and composition) is not equally good for implementing wizlets. Instead of being mixed, components are instantiated independently and communicate using programming language function calls or operating system remote procedure call mechanisms. If appropriate, such possibility will extend the use to other programming languages not supporting inheritance.
- *Bundling code and predicates inside generative components.* Components and configuration predicates suggested in this dissertation were implemented separately; predicates are executed at composition time and are stored in a knowledge base, while application code is maintained as source code in a

component repository. It can be interesting to construct and analyze components including both predicates and a generator that produces the application code; among the benefits that compiled wizlets may offer are application code optimization. Static components contain the code that is used every time, compiled components can produce different code, when the component participates in different compositions. Such variations may be due to performance reasons, thus the resulting applications can perform better.

- *Other domains.* Distributed applications frequently require a variable number of components (i.e., components are added and deleted at run-time). Configuration wizards verify compositions statically, thus other mechanisms for validation may be necessary.
- *Organizational issues.* All projects presented in this dissertation were developed by a single person. For an industrial implementation of the approach, where a team or group of collaborating teams work on a single project, organizational aspects have to be researched. It is also necessary to define roles and responsibilities of team members, thus the different aspects involved in our approach can be assigned to people with the appropriate knowledge.

8.6 Recap

The increased demand for new applications can not be solved in traditional ways. Instead, we believe that automated application generation and evolution is necessary in well known domains. Once the requirements in a domain are well understood and applications in that domain characterized, automated facilities can be implemented. In this dissertation we proposed a methodology for software product line implementation. We presented three study cases we conducted to validate our methodology. As our intention was to demonstrate the feasibility of the

methodology to meet functional requirements, we concentrated on meeting functional requirements, without analyzing other aspects (e.g., high performance, security, etc.). Thus functional requirement satisfaction was our goal, and do not demonstrate that the approach was able to surpass what an expert programmer can be able to implement in a domain. As we were able to satisfy all requirements, then we consider our goal was achieved.

We realize that certainly there may be domains whose requirements could not be met by our approach. We did not conduct any research to find requirements that could not be met using the proposed approach. At least, we can be sure that software product lines can be implemented in domains with requirements similar to those we presented.

Bibliography

[AAG93] G. Abowd, R. Allen, D. Garlan, "Using Style to Understand Descriptions of Software Architectures", *Proceedings of SIGSOFT '93*, 1993, 9-20.

[ABM00] C. Atkinson, J. Bayer, and D. Muthig. "Component-Based Product Line Development: The KobrA Approach", in *Software Product Lines: Experience and Research Directions*, P. Donohoe, Kluwer Academic Press, 2000, 289-309.

[AFM97] O. Agesen, S. Freund, and J. Mitchell, "Adding Type Parameterization to the Java Language", *OOPSLA 1997*, 49-65.

[AG96] K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, 1996.

[Ara94] G. Arango, "Domain analysis method". In *Software Reusability*, Ellis Horwood, 1994.

[ASC00] Advanced Separation of Concerns, Workshop, OOPSLS'2000. http://trese.cs.utwente.nl/Workshops/OOPSLA2000/

[Bas97] P.G. Basset, *Framing Software for Reuse: Lessons from the Real World*, Yourdon Press, 1997.

[Bat88] D. Batory. "Concepts for a DBMS synthesizer". In *Proceedings of ACM Principles of Database Systems Conference*, Also in R. Prieto-Díaz and G. Arango, editors, *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, 1991.

[Bat97] D. Batory, "Intelligent Components and Software Generators", Invited presentation to the *Software Quality Institute Symposium on Software Reliability*, Austin, Texas, April 1997. Technical Report 97-06, Department of Computer Sciences, University of Texas at Austin, February 1997.

[Bat98] D. Batory, "Product-Line Architectures". Invited presentation, *Smalltalk un Java in Industrie und Ausbuildung*, Erfurt Germany, October 1998.

[Bat00] D. Batory, "Refinements and Separation of Concerns." *2nd Workshop on Multi-Dimensional Separation of Concerns*, International Conference on Software Engineering, Limerick, Ireland, 2000.

[Bax92] I. Baxter, Design Maintenance Systems. In *Communications of the ACM*, April 1992, pages 73-89.

[BB99] R.J.A. Buhr and D.L. Bailey, *An introduction to real-time systems : from design to multitasking with C/C++*, Prentice Hall, 1999

[BBB+00] F. Bachman, L. Bass, C.Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau, "Volume II: Technical Concepts of Component-Based Software Engineering", Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon, University, May 2000.

