BROOD - Business Rule-Driven Object-Oriented Design

A thesis submitted to the University of Manchester for the degree of
Doctor of Philosophy
in the Faculty of Humanities

2005

Wan Mohd Nasir Wan Kadir

School of Informatics

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

List of Contents

List of Tables

List of Figures

Abstract

In order to remain useful, it is important for software to evolve in accordance with the changes in its business environment. Most of the current approaches to software evolution focus primarily on improving software technology.

This research is based on the premise that a more effective solution to software evolution can be achieved by considering the sources of changes in addition to software technologies. The thesis identifies a major source of changes in the *business rules* that dictate how the system's environment (i.e. the business) needs to function. Evolution of software is therefore considered in this thesis as being driven by changes to business rules. Therefore, an important consideration is the linking between business rules as specified in the business ontology and the way that these rules manifest themselves in software architectures.

Current state of the art considers business rules mostly at the conceptual level but lack the necessary constructs, techniques and tools to link these specifications to software components. The few software evolution approaches that include business rules in their solution lack the suitable concepts and structure of business rules. The research presented in this thesis attempts to ameliorate these shortcomings by developing the **B**usiness **R**ule-Driven **O**bject-**O**riented **D**esign (**BROOD**) approach. The thesis presents both product and process components. The product component is structured according to a metamodel that defines the semantics and syntax of business rules statements and links these rules to their related software design components. The Unified Modelling Language (UML) was adopted to define the software design part of the metamodel. The process component describes a flow of activities that guide the development and evolution of a software system, which is specified using the Software Process Engineering Metamodel (SPEM).

The BROOD approach is demonstrated using the industrial strength MediNET application. MediNET was also used throughout this research, such as in understanding the state-of-the-art and improving the BROOD metamodel. The software tool prototype was also developed to demonstrate the potential of automating the important tasks in the BROOD approach. It was developed on top of the configurable Generic Modeling Environment (GME).

Declaration

I hereby declare that no portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright Statement

Acknowledgement

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. Firstly, I am deeply indebted to my supervisor Professor P. Loucopoulos from the School of Informatics, The University of Manchester whose help, stimulating suggestions, and encouragement helped me during my study years in Manchester, and from whom I've received a great balance of freedom, monitoring, and guidance in gaining a challenging yet enjoyable learning experience of doing research.

Secondly, I would like to pay tribute to the Universiti Teknologi Malaysia for sponsoring my PhD study. I also would like to thank Penawar Medical Group for providing me with the information on WebCare and MediNET especially Dr. A. Sulaiman (CEO), Masrizam (IT Manager), and Mohzasiraj (Software Engineer).

Thirdly, I wish to thank all academic and support staff from the School of Informatics for assisting me in academic, administrative, technical, and socializing matters especially Dr. Petrounias, Shermain, Willey, and Paul. I also want to thank Taiseera for offering her generous help with proof-reading this thesis, and all of my friends (too many to mention) in K13, K12, and K10 research laboratories for creating 'short breaks' during my work hours.

Last, but not least, I would like to give my special thanks to my wife Rasidah, whose patient and love enabled me to complete this work, as well as my sons Zakwan and Syazwan, whose laugh makes me forget about the pressures of doing this research.

For Rasidah,

Zakwan and Syazwan

Chapter 1

Introduction

This chapter provides an introduction to the research work presented in this thesis. It describes the research background that motivates the introduction of a new business rule-driven approach to software evolution. This is followed by the discussion on the premises, hypothesis, aim, and objectives of the reported research work. Subsequently, it summarizes the main contributions of the research work in the related fields of research and practice. Following this, it briefly explains the description and roles of the selected case study namely the MediNET application. Finally, it introduces the structure of the thesis.

1.1 Motivation

Nowadays, nearly all of commercial and government organizations are highly dependent on software systems. Due to the inherent dynamic nature of their business environment, software evolution is inevitable. The changes generated by business policies and operations are propagated onto software system. A large portion of total software lifecycle cost is devoted to introduce new requirements, and remove or change the existing requirements [Grubb and Takang, 2003]. However, software evolution must be accomplished; otherwise no one will use the software [Lehman, 1997]. In other words, the evolution of a software system is inevitable for the software to remain useful in its environment. Due to this reason, software evolution is considered as a key research challenge in software engineering.

Many research projects attempt to find a more applicable way for building a software system that is flexible to changes as well as predicting the effect of requirements change [Finkelstein and Kramer, 2000]. Most of the approaches in software evolution utilize or adapt the existing benefits in object-oriented, distributed system, software architecture, and component-based technologies. They strive to propose a software model, or architecture, that has the ability to reduce the evolution efforts or minimize the effect of changes [Bennett and Rajlich, 2000; Garlan, 2000]. These approaches consider 'design for change' as their main goal in software modelling. Some of them also define a software process that provides a detailed flow of activities in performing the development and evolution of a resilient software system.

However, there are at least three problems observed by the investigation of the current software evolution approaches. First, most of them focus more on software technology, ignoring the consideration of the sources of changes in a software operational environment. Second, the majority of them confine their solutions at a concrete modelling level although more evolvable software can be achieved by systematically addressing the problem at the metamodel level. Finally, they generally define software evolution as post-implementation activity even though there are changes occurred prior to implementation especially in today's 'emergent organization' i.e. the organization that is 'continually changing and never arriving at stable state' [Truex et al., 1999].

Business rules, which are frequently changing in accordance with the business changes [Rouvellou et al., 2000], have been identified as the important sources of changes. In addition, their changes bring the highest impact on both software and business processes compared to other changes such as altering code for elegance and readability, changing data naming convention, speeding execution, or reducing internal storage usage [Andrade and Fiadeiro, 2001; Chapin et al., 2001]. The explicit consideration of business rules in software development is important in assisting future evolution [Loucopoulos et al., 1991].

The advantages of externalizing business rules lead to the emergence of numerous business rule approaches to software development. These approaches vary in terms of the roles of business rules in the software lifecycle. For example, some approaches address the use of business rules throughout the software lifecycle while others merely consider the role of business rules in a particular stage of software development. However, they share a common goal i.e. to externalize business rules for future software evolution. Business rule externalization may localize changes to a volatile part of software systems. It may also simplify the maintenance of user requirements since business rules are also considered as part of the requirements [Rosca and Wild, 2002].

From the investigation of the state-of-the-art business rule approaches to software evolution, there are two identified prominent categories that are closely related to the purpose of this research: *business rule conceptual modelling* and *evolvable software systems*. The former provides the excellent typology and structure that help in capturing and representing business rules but the approaches in this category supply little detailed information on the implementation of business rules in a software system. The latter provides the exhaustive explanation on the design and implementation of an evolvable software system that implements business rules but the approaches in this category lack the business rule modelling concepts. The gap that exists between these two categories indicates that serious attention is needed to link the business rule specification and its related software components for the purpose of future evolution [Appleton, 1984; Kovacic, 2004].

The approach proposed in this thesis, which is called **B**usiness **R**ule-Driven **O**bject-**O**riented **D**esign (BROOD), attempts to close the gap between the above two categories

as well as to address the earlier mentioned problems in software evolution. In achieving a holistic approach to software evolution, BROOD tackles both product and process perspectives. Regarding product perspective, BROOD introduces a metamodel that defines the structure of business rules, software design, and their linking elements. Unified Modelling Language (UML) was chosen to represent the software design part of the metamodel. Concerning process perspective, BROOD defines a software process that describes its development and evolution activities. The process is described using Software Process Engineering Metamodel (SPEM). In addition, the automated tool that supports BROOD is developed on top of the configurable Generic Modeling Environment (GME).