[BC90] G. Bracha and W. Cook, "Mixin-Based Inheritance", ECOOP/ OOPSLA 90, 303-311.

[BCGS95] D. Batory, L. Coglianese, M. Goodwin, and S. Shafer, "Creating Reference Architectures: An Example from Avionics", In [Sam95], pages 27-37.

[BCK98] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998.

[BCRW99] D. Batory, G. Chen, E. Robertson, and T. Wang, "Design Wizards and Visual Programming Environments for GenVoca Generators", *IEEE Transactions on Software Engineering*, 1999.

[BCS00] D. Batory, R. Cardone, and J. Smaragdakis, "Object-Oriented Frameworks and Product Lines". In *Software Product Lines: Experience and Research Directions*, P. Donohoe (editor), Kluwer Academic Press, 2000, 271-288.

[BFK+99] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, J.M. DeBaud, "PuLSE: A Methodology to Develop Software Product Lines", *SSR'99*, Los Angeles, California, USA, 1999.

[BG97] D. Batory and B. Geraci, "Composition Validation and Subjectivity in GenVoca Generators". In *IEEE Transactions on Software Engineering* (Special Issue on Software Reuse), February 1997, pages 67-82.

[Big92] T. Biggerstaff, "An Assessment and Analysis of Software Reuse", *Advances in Computers*, 1992.

[Big94] T. Biggerstaff, "The Library Scaling Problem and the Limits of Concrete Component Reuse", *International Conference on Software Reuse*, November 1994.

[BK94] G. Boothroyd and W. Knight, *Fundamentals of Machining and Machine Tools*, Marcel Dekker, Inc., 1989.

[BO92] D. Batory and S. O'Malley. "The Design and Implementation of Hierarchical Software Systems with Reusable Components". In *ACM*

Transactions on Software Engineering and Methodology, 1(4): 355-398, October 1992.

[Bos99] J. Bosch, "Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study", *Software Architecture*, Kluwer Academic Publishers, 1999.

[Bos00] J. Bosch, *Design and Use of Software Architectures : Adopting and Evolving A Product-Line Approach*, Addison Wesley, May 2000.

[BOSW98] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler, "Making the future safe for the past: Adding Genericity to the Java Programming Language", OOPSLA 98, Vancouver, October 1998.

[BR87] T. Biggerstaff and C. Ritcher, "Reusability framework, assessment and directions." *IEEE Software*, pages 41-49, March 1987.

[Bra92] G. Bracha, *The Programming Language JIGZAW: Mixins, Modulatiry and Multiple Inheritance*, PhD Dissertation, Department of Computer Science, The University of Utah, March 1992.

[BRJ98] G. Booch, J. Rumbaugh, and J. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1998.

[BSST93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. "Scalable Software Libraries". In *ACM SIGSOFT*, December 1993.

[BST+94] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. "The GenVoca Model of Software-System Generators". In *IEEE Software*, September 1994.

[CE99] K. Czarnecki and U.W. Eisenecker, "Components and Generative Programming", *SIGSOFT 1999*, LNCS 1687, Springer-Verlang, 1999.

[CE00] K. Czarnecki and U.W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[Cha02] Gary J. Chastek (editor), *Software Product Lines*, Proceedings of the Second Software Product Lines Conference (SPLC2), Aug. 19-22, San Diego,USA, Springer Verlang, 2002.

[Cox85] B. Cox, "Component-IC's", *BYTE Magazine*, May, 1985.

[CN98] S. Cohen and L. M. Northrop, "Object-Oriented Technology and Domain Analysis", Proc. Fifth International Conference on Software Reuse, June 2-5, 1998. Victoria, Canada.

[CSPK91] S. Cohen, J. Stanley, A. S. Peterson, and R. Krut, "Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain", *Technical Report CMU/SEI-91-TR-28*, Software Engineering Institute, Carnegie-Mellon University.

[CUO02] Credit Union Online Library, `www.cybercu.org`.

[DFK98] J.M. DeBaud, O. Flege, and P. Knauber, "PuLSE-DSSA - A Method for the Development of Software Reference Architectures", *SSR'98*, Orlando, Florida, USA. 1998.

[DJ00] G. Dudek and M. Jenkin, *Computational Principles of Movile Robotics*, Cambridge University Press, 2000.