The applicability of the BROOD approach is demonstrated using the industrial strength MediNET application. MediNET is an Internet application that is used by healthcare industry community such as healthcare providers and paymasters to manage the patient registration, patient billing, and paymaster invoicing. The learning outcomes of this study provide valuable insight into several issues and implications of practising BROOD in an industrial application setting, and contribute to a deeper understanding of the software evolution issues. Throughout this research, MediNET is also used to obtain the feedback in the BROOD application that repeatedly improves the BROOD approach.

1.2 Thesis Aim and Objectives

Summarizing the problem with the state-of-the-art approaches, a holistic approach to software evolution may be achieved if the approach not only confines the solutions to the software technologies such as object-oriented abstractions or software architectures, but it also considers the sources of changes i.e. business rules. This fact leads to the first premise of this research i.e. *the explicit consideration of business rules in software modelling, in addition to the adopted software technologies, may improve the evolvability of a software system based on the fact that business rules are the most volatile component in a business information system.*

Apart from the above solution components, it is also important for the approach to provide a software process that guides the development and evolution of a software

system. Obviously, the proposed software process will be more complicated than the traditional software process since it needs to include the additional activities that deal with the newly introduced business rule components. However, the availability of an automated tool may simplify the proposed software process. The roles of the software process and automated tool lead to the second premise of this research i.e. *the software process and automated tool may enhance the practicability of the proposed software evolution approach.*

The above premises serve as the principles of the conceptual framework adopted by this thesis, which is illustrated in Figure 1-1. In this framework, a holistic approach to software evolution considers all three components in producing a software system that is resilient to business changes: software technologies, business rules, and software process. *Software technologies* have been addressed by majority of the recent software evolution approaches. This research is advanced by considering *business rules* as sources of changes in order to reach the root of the evolution problem and to strengthen the intended technological solution. Both *software technologies* and *business rules* should be considered in defining the metamodel, which in turn determine the characteristics of the software model. The third component, i.e. the *software process* (and automated tool), improves the pragmatics aspects of the approach by guiding and facilitating the complex development and evolution tasks.

The above premises also lead to the following research hypothesis:

> The evolution of a business software system may be simplified by a practical holistic approach that (i) explicitly considers business rules in software modelling in addition to the adopted software technologies and (ii) provides a process and tool that facilitate the development and evolution activities.

The above hypothesis is subsequently transformed to the aim and objectives that systematically set the direction of this research. The aim of this research is to improve software evolution.

To this end, the objectives of this research are listed as follows:

1. To analyse the state-of-the-art business rule approaches in software evolution.

2. To develop a metamodel that externalizes the representation of business rules and provides traceability to their implementation in software design.

3. To specify a software process that guides the development and evolution of a software system using the proposed metamodel.

4. To demonstrate the practicability of the proposed approach in an industrial strength software application.

5. To develop a software tool prototype that provides facilities that make BROOD practically applicable.

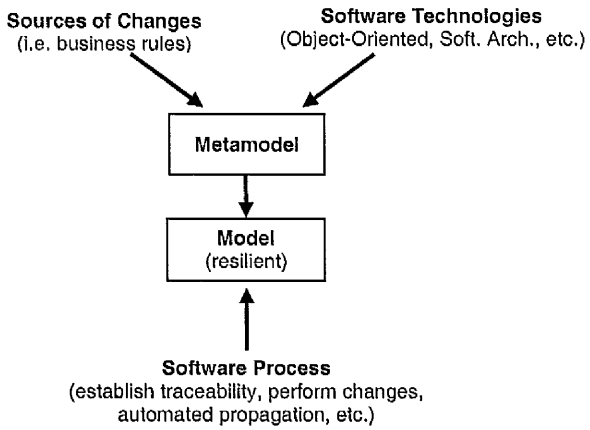

Figure 1-1 A framework of a holistic approach to software evolution

This research considers evolution in the pre-implementation stages of a software system. It assumes that the technique to transform the design to software implementation is available. In addition, this research addresses the evolution of business rule changes relating to existing business components. It does not deal with the other types of changes or with undocumented business rules.

1.3 Research Contributions

BROOD is a holistic approach to software evolution that considers all of the influential factors in software evolution such as the sources of changes, software technology, and software process. It is similar to the holistic approach to health or treatment that deals with the whole person or system rather than treating isolated symptoms. Fixing only the

symptoms is similar to only focusing on software technologies and addressing the problems at the concrete modelling level or post-implementation phase, which were identified as the problems with the current approaches in software evolution.

The proposed metamodel that can be used as the guidelines to capture and specify business rules provides a more structured representation of user requirements. The linking elements of the metamodel improves the business rules traceability in software design, which in turn facilitate the propagation of business rule changes to their related software design components. The detailed description of the BROOD process provides a guideline in performing the development and evolution of a business rule intensive software system. The proposed approach is also supported by an automated software tool that simplifies the tasks of linking business rule specification to its related design components during development and evolving the software in accordance with business rule changes during evolution phase.

In particular, the contributions of this work to current business rules and software evolution research include:

1. investigation of the roles that business rules play in facilitating the software evolution process;
2. development of a metamodel that defines the typology and structure of business rules and links business rules to software design;
3. development of a software process that guides the development and evolution of a business rule intensive software system;
4. development of a tool prototype that demonstrates the automated feasibility of the proposed approach; and
5. reflection on the implications of practising the business rule-driven software evolution based on an industrial-strength application.

1.4 Industrial Application – MediNET

MediNET is a suite of Internet applications that addresses the administrative and back-end processing requirements of the healthcare business community. The author of the

thesis was involved in the original project of developing MediNET. Therefore, apart from being a typical information systems application, MediNet provided the opportunity for the author to compare in depth the traditional design to BROOD. MediNET acts as a secondary layer to the existing administrative and information systems. It allows various components of the healthcare industry to exchange business data instantaneously and automate their routine administrative tasks. Therefore, facilitated businesses are able to reduce their administrative burdens, become more efficient and make better informed business decisions. In contrast to the traditional applications, MediNET does not require its users to maintain separately installed software. It allows its users to leverage the power of technology without having to bear massive development, acquisition, infrastructure or maintenance costs. The MediNET users only need to pay as and when they use the application.

In general, MediNET users can be divided into three categories: paymasters, healthcare providers (HCPs), and supplier. Paymasters are those who pay for medical or healthcare services, for examples employers, insurers and managed care organizations. They use MediNET to maintain the basic parts of the patient records such as performing their payee registration and defining the healthcare benefit coverage of their payees. HCPs are the professionals who dispense medical treatment, for example general practitioners (GPs), hospitals and dentists. HCPs use MediNET to manage patient records, patient billing and paymaster invoicing. The current implementation of MediNET is only limited to employers as the paymasters and GPs as the HCPs. The supplier is the company who owns, provides and maintains the MediNET applications. It rents MediNET to HCPs and paymasters as and when the applications are needed and charges them based on the number of performed transactions. The detailed descriptions on MediNET are available in Appendix A.

MediNET was chosen as a case study due to the various frequently changing business rules introduced by its different users. For example, HCPs provide different packages to the paymasters that constrain the way they perform the billing and invoicing processes. Paymasters may also want to introduce different healthcare benefit coverage to different staff levels that control the eligibility of the staff's treatments. The business rules related to the packages and benefit coverage are frequently changed by the HCPs and

paymasters. Other common changes to business rules include the introduction of invoice discounts, the rules to block non-paying paymasters, and the conditions to issue reminder for the past due invoices. These frequent changes indicate the needs for the approach that may simplify the implementation of business rule changes in MediNET.

MediNET plays at least three important roles throughout this research. First, it is used to investigate and demonstrate the modelling concepts of the prominent state-of-the-art business rule approaches to software evolution. The use of MediNET as their examples provides useful information for the evaluation of these approaches, which in turn generates the inspiration to improve the existing approaches. Second, MediNET is used to provide a feedback for improving the proposed approach. The proposed metamodel, process, and tool are repeatedly used against MediNET to identify their applicability in the real-world application. Finally, it is used to demonstrate the work proposed in this thesis.

1.5 Structure of the Thesis

Chapters 2 to 7 of this thesis are organised as follows:

Chapter 2 investigates the state-of-the-art software evolution and business rule approaches. It presents the prominent views in software evolution and reviews the recent software evolution approaches in three different dimensions i.e. process, product, and elaboration. It also reviews the roles of business rules in various software engineering research areas and evaluates the approaches that are closely related to this research. The investigation on the state-of-the-art and its evaluation provide a foundation for the BROOD approach.

Chapters 3 and 4 present a new holistic software evolution approach, i.e. the BROOD approach. BROOD consists of both product and process components. **Chapter 3** presents the BROOD product component i.e. the metamodel that defines the business rules, software design and the linking elements. **Chapter 4** describes the process component i.e. the flow of activities that may be followed in developing a traceable business rule specification and performing business rule-driven software evolution.

Chapter 5 discusses the application of the BROOD approach in an industrial strength software application i.e. MediNET. It demonstrates the applicability of the BROOD metamodel and process in the development and evolution of the MediNET application. The analysis of the BROOD application on the selected change scenarios shows that the business rule-driven software evolution reduces the evolution efforts in MediNET.

Chapter 6 describes the BROOD tool prototype which is built on top of the configurable Generic Modeling Environment (GME). It describes the GME modelling concepts and architecture which build the modelling vocabularies used to design and implement the BROOD tool prototype. Subsequently, it discusses the implementation of the BROOD metamodel using the GME meta-metamodel and explains how the BROOD tool automatically performs the main BROOD activities.

Chapter 7 concludes the thesis by discussing the research achievements and overall contribution of the research in the context of related work in the area. In addition, it discusses the limitations of the approach and points to future research directions.

There are three appendices that are included at the end of this thesis to supply more detailed information about certain subjects discussed in this thesis. **Appendix A** provides the detailed description of the MediNET application including the business processes, user types, business entities, and database design of the MediNET system. **Appendix B** provides a complete specification of the BROOD metamodel, which is written in a context-free grammar definition using Extended Backus-Naur Form (EBNF). **Appendix C** presents the structured textual specification that provides a more detailed description of the BROOD process.

Chapter 2

Software Evolution and Business Rules: State-of-the-art

This chapter reviews the state-of-the-art approaches in software evolution and business rules. It starts with the overview of the popular views on software evolution and reviews the recent software evolution approaches. The chapter then proceeds with the review of the historical background and state-of-the-art business rule approaches to software development. This is followed by the review of the prominent approaches from two categories that are closely related to the current research work: business rule conceptual modelling and construction of evolvable software system. The results of the evaluation of these approaches using the proposed evaluation framework are presented in the following section. Finally, the chapter summary, which includes the explanation on the improvement opportunities of the current state the art business rule approaches to software evolution, is presented at the end of this chapter.

2.1 Introduction

In the effort to investigate the appropriate approach to developing evolvable software system, this chapter focuses on the review of the literature concerning software evolution. It is found that most software evolution approaches often have different views and address different specific problem areas. Due to this diversified nature, in the first part of this chapter, the author will discuss state-of-the-art approaches to software evolution based on three different dimensions i.e. *process*, *product* and *elaboration*. The three-dimensional view is not claimed to be mutually exclusive. It is merely used to systematically review the current trends in software evolution approaches, as well as to locate the proposed research in the field. A critique on the state-of-the-art software evolution approaches is given at the end of this part.

Having identified that business rules are the important component in software evolution, and they play a key role in rapidly changing business environment, business rules approaches are discussed in the second part of this chapter. Based on the extensive reviews on the available literature, the discussion on business rule approaches is organized into the roles of business rules in various areas of research and practice: software development lifecycle, object-oriented, and other research areas.

The above reviews of business rules and software evolution suggest that there are two different extremes of a spectrum in business rule approaches to the development of evolvable software systems: business rule conceptual modelling and evolvable software systems. Three prominent examples of approaches for each extremity are examined in the third part of this chapter. The comparative evaluation of these approaches is presented in the fourth part using the proposed evaluation framework that includes concept, modelling language, process, and pragmatics criteria. In closing the chapter, the author will draw up summary and further remarks on the reviewed literature.

2.2 Software Evolution

Currently, there is no standard view or generally accepted definition on the term software evolution. It is often used interchangeably with software maintenance [Bennett and Rajlich, 2000; Chapin et al., 2001]. For that reason, instead of reviewing the

terminology definitions, the author will summarize views and comments of other researchers, and elaborate his own view on software evolution at the end of this section.

Perry views software evolution as three fundamental dimensions i.e. the domains, experiences and process [Perry, 1994]. The domains are real world environments within the context of software systems such as business policies, government regulations, and technical standards. The experience is the knowledge gained from the implemented software system such as feedback and empirical study. The process is the way of building and evolving a software system such as methods, techniques and tools. All of the development and changes of these dimensions are considered as important sources of evolution. According to Perry, it is more effective to understand and manage the evolution of a software system by a deep understanding of these three dimensions.

Bennett proposed a slightly different view by considering software evolution as three-levels model, in terms of the organization, process and technology [Bennett, 1996]. The top level of this model emphasizes how software evolution meets the organizational goals and needs within a defined time scale and budget. The middle level is concerned with a set of activities for software evolution including initial request, change assessment, cost-estimation, and change impact analysis. At the lowest level is technology to support the process such as automated tools, languages, and notations.

The above process is organized in a staged model for software lifecycle. In short, the model represents the software lifecycle as a sequence of stages namely initial development, iterative evolution, iterative servicing and phase out stages [Bennett and Rajlich, 2000]. During the initial stage, software architecture and team knowledge are developed, which lead to the first version of a software system. The successful first version is then evolved to adapt the changing user requirements and operating environment at evolution stage. Servicing stage starts when there are no major changes, and minimum involvement of domain or software engineering experts. Finally, the software system that becomes less useful in its environment, and costly to maintain will enter the phase-out and close down stage.

The effort to classify software evolution and maintenance by Chapin et al. brings a useful different view to software evolution. Instead of using the traditional intention-

based classification such as corrective, adaptive, perfective, and preventive, they use the classification based on "*objective evidence of maintainers' activity*" [Chapin et al., 2001]. From their observations, they identify mutually exclusive software evolution types, and arrange them into four clusters namely support interface, documentation, software properties, and business rules. The degree of changes will determine which activity is belonging to which cluster. They conclude that different types of maintenance or evolution may have different impact on software and business processes as shown in Figure 2-1. Based on their findings, it can be observed that business rule changes brings the highest impact on both software and business processes.