[Don00] P. Donohoe (editor), *Software Product Lines: Experience and Research Directions*, Proceedings of the First Software Product Lines

Conference (SPLC1), Aug. 28-31, Denver,USA, Kluwer Academic Pub., 2000.

[DP98] D. D'Souza and J. Perlis "Frameworks",Addison Wesley,1998.

[DW98] D.F. D'Souza and A.C. Willis, *Objects, Components, and Frameworks with UML: The Cathalysis Approach*, Addison Wsleyy, 1998.

[El194] J.R. Ellis, *Objectifying real-time systems*, SIGS Books, 1994

[Edw95] S. Edwards. "Representation Inheritance: A Safe Form of White Box Code Inheritance". *Technical Report OSU-CISRC-9/95-TR38*, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, Sep 1995.

[Esh98] R. Eshuis. *Refinement in object-oriented analysis and design.* Master's thesis, University of Twente. August 1998.

[FJ99] M.E. Fayad and R.E. Johnson, *Domain-Specific Application Frameworks: Frameworks Experience by Industry*, John Wiley and Sons Inc., 1999.

[GFD97] M.L. Griss, J. Favaro, and M. d'Alessandro, "Featuring the Reuse-Driven Software Engineering Business", Object Magazine, September, 1997.

[GFD98] M.L. Griss, J. Favaro, and M. d'Alessandro, "Integrating Feature Modeling with the RSEB", Proc. Fifth International Conference on Software Reuse, June 2-5, 1998. Victoria, Canada. IEEE Computer Press pp. 354-355.

[GH91] G. Genebro and S. Heineman, *Machine Tools: Processes and Applications*, Prentice Hall, 1991.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GK96] M. Griss and R. Kessler, "Building Object-Oriented Instrument Kits", *Object Magazine*, April 1996.

[Gog86] J. Gogen, "Reusing and Interconnecting Software Components", *Computer*, February 1986, 16-28.

[Gri00a] M.L. Griss, "Implementing Product-Line Features by Composing Aspects", in *Software Product Lines: Experience and Research Directions*, P. Donohoe (editor), Kluwer Academic Press, 2000, 271-288.

[Gri00b] M.L. Griss, "Implementing Product-Line Features by Component Reuse", *Proc. of 6th International Conference on Software Reuse,* Springer-Verlang, Vienna, Austria, June 2000.

[GS93] D. Garlan, M. Shaw, "An Introduction to Software Architecture", *Advances in Software Engineering and Knowledge Engineering*, Volume I, World Scientific, 1993.

[GW94] M. Griss and K. Wentzel, "Hybrid Domain-Specific Kits for a Flexible Software Factory", In *Proceedings of SAC'94*, Reuse and Reengineering Track, Phoenix Arizona, March 1994.

[HO93] W. Harrison and H. Ossher. "Subject-Oriented Programming (A Critique of Pure objects)". In *OOPSLA '93 Conference Proceedings*:

Object-Oriented Programming Systems, Languages and Applications, Washington, DC, September 26 - October 1, 1993, pages 411-428.

[HOSM95] W. Harrison, H. Ossher, R. Smith, and H. Mili. "Subjectivity in Oriented-Oriented Systems: Workshop Summary." In *Addendum to OOPSLA'95 Conference Proceedings*: Object-Oriented Programming Systems, Languages and Applications, Austin, Texas, October 1995..

[JB97] G. Jiménez-Pérez and D. Batory, "Memory Simulators and Software Generators", In *Proceedings of the Symposium on Software Reusability*, Boston, Mass., May 1997.

[JF88] R. Johnson and B. Foote. "Designing Reusable Classes". In *Journal of Object-Oriented Programming*, June/July 1988, Volume 1, Number 2, pages 22-35.

[JGJ97] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture Process and Organization for Business Success*, Addison-Wesley, 1997.

[JCJO93] I. Jacobson, M. Christerson, P. Jonsson, and G. Övargaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley. (Revised 4th printing, 1993).

[Joh97] R. Johnson, "Frameworks = (Components + Patterns)", *Communications of the ACM*, 40(10): 39-42, October 1997.

[JRV00] M. Jazayeri, A., and F. Van Der Linden, *Software Architecture for Product Families: Principles and Practice*, Addison Wesley, May 2000.

[KCH+90] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study",

Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[Kie96] R. Kieburtz, L. McKinney, J. Bell, J. Hook, A. Kotov, J. Lewis, D. Oliva, T. Sheard, I. Smith, and L. Walton , "A Software Engineering Experiment in Software Component Generation", *International Conference on Software Engineering*, 1996.