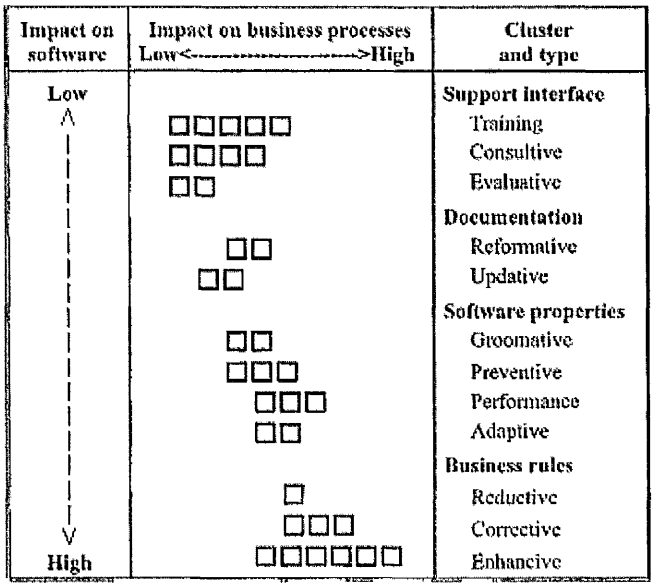

Figure 2-1 Impact of the types of software evolution and maintenance (cited from [Chapin et al., 2001])

Apart from lacking of standard definition of the terms software evolution, there is also a problem related to inconsistencies in terminology used to define the attributes of evolvable software systems; for example adaptability [Evans and Dickman, 1997; Liu, 1998], flexibility [Schneider, 1999], tailorability [Stiemerling et al., 1997], extensibility [Liu, 1998], customizability [Masuhara et al., 2000], scalability [Kon et al., 2000], maintainability [Burd and Munro, 1999], compatibility [Senivongse, 1999], changeability [Lam et al., 1999], and stability [Diaz et al., 1998]. These terminologies

often reflect the attributes of their proposed solutions to software evolution. Although there is a minor different in their definitions, they share the common aim i.e. to minimize the evolution (or maintenance) efforts of a software system.

In the following discussion, state-of-the-art approaches to software evolution will be investigated by naturally categorizing the current approaches into three main dimensions, as mentioned at the beginning of this chapter, namely process, product and elaboration. The summary and further remarks are presented at the end of this section. It is important to note that, in this thesis, the meaning of the term 'approach' may represent a solution that consists of all or some methodological elements such as method, technique, tools, process, or modelling language. It is more lenient than the term 'method' defined in method engineering field [Brinkkemper, 1996].

2.2.1 Process Dimension

In the current context, process is referred to a set of well-organized activities to accomplish certain tasks in order to produce well-defined results. Along this dimension, software evolution approaches can be further classified into three main groups i.e. development approaches, evolution management and evolution support.

Development approach provides a methodology for the development of software system for future evolution. Even though most approaches serve for this same purpose, each of them has distinctive flavour. For instance, DRASTIC aims at software evolution during execution by exploiting object persistence and distributed systems implementation [Evans and Dickman, 1997]. This approach is further refined by introducing the notions of zones, contracts and change absorbers [Evans and Dickman, 1999]. Perrochon and Mann introduce 'inferred design' to deal with rapid software evolution [Perrochon and Mann, 1999]. Unlike conventional software development approach, inferred design inverts the process by deriving design from the implementation. It is believed that this approach enables one to deal with rapid software evolution since the changes to implementation will cause the new model to be generated. Misra et al. take one step further by considering the modelling of the environment of the software. This approach is based on a 'meta-architecture' that provides a framework to software evolution using COTS and custom component [Misra et al., 1997a; Misra et al., 1997b]. Lied considers

a separation development of domain model at the beginning of development, which is called 'domain engineering', and uses this model to generate an application software [Lied, 1997]. EVOLVE uses adaptive schema and propagation pattern to make object-oriented design easy to change [Liu, 1998]. There are also approaches that specifically deal with legacy system. Mehta and Heineman address legacy system evolution using feature engineering to identify system features, and creates software component based on these features [Mehta and Heineman, 2001]. MORALE (Mission Oriented Architectural Legacy Evolution) that provides a complete cycle for evolving legacy system starts from eliciting change requirements up to performing the evolution [Abowd et al., 1997].

The next group that lies on process dimension is *evolution management*. Evolution management relates to a process of handling evolution activities and software artefacts. The general aims of evolution management are twofold, to record evolution information for future references as well as to reduce the complexity of the evolved software. For example, EMMA (Evolution-Memory Management Assistant) provides details of evolution information for several software releases [McCullough et al., 1998]. Mens uses graphs to manage evolution of software artefacts such as architectures and codes [Mens, 2000]. Ohlsson et al. reduce component complexity, keep track of software evolution and react before the system is difficult to maintain [Ohlsson et al., 2001]. Since evolution may jumble up the software components, it is important to maintain their traceability. There are approaches that address this problem by maintaining traceability links between software releases [Antoniol et al., 2001], visualizing traceability links, and using knowledge engineering to link source code and domain knowledge [Li et al., 2000b].

The final group in the current dimension is *support* for software evolution, which includes language [Shibayama et al., 2000], operating systems [Saranauwarat and Taniguchi, 2000] and tools support. Although most of the above mentioned development approaches come with their own software tools, it is worth to mention a number of tools for their distinctive features. For example, there are tools that transform existing software (legacy system) to a new programming paradigm such as from procedural to object-oriented program [Fanta and Rajlich, 1999]. This action is taken as

a preventive measure to deal with future evolution. Software tools like Aspect Browser aids evolution at higher level than codes by performing global software changes in a system that consist of separate modules [Griswold et al., 2001].

2.2.2 Product Dimension

Under the scope of product dimension, most of work on software evolution is predominantly associated to several distinct but closely related fields namely software architectures, distributed systems, object-oriented and component-based software system. Apart from these main fields, there are few observable trends in other fields such as agent-based computing and design patterns. Due to the limited space, this section discusses only the main fields.

In general, *software architecture* consists of components, connectors and organization of its components and connectors [Perry and Wolf, 1992; Shaw and Garlan, 1996]. These architectural constituents can be manipulated and further defined to achieve an evolvable architectural design, which in turn improves evolvability of a software system. For example, refining the role of connectors makes run-time evolution of software architectures feasible [Oreizy et al., 1998], and introducing good abstractions of the components for composition improves software evolvability [Andrade et al., 2002]. In product-line architectures, i.e. a set of software systems that share core product architecture, the architectural constituents are carefully identified and defined for a future product member evolution [Svahnberg and Bosch, 2000].

The effect of evolution in *distributed systems* is more complex since most of the components are distributed across different address spaces; however, many benefits can be gained from the inherent nature of distributed systems such as low coupling and high cohesion of their architectural components. Although the approaches directly, or indirectly, aim to utilize the distributed implementation features, their discussions still focus on software architectures. For example, 'mediator' is used as a middle component between changed server and older client in [Senivongse, 1999], change absorbers in DRASTIC architecture [Evans and Dickman, 1999], ports in [Magee et al., 1994], and actor 'liaison' in [Astley and Agha, 1998]. These architectural components attempt to absorb or reduce the effect of changes in distributed software systems.

There are also a number of related works on software evolution in *object-oriented* software system. Although object-oriented features such as encapsulation, inheritance and polymorphism are helpful in supporting software evolution, in practice they are not enough to support evolution of large, complex software systems. Therefore, the approaches in object-oriented software evolution emerge in divergent topics. For example, the formalization of object behaviour specification [Itou and Katayama, 2000], and the generalization of reuse contract formalism and its integration into Unified Modeling Language (UML) metamodel [Mens and D'Hondt, 2000] proved to be useful in supporting software evolution. Liwu uses UML extensions capability to express the interrelations between classes based on the notions of 'contract' and 'protocol'; contract specifies the services required and provided by a class whilst protocol register cooperation between classes [Li, 1999]. Liu increases adaptability of object-oriented design against requirement changes using adaptive schema style rules [Liu, 1998]. The rules are used to transform any object-oriented design into a more adaptable design. Diaz et al. provides method to explicitly identify, design and implement business policies in object-oriented software system [Diaz et al., 1998]. The explicit description of business policy is able to separate a volatile part from the stable part hence localizes change and support evolution. Hursch and Seiter propose a framework to detect and manage inconsistencies between objects and programs following object-oriented schema transformation [Hürsch and Seiter, 1996]. The consistencies are re-established via objects and programs transformation.

Component-based software is a software system that is developed by composition of reusable predefined, pre-tested software components. It gains popularity from software engineering practitioners for shorten time to market. Recent work shows that, with some improvements, component-based software may benefit evolution. For instance, the separation of components, connectors and configuration [Schneider, 1999], the decomposition into a set of components based on business consideration [Jarzabek and Hitz, 1998], and the use of precise specifications to determine and maintain the semantic dependencies between components [Perry, 1999] are proved to facilitate software evolution.

2.2.3 Elaboration Dimension

Elaboration dimension embraces any effort to understand and describe particular characteristics of software products or processes pertinent to software evolution. The characteristics include static aspects such as consistency, traceability, complexity and productivity, as well as dynamic aspects such as the impact of changes upon software structure or behaviour. Evolution metrics and change impact analysis are two considerable research trends that fall under this category. The former is concerned with the study and application of software measurements, for example, the use of design and implementation 'instability' metrics to track object-oriented software evolution [Li et al., 2000a], and the use of metrics for measuring the aspects of software for the process of reverse engineering [Yang et al., 1997]. The latter is concerned with the study of effect from software changes; for example, extracting evolution effect from source codes [Tahvildari et al., 1999], and using multi-graphs that represent software components for early evaluation of change impact [Deruelle et al., 1999].

2.2.4 Summary and Further Remarks on Software Evolution Approaches

As a summary, the views on software evolution discussed at the beginning of this section emphasize two important issues in software evolution: *the environment* that generates changes and *the process* that describes the way to perform changes. The former includes any source that causes a software system to evolve such as domain, experience, and organization. Based on the objective evidence in software maintenance, *business rule changes* are always considered as the most important sources to software evolution. The latter refers to the issues related to software process such as software technologies, method, tools, techniques, and languages. The solution to software evolution problems is likely to success by addressing both of these issues.

The three different dimensions of software evolution approaches can be further explained using a three-dimensional view graph as illustrated in Figure 2-2. In short, *process* dimension refers to a set of activities, procedures, rules, guidelines and tools to manage software changes as well as to develop a more evolvable software system. *Product* is concerned with the improvement of the evolvability of software artefacts such as the study of software architectures, distributed, object-oriented and component-

based software systems. *Elaboration* dimension includes means to understand the attributes and effects of requirement or software changes to software components. Since these three dimensions are not mutually exclusive, some approaches may be located at any intersection point in the three-dimensional graph.

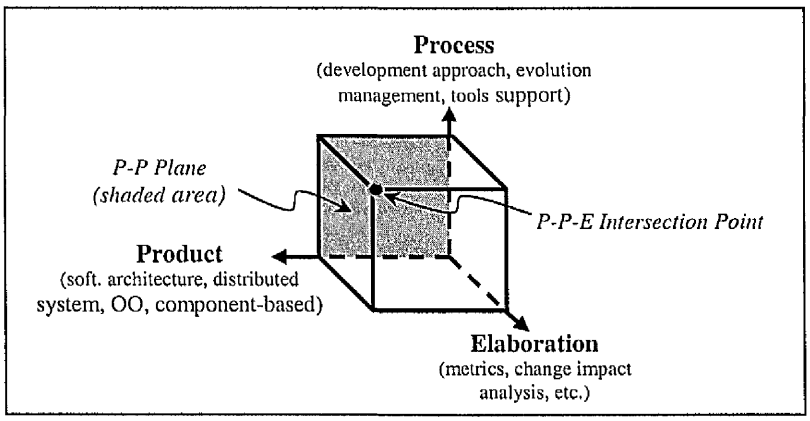

Figure 2-2 The three-dimensional view of software evolution approaches

A more careful consideration of the discussed software evolution approaches leads to other further remarks. Based on their success stories in real-world applications, it is found that software evolution approaches that address both product and process dimensions are more likely to success. In terms of the three-dimensional graph shown in Figure 2-2, the approaches must be at least located at product-process intersection (shown as *P-P Plane*) to ensure that they are practically accepted. In other words, all software evolution approaches should define the structure of the product and describe a fairly detailed process to produce and evolve the product. It is also found that very few approaches include all of the process, product, and elaboration dimensions. Therefore, such approaches (shown as *P-P-E Intersection Point*) are considered as the ideal approaches since they are desirable but very hard to achieve.

Although they end up with different degrees of success, most approaches make their own constructive contributions to the field of software evolution. However, these approaches have several common observable drawbacks that are listed as follows:

- The majority of them ignore the important aspect, or the most volatile part of rapidly changing business environment i.e. business rules. They tend to focus on technological issue such as software architectures, modelling languages, and software artefact management issues. As mentioned earlier in this section, business rules are the important sources to changes, and their changes may bring the highest impact to software system. Without considering the source of changes, these approaches only confine their solutions to the problem, and not to the sources of problem.

- Most of them concentrate on application model instead of the metamodel that may provide a higher flexibility to software system. By considering the evolution issues during the development of the metamodel, the components that facilitate evolution can be enforced to be included in the developed software model.

- Software evolution is often defined as the process of changes to the software system after delivery or successful operation, which implies the changes are only made after getting a feedback from the actual system deployment. This is not always true for rapidly changing business environment. In a highly dynamic business environment, the requirements are frequently changing even during development phases. Future approaches should be able to deal with rapid software evolution, in other words provide an application model that can be evolved prior to actual system deployment.

2.3 Business Rules

In the previous section, we have identified the importance of business rules in software evolution. By explicitly considering business rules in software development, we may reduce evolution effort. Business rule changes should only change those related volatile components, and they should not affect the core components of a software system. In this section, an overview of business rule approaches to software development will be examined.

The term business rule was introduced by Daniel S. Appleton. He defines business rule as *"an explicit statement of a constraint that exists within a business ontology"* [Appleton, 1984]. This definition emphasizes the importance to explicitly represent the constraints which are embedded in business ontology, i.e. a conceptual schema within a business domain that is often represented by a typical hierarchical data structure containing all the relevant entities, their relationships and rules. Without the explicit statements of the constraints, it is difficult to deal with their inconsistencies and changes within business ontology. Business rules, which explicitly represent the constraints, play an important role in managing business ontology.

To date, there are many business rule definitions introduced by various areas of research such as conceptual modelling, database, object-oriented paradigm, and business modelling. The most popular business rule definition, which is adopted by many current business rule approaches, is the definition proposed by the Business Rule Group (BRG). BRG defines a business rule as

> "*a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behaviour of the business*" [Hay and Healy, 2000].

In the above definition, the statement ranges from the definition of each stored data to the constraints on insertion, deletion, and modification of the data which specify the structure and behaviour of business operation. They are often imposed by business owners. For example, only validated purchase order will be accepted, and each purchase order is only can be approved by an authorized person. Apart from business owners, constraints may also be imposed by industry, economic, and political environments [Moriarty, 1993]. The business rule definition proposed by the BRG is found adequate to be adopted for the purpose of this research since it applies to both business and information system perspectives.

From the above definitions, business rules are not newly discovered software artefact; they already exist or are embedded in other software artefacts such as analysis models, design components, source codes, and data constraints. Recently, there is increasingly interest on separating business rules from other aspects of software development for a

number of reasons. The most common reason is to facilitate software evolution since business rules are the most volatile part in business information system. By explicitly model and implement business rules, the rule changes are localised only to volatile part without affecting the remaining stable part of software system.

Business rules also assist effective communication between users and software engineers since they are understandable by users as business process owners [Moriarty, 1993; Gottersdiener, 1997]. Business rules can also be considered as a source for requirement determination [Bajec and Krisper, 2001]. The intended users found that it is easy to express their requirements, which in turn lead to faster application development and better quality of requirements. They also act as a means to align software solution with business environment by propagating their changes to the associated software system [Morgan, 2002]. Other benefits of the explicit consideration of business rules include facilitate change, promote reuse, and improve software scalability. The effort to emphasize business rules in software development lead to a new approach to software development namely business rules approach which is the area of discussion in this section.