[Kic00] G. Kiczales, "AspectJ™: Aspect Oriented Programming Using Java™ Technology". JavaOne, June 2000.

[KKL+98] K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, andM. Huh, "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures", *Annals of Software Engineering*, V5, pp. 143-168, 1998.

[KLM + 97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.Loingtier, and J. Irwin, "Aspect-Oriented Programming", *ECOOP 1997*, 220-242.

[KMS+00] P. Knauber, D. Muthig, K. Schmid, and T. Widen, "Applying Product Line Concepts in Small and Medium-Sized Companies", *IEEE Software*, Sep/Oct 2000, pp. 88-95.

[KS97] C.M. Krishna, K.G. Shin, *Real-time systems*, McGraw-Hill, 1997.

[KT01] T. Katayama and N.T. Thang, "Evolution in Collaboration-based Methodology, Workshop on Engineering Complex Object-Oriented Systems for Evolution", *OOPSLA 2001*.

[Lak96] J. Lakos, *Large-Scale C++ Software Design*, Addison-Wesley, 1996.

[Lat93] J.C. Latombe, *Robot Motion Planning*, Kluwer Academic Publishers; 1993.

[LC96] K.D. Larson & B. Chiappetta, *Fundamental Accounting Principles*, Mc Graw Hill, 1996.

[Lew95] T. Lewis, et. al., *Object-Oriented Application Frameworks*, Manning, 1995.

[LKCC00] K. Lee, K.C. Kang, W. Chae, and B.W. Choi, "Feature-Based Approach to Object-Oriented Engineering of Applications for Reuse", *Software Practice and Experience*, Vol. 30, Issue 9, pp. 1025-1046, 2000.

[LKK+00] K. Lee, K.C. Kang, E. Koh, W. Chae, B. Kim, and B.W. Choi, "Domain-Oriented Engineering of Elevator Control Software", in *Software Product Lines: Experience and Research Directions*, P. Donohoe, Kluwer Academic Press, 2000, 3-22.

[MBL97] A.C. Myers, J.A. Bank, and B. Liskov. "Parameterized Types for Java". *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[MBY+00] J. Michaloski, S. Birla, C.J. Yen, R. Igou, and G. Weinert, "An Open System Framework for Component-Based CNC Machines", ACM Computing Surveys, Vol.32, No.1es, March 2000.

[McI68] M. McIlroy. "Mass-produced software components". In [Nau68].

[ML98] M. Mezini and K. Lieberherr, "Adaptive Plug-and-Play Components for Evolutionary Software Developement", OOPSLA'98, 97-116.

[MS96] D. Musser and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Addison-Wesley, 1996.

[NM95] O. Nierstrasz and T. Meijler, "Research Directions in Software Composition", *ACM Computing Surveys*, 27(2):262-264, June 1995.

[OH92] H. Ossher and W. Harrison, "Combination of Inheritance Hierarchies", In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications - OOPSLA'92.*

[OKK+95] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal, "Subject-Oriented Composition Rules", In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications - OOPSLA'95.*

[OSA] OSACA Consortium - Open System Architecture for Controls within Automation Systems, Internet: http://www.isw/uni.

[OSE97] OSEC-II Project Technical Report, "Development of OSEC(Open System Environment for Controller" OSE Consortium, March 1997.

[Par72a] D.L. Parnas. "A Technique for Software Module Specification with Examples", Communications of the ACM, Vol 15, No. 5, May. 1972.

[Par72b] D.L. Parnas. "On the Criteria To Be Used in Decomposing Systems into Modules", Communications of the ACM, Vol 15, No. 12., Dec. 1972.

[Par79] D.L. Parnas. "Designing Software for Ease of Extension and Contraction". In *IEEE Transactions on Software Engineering*, March 1979, pages 128-138.

[PW92] D. Perry, A. Wolf, "Foundations for the Study of Software Architecture", *Proceedings of ACM SIGSOFT*, October 1992, 40-52.

[PY98] K.M. Passino and S. Yurkovich, *Fuzzy Control*, Addison-Wesley Longman, 1998.

[Ram98] M. Ramirez-Cadena, "Open Low-Cost Universal Numeric Controller", Master Thesis, ITESM, México 1998.

[RG98] D. Riehle and T. Gross, "Role Model Based Framework Design and Integration", In *Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages, and Applications - OOPSLA'98*.