Business rules approaches vary in terms of the level of the roles of business rules incorporated in software development activities. At a lower level, there are very minimum differences from the current approaches with reference to business rules. They use the existing techniques and notations in object-oriented, software architectures, component-based, or databases. They may be only address the issue of mapping business rules to other software artefacts [Hruby, 1998], encapsulate business rules in an architectural component [Vives and Dombiak, 2000], implement business rules as the specific distributed object components [Rouvellou et al., 1999; Rouvellou et al., 2000], or provide business rules repository [Haggerty et al., 2001].

At a higher level, the approaches contribute more efforts to promote business rules as central constructs in software development such as the specific models for business rules modelling, and the specific software process for business rules [von Halle, 2001b]. In certain attempts, new notations, formalisms [Diaz et al., 1998], and framework [Kardasis, 2001] were introduced for business rules modelling. Although business rule approach provides a new way of thinking on how to effectively analyze, design and

build a software system, in practice it is too early to claim that it is a new software development paradigm. It still largely based on the previous paradigms such as object-oriented and information system engineering.

Based on the in-depth review of literatures, the discussion on business rules approaches can be organized into three sub-sections i.e. business rule approach to software development, business rule in object-oriented software development, and other areas of research on business rules. Since this research focus on a special category of software namely business information system, the terms software, information system (IS), and business IS are used interchangeably unless otherwise stated.

2.3.1 Business Rules Approach to Software Development

The importance of business rules in software development was recognised since 1980s. One of the prominent example is RUBRIC (Rule-based Representation in Information Construct) approach that separates and explicitly model business policy from operational procedure in business information systems [Layzell and Loucopoulos, 1988]. In general, RUBRIC considers two types of rules i.e. static constraints and dynamic rules. The former is used to augment E-R notation in the expression of constraints; it is written in **IF...THEN** format. The latter describes rules in terms of 'trigger', preconditions', and 'reference'. It is written in the following format: **WHEN** `<trigger>` **IF** `<preconditions>` **THEN** `<reference>`, which means the behaviour of an entity (`reference`) is executed on the occurrence of events (`trigger`), and the satisfaction of certain conditions (`preconditions`). RUBRIC also provides system architecture which consists of modules that capture, validate, implement, and maintain business rules [Loucopoulos and Layzell, 1989].

In the following decade, there was more awareness that business rules are the essential aspect of information systems. Moreover, the existing common software development methods were found "*insufficient or at least inconvenient*" in expressing business rules [Herbst et al., 1994]. Therefore, there were increasing interest in the method for elicitation, documentation, modelling, deployment, and evolution of business rules. Until now, they are still relevant as the important research and practice issues. Among the popular focuses of the current research in business rule approach are the role of

business rule in system development lifecycle, business rules as the link between business and IS, and business rule representation.

Business rules in system development lifecycle

With regard to the coverage of software development lifecycle, business rule approaches can be generally divided into two main groups: conceptual modelling and the complete lifecycle. The former is concerned with the capture and modelling of business rules during system analysis. The latter provides a more complete set of activities for business rule lifecycle which consist of elicitation, modelling, deployment, and evolution of business rules.

In conceptual modelling, Loucopoulos et al. introduce a conceptual model that consists of Conceptual Rule Language (CRL) and entity relationship time (ERT) [Loucopoulos et al., 1991]. CRL is used to facilitate the capturing and specification of business rules, which are classified into three main types: constraint, derivation, and action rules. The business rules, which are specified in CRL, makes reference to the ERT conceptual data model and mapped to the object-oriented representation expressed in PROBE language. The syntax for business rule expressions in CRL is described by its BNF definition.

For another example, Herbst proposes a set of steps in specification and validation of business rules together with the metamodel for specifying the rule repository [Herbst, 1996]. The steps include the development of specifications for process structure, processes, conceptual data model, and integrity constraints. In this approach, business rules are used to specify processes and constraints. Regarding the metamodel, it can be divided into core and environment elements. The core elements include event, condition, and action which form the rule expression. This type of rule is also known as ECA rule. The environment elements comprise the closely related components in information system such as business process, organizational unit, data model component, and actor.

The Business Rules Group (BRG) aims to formalize an approach for identifying and expressing business rules [Hay and Healy, 2000]. BRG proposed a metamodel that defines the conceptual model for business rule expression. In this metamodel, business rules are classified into structural assertion, action assertion, and derivation. Structural

assertion consists of terms and facts, which is the constraints on terms such the properties of entities, attributes and relationships. Action assertion constrains the behaviour of business activities such as data updating or operation invocation. Derivation derives a new fact based on the existing facts.

In information system analysis, Kardasis and Loucopoulos propose a process to facilitate the business rule elicitation and a repository schema to assist business rule management [Kardasis and Loucopoulos, 2003; Kardasis and Loucopoulos, 2004]. Their approach focuses on the modelling of business rules within the intentional and operational views of information system analysis. The intentional view is concerned with business context perspective whilst the operational view relates to business process perspective. The informal business rules in business context perspective, which are often expressed in natural language, are transformed to a more formal business rules within business process perspective.

Apart from the above approaches, there are also other efforts in conceptual modelling such as separating business policies, which can be broken on certain occasions, from business rules which can never be broken [Snoeck, 2002]. There are also reported work on the role of business rules as a complementary technique in other requirement engineering methods such as objective driven [Bubenko and Wangler, 1993], client oriented requirement baseline [Leite and Leonardi, 1998], workflow modelling [Liu and Ong, 1999], extreme requirements [Leonardi and Leite, 2002], and enterprise modelling [Skersys and Gudas, 2004].

For a broader coverage of business rule modelling in software development lifecycle, Ho et al. propose a framework based on 'data-rule-process' design procedure [Ho et al., 2003]. In this framework, the terms and facts are identified during data design. The rules, which are built based on the terms and facts, are determined during rule design. A set of Scenario Functions (SFs) is developed during process design. SF is a function with a number of elements that can be expressed in programming language. Business rules are implemented as an independent element in SF that can be maintained without affecting other elements.

Rosca et al. propose a more complete business rule lifecycle which comprises of acquisition, deployment, and evolution of business rules [Rosca et al., 2002]. Their approach is closely related to the techniques in decision support system. During acquisition, the goal-oriented rules which relate to enterprise objectives are captured using the method that exploits knowledge from the decision support systems. The goal-oriented rules are further refined into operational rules. The determinism, conflict and ambiguity of operational rules are addressed during deployment. During evolution, the data gathered by monitoring activities supply information about the required rule changes.

Business Rule Solutions (BRS) provide a comprehensive steps and guidelines for business rule approach using its Proteus™ methodology [Ross and Lam, 2003]. BRS also introduce sentence templates and automated tools for capturing and managing business rules [Ross and Lam, 2001]. This approach is also complemented by a comprehensive explanation on theoretical aspects of business rule using predicate logic [Ross, 2003]. Similar business rule approaches are also proposed by other information technology practitioners [Morgan, 2002; von Halle, 2002].

Business rules as the link between business and IS

In connection with linking business rules to IS, Hars and Marckewka introduce a CASE tool that utilizes natural language processing (NLP) for interpreting and mapping business rules to information systems design [Hars and Marchewka, 1996]. This tool allows users to express business rules in their natural language, which in turn allows them to provide complete and appropriate requirements without being restricted by models and methods associated with IS design. The tool is capable of identifying condition-action (IF-THEN) structures used in natural language. It is also capable of identifying the key concepts and categories that are involved in each action and condition such as person, location, and time. These business rules are used to generate a business process diagram in flowchart notations. The limitation of this tool is it only deal with a single word concept, and not applicable for word combination. Due to linear nature of natural language, it is also very hard to automatically build complex branching structure.

Bajec and Krisper advocates business rules to be used as information resources that help in establishing link between organization's business and its supporting IS [Bajec and Krisper, 2001]. A number of activities related to business rule modelling and implementation are included in both business modelling and IS development. Business rule repository, which is accessed by these activities, act as an explicit link between business and IS. The repository is independently managed by a set of business rule maintenance sub-activities. However, there is no detailed explanation on how to actually implement the link.

Groznik and Kovacic attempt to establish the relationships between business rules and other business related concepts in business modelling [Groznik and Kovacic, 2002]. Their motivation is to establish an environment in which business rules can be traced from their origin in business environment through to their implementation in IS. In their approach, the initial rule-based description of business process (in business modelling), which is written in natural languages, is broken down into detailed rules which act as specifications of IS requirements. These detailed rules, which are now in a more structured form, are consequently used to develop workflow model of IS design.

There are also suggestions to link business rules with UML use cases. Hruby outlines the link between business process and business object views at organizational, system, and architectural level using use case model [Hruby, 1998]. In a workshop on the styles of documenting business rules in use cases, many participants suggest that if the number and complexity of the rules is high, business rules should be located external to use case [Anderson et al., 1997]. The business rules can be referenced from use cases using pointers. In his thesis, Hurlbut proposes an adaptive use case, which is extended from UML use case, to interact with business rule statements in facilitating domain evolution [Hurlbut, 1998].

Business rule representation

Most of the above discussed approaches are supported by their underlying rule language for the purpose of business rule representation. The main objective of the rule language is often to provide an understandable business rule statements to both business and technical users with enough formalization for automated validation and implementation.

For this reason, there are different formality levels of the language. High level of formality is important to ensure the preciseness and completeness of rule expression for the modelling and implementation purpose. However, for the purpose of communication with the business users, natural language which is less formal is more suitable. With regard to the formal and structured representation of business rules, there are three main category of languages i.e. sentence templates, mathematical logic, and graphical representation.

Sentence template is the most popular form of representing business rules [von Halle, 2001a; Morgan, 2002; Ross, 2003; Wan Kadir and Loucopoulos, 2003; Skersys and Gudas, 2004]. It can be easily mapped to natural language to provide effective communication between business users and system developers. In terms of implementation, it is structured enough to be mapped to design or implementation components. For a more precise representation, these templates can be further defined using context-free grammar meta-language such as Extended Backus-Naur Form (EBNF) [Herbst, 1997; Kardasis, 2001].

A more formal approach in rule representation uses mathematical logic which mostly comes from the field of knowledge representation. Using logic, the formal representation of business rules can be automatically used in decision processes. External Rule Language (ERL) is an example of a formal logic language that describes business rules in terms of entity and relationship constraints and the details of information flow [McBrien et al., 1991]. Courteous Logic Program (CLP), which extends ordinary logic programs with prioritized conflict handling, is another example of the use of logic in business rule representation [Grosof et al., 1999]. CLP is encoded using XML to produce Business Rule Mark-up Language (BRML) which is used in the implementation of e-commerce applications. Defeasible logic offers more expressive power and lower computational complexity compared to CLP [Antoniou and Arief, 2002]. It consists of strict rules and defeasible rules. The former is different from the latter in that the conclusion of the former is always valid whenever its conditions are true whilst the conclusion of the latter can be cancelled by the existence of another rule with an opposing conclusion.

In a graphical representation of business rules, UML-A is proposed as an extension of UML notations and guidelines in explicit modelling of active rules (event-condition-action rules) during analysis and design [Berndtsson and Calestam, 2003]. A simple active rule is possible to be represented by UML statecharts using the concepts of events, event parameters, conditions, and actions. A rhomb notation is used to represent active rule that guards the transition between two states. For a more complex rule such as multiple rules concurrently triggered by the same event, a 'junction pseudo-state' state is introduced to split the transitions. Situation/Activation diagram is another example of the graphical representation of business rules [Lang and Obermair, 1997]. Situation diagram defines the situations that trigger business rules whilst Activation diagram specifies business rules according to event-condition-action structure. There is also Object Role Model (ORM) that provides a rich language for expressing business rules, either graphically or textually [Halpin, 1996]. ORM views the application domain as a set of objects, i.e. entities or values that play roles, i.e. parts in relationships. It provides different graphical notations that are attached to the roles to specify business rules such as pair-exclusion constraint, asymmetric constraint, uniqueness constraint, and entity subtype.

2.3.2 Business Rules in Object-Oriented Software Development

Object-oriented software development paradigm gains popularity in academia and industry since 1980s due to its various benefits. The concepts found in object-oriented paradigm such as object, class, attribute, operation, and relationship are closely naturally related to the concepts in the real world. Its modelling mechanisms, such as encapsulation of object details, inheritance, and polymorphism, are able to produce extendable, evolvable, and reusable software components. Other benefits of object-oriented paradigm include faster development, smooth transition from analysis to implementation, higher cohesion components, and lower coupling between components. Nowadays, object-oriented techniques are still incorporated in most contemporary software development approaches within various fields including distributed systems, business modelling, and adaptable software systems to name a few. A brief overview of software evolution in object-oriented software systems was presented in section 2.2.2.

Although object-oriented techniques bring many benefits towards a more maintainable software system, they are still focusing on the software issues. There are a lot of improvement opportunities if the source of changes, i.e. business rules, is taken into account. This leads to many attempts to incorporating business rules in object-oriented paradigm. It is observed that majority of the approaches focus on the design and implementation stages. They may be generally classified into three recognised fields: specification, architecture, and framework.

Some approaches only consider business rules as a specification which is linked or transformed to other modelling elements. For example, Eriksson and Penker suggest to use Object Constraint Language (OCL) for business rule specification, which is linked to a specific element in a business model [Eriksson and Penker, 2000]. OCL is a declarative specification language that provides a precise and easily understood specification for constraints. The specification is written in a curly bracket or note attached to model element. Business rule is categorised into three main types: constraint, derivation, and existence rules. Constraint rules specify the possible structural and behavioural aspects of objects or processes. They are further divided into structural, operational, and stimulus/response rules. Structural rules are often specified by the specification of classes and relationships in the class diagrams. If necessary, short OCL expression can be written in curly bracket and attached to the related model elements. Operational rules define the pre- and post-conditions that must hold before or after an operation is performed. The OCL expressions for operational rules are often modelled as a note attached to a class. Stimulus/response rules define the flow of business process. OCL cannot be used to specify stimulus/response rules since it is not a procedural language. Instead, they are presented using an activity diagram. Derivation rules define how to derive new information from other information. They are further divided into inference and computational rules. OCL invariants (conditions that must be true at all time) and post-conditions are used to specify derivation rules. Existence rules describe the time condition of the object life. OCL invariant or the existing features in class diagram, such as aggregation, can be used to specify this type of rules. During implementation, the above rules are coded in their implemented class.

Another example is an event script, which is used to identify and specify business rules [Poo, 1999]. The event script extends the use case modelling technique by adding additional structured specification which includes the business rule descriptions such as pre-event condition, post-event triggers, and derivative policies. These descriptions are consequently transformed into class specification which provides an action for each event.

There are also efforts that propose the use of business rules in their object-oriented architectures to improve the evolvability of the produced software. Among their popular examples are coordination contract [Andrade et al., 2002] and Adaptive Object Model (AOM) [Yoder and Johnson, 2002]. In coordination contract, business rules are implemented in an association class called 'contract' that intercept the method invocation between software components. If the invoked method is registered as a business rule event in a contract and the value of the rule conditions are true, then the contract triggers the listed actions. AOM includes several design patterns and rule objects in its meta-architecture. Design patterns simplify the core entities evolution whilst rule objects externalize business rules, thus localise business rule changes.

Another example of business rule in object-oriented architecture is the three-layer architecture that aims to reduce development and maintenance cost by simplifying the specification and reuse of business rules [Mohan et al., 2000]. It employs the well-known three-layer software architecture: presentation, application, and persistence layers. The application layer is further divided into domain and business rule layer. The business rule layer consists of three packages: rule, rule handler, and observer. During operation, an observer object detects any event object sent by an object in a domain package. It then passes the information from event object to a rule handler. A rule handler, which manages rule objects, triggers the necessary business rules. A rule exception object is thrown if the rule execution fails for any reason.

Business rules are also found as the components of object-oriented framework, which is an infrastructure for the design and implementation of object-oriented system. One of the prominent examples this type of framework is Business Rule Beans (BRBeans) [I.B.M., 2003]. In BRBeans, business rules are implemented and managed in a separate component developed using JavaBeans technology. Business rules are fired by rule

client component located in the software application. Another example of object-oriented framework is Aspect Bean [Cibrán et al., 2003]. Aspect Bean aims to address the problem of unsystematic separating business rules from the core application which causes the trigger points to be embedded and scattered within application. Consequently, it is very hard to locate these trigger points if any business rule change occurred. Therefore, Aspect Bean is developed to separate and encapsulate the linking code using aspect-oriented programming technique.

Other than the above areas, there are also different topics relevant to the role of business rules in object-oriented software systems. For instance, rule object design patterns [Arsanjani, 2000], object-role modelling for conceptual data modelling [Halpin, 2001], and the requirements for tools and environments to support business rules [Mens et al., 1998].

2.3.3 Other Areas of Research on Business Rules

In addition to the above mentioned software development approaches, there are also other notable areas of research on business rules. They are: the architecture of business rule system, business rules extraction, and in the field of database research.

With regard to the architecture for business rule system, Kang et al. propose an architecture that allows user to manage business rules using rule editor [Kang et al., 2004]. The rule editor interacts with rule repository via rule object model handler. For the purpose of implementation, inference engine and rule language in XML format are also included in this architecture. Rosca and Attilio propose an architecture that allows the sending and receiving among proprietary rule repository of heterogeneous business information systems [Rosca and D'Attilio, 2001]. In this architecture, business rule document in XML format is generated by rule generator of one system, which is consequently processed by rule parser of another system.

Business rules extraction also gained considerable interest given that many business rules are buried in the source codes of legacy software systems or other software artefacts. The study of business rule extraction from legacy systems is closely related to the study of software reverse engineering. For example, Shao and Pound combine data

and program understanding techniques to discover business rules [Shao and Pound, 1999]. They developed automated tools to produce generic representation of programs and databases: parsing tools are used to convert program into an abstract syntax tree representation whilst schema tools are used to derive conceptual data models from databases. Then, business rules are extracted from these generic representations. Wang et al. propose a framework for extraction of business rules from large legacy systems [Wang et al., 2004]. This framework consists of five steps: program slicing, domain variable identification, data analysis, business rules presentation, and business validation. Ramsey and Alpigini propose a mathematically based framework to extract business rules from heterogeneous programming language using Wide Spectrum Language [Ramsey and Alpigini, 2002]. Other than business rules extraction from legacy systems, there is also a technique proposed to extract business rules from structured analysis specifications such as data flow diagram, data dictionary, structured English descriptions and general system descriptions [Leite and Leonardi, 1998]. There is also an attempt to help the understanding of the extracted rules such as multi-format presentation framework that consists of presentation and representation layers [Fu et al., 2001].

In the database field, the Conceptual Rule Language (CRL) is introduced to model constraint, derivation, and event-action rules, and consequently map to deductive database [Petrounias and Loucopoulos, 1994]. Attribute-oriented business requirements and constraints is proposed to bridge a gap between requirements elicitation and database implementation [Khan et al., 2004]. This approach extends the existing entity relationship model with a new construct that would express business rules along with the conceptual model. Object Constraint Language (OCL) is used to express constraint in the conceptual database model [Demuth and Hussmann, 1999]. In this approach, business rules are encoded as a database schema, and not in an application program. Minor modification to OCL is suggested to enable the transformation of its specification into SQL. There is also suggestion to develop and implement business rules similar to the development and implementation of database i.e. as meta-data [Perkins, 2002].

2.4 Business Rule Conceptual Modelling and Evolvable Software Systems

In the previous sections, the extensive reviews on state-of-the-art business rule approaches suggested that the approaches can be divided into two categories with regard to the objectives of this research: business rule conceptual modelling and construction of evolvable software systems. The former focuses on capturing and representing business rules but they lack detail in the information about linking or transforming business rules to software systems. On the other hand, the latter places emphasis on providing the details on developing a rule-centric evolvable software system but the business rules are poorly specified and their software components are not clearly linked to the conceptual level of business rules. In investigating the finest way of incorporating business rules, which are originated from business domain, with the development of an evolvable software system, both business rule conceptual modelling and evolvable software systems must be systematically reviewed. In this section, a number of prominent approaches were selected to be reviewed based on the availability of literatures and examples. The MediNET case study is occasionally used to provide business rule examples in describing the selected approaches.

2.4.1 Business Rule Conceptual Modelling

Although many approaches include business rules in their modelling constructs, most of them are coincidence, implicit, informal, and lack of details. There are very few approaches that directly provide a set of modelling abstractions, such as a metamodel, and process guidelines to externally systematically model the business rules. In this section, we will discuss three approaches that attempt to formally model business rules at the conceptual modelling or system analysis stage of software development i.e. Business Rule Group (BRG), Business Rule Oriented Conceptual Modelling (BROCOM), and Business Rule Solutions (BRS) approaches.

2.4.1.1 Business Rule Group (BRG)

The Business Rules Group (BRG), formerly known as the GUIDE Business Rule Project, investigated an appropriate formalization for the analysis and expression of business rules [Hay and Healy, 2000]. According to BRG, business rules are often

neglected, implicitly modelled, and informally expressed during system analysis. They are not adequately articulated until the time to translate the constraints into program code. Therefore, BRG proposes a solution to this problem by defining a conceptual model that completely described business rules as well as their formal expression for a smooth translation into implementation constructs.

As mentioned in section 2.3, BRG defines business rule as

> "...a statement that defines or constraints some aspect of the business. It is intended to assert business structure or to control or influence the behaviour of the business" [Hay and Healy, 2000].

Although the above definition can also be viewed from a business perspective, BRG choose to limit the scope of their work to information system perspective. In this perspective, business rule is considered as an explicit expression, either graphical or textual, which represent the constraints on the structure (i.e. entities, attributes, and relationships) of the stored and manipulated data, as well as the control on the operations (i.e. creation, updating, and removal) that manipulate the data. The expression must be declarative in nature, which specifies what is suggested, required, or prohibited during the operation of an information system. It may describe 'what' must or must not happen under certain conditions but it does not describe 'how' the thing happens in a procedural way.

There are two main parts of the BRG's business rule metamodel i.e. the origins of business rules and business rule types. The metamodel is shown in Figure 2-3. The origins of business rules help system analyst to identify business rules in an information system. It starts with a reviewing of a general statement of business policy. Each general policy may be composed of detailed policies. A policy may be the basis for one or more business rule statements. A business rule statement is a declarative statement of structure or constraint, and it is a source of one or more atomic business rules. Each atomic business rule is expressed in one or more formal rule statements. A formal rule statement is an expression in a specific formal grammar, which is specified by a formal expression type such as structured English. The examples of policy, business rule statements, business rule, and formal rule statement that found in MediNET case study is shown in Table