[RJ97] D. Roberts and R. Johnson, "Evolving Frameworks: A Pattern Language for Developing Frameworks", in D. Riehle, F. Buschmann, and R. Martin, Eds., *Pattern Languages of Program Design 3*, Addison-Wesley, 1997.

[RP01] M. Riebisch and I. Philippow, Evolution of Product Lines Using Traceability, Workshop on Engineering Complex Object-Oriented Systems for Evolution, OOPSLA 2001.

[SB97] Y. Smaragdakis and D. Batory, "DiSTiL: a Transformation Library for Data Structures". *USENIX Conference on Domain-Specific Languages (DSL97)*

[SB98] Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers", ECOOP 1998.

[SEI01] "Framework for Software Product Line Practice - Version 3.0", Software Engineering Institute, Carnegie Mellon University, http://www.sei.cmu.edu/plp/framework.html

[SG96] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

[SGB01] M. Svahnberg, J. van Gurp and J. Bosch, On the Notion of Variability in Software Product Lines, Proceedings of The Working IEEE/IFIO Conference on Software Architecture (WICSA 2001), The Netherlads.

[Sim95b] M. A. Simos, "Organization Domain Modeling (ODM): Formalizing the Core Domain Modeling Life Cycle", *Simposium on Software Resusability, Seattle*, Washington, USA, 1995.

[Sin96] V. Singhal, *A Programming Language for Writting Domain-Specific Software Systems*, PhD dissertation, Department of Computer Sciences, The University of Texas at Austin, August 1996.

[Sma99] Y. Smaragdakis, Phd Dissertation, Department of Computer Sciences, The University of Texas at Austin, September 1999.

[SMB00] R.C. Seacord, D. Mundie, and S. Boonsiri, "K-BACEE: A Knowledge-Based Automated Component Ensemble Evaluation Tool", *Technical Note, CMU/SEI-2000-TN-015.* Software Engineering Institute, Carnegie Mellon University.

[SSS00] K.F. Skousen, E.K. Stice, and J.D. Stice, *Intermediate Accounting*, South-Western College Publishing, 2000.

[Sta00] J.A. Stankovic, "VEST: A Toolset For Constructing and Analyzing Component Based Operating Systems For Embedded and Real-Time Systems", *Technical Report TR CS-2000-19*, University of Virginia, July 2000.

[Szy98] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[Tek94] Architecture-Based Acquisition and Development of Software Guidelines and Recommendations from the ARPA Domain-Specific (DSSA) Program. Technical report, Teknowledge Federal Systems, October 1994.

[Tho98] J. Thomas, *P2: A lightweight DBMS Generator*, Ph. D. Dissertation, Department of Computer Sciences, The University of Texas at Austin, December 1998.

[Umh01] M. Umholtz, Six CU sucess characteristics for 2002, *Credit Union Executive Journal*, Vol. 11, No. 6, pp. 26-29, Sep/Oct 2001.

[Van02] F. van der Linden, "Software Product Families in Europe: The Esaps and Café Projects", IEEE Software, Vol. 10, No. 4, pp. 41-49, Jul/ Aug 2002.

[VAM+98] A. D. Vici, N. Argentieri, A. Mansour, M. d'Alessandro, and J. Favaro, "FODAcom: An Experience with Domain Analysis in the Italian Telecom Industry", *Proc. Fifth International Conference on Software Reuse*, June 2-5, 1998. Victoria, Canada.

[VN96a] M. VanHilst and D. Notkin. "Using C++ Templates to Implement Role-Based Designs". *JSSST International Symposium, ISOTAS '96: Proceedings*, Japan, Springer-Verlag,1996.

[VN96b] M. VanHilst and D. Notkin. "Using Role Components to Implement Collaboration-Based Designs", OOPSLA 1996.

[Wit96] J. Withey, "Investment Analysis of Software Assets for Product Lines". Technical Report, CMU/SEI-96-TR-010, Software Engineering Institute, Carnegie Mellon University.

[WJB95] P. Wilson, M. Johnstone, ad D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review", In *International Workshop on Memory Management*, September 1995.

[WKK93] J.J. Weygandt, D.E. Kieso, and W.G. Kell, *Accounting Principles*, John Whiley & Sons, Inc., 1993.

[WL99] D. M. Weiss and C. T. R. Lai, *Software Product-Line Engineering*, Addison-Wesley, 1999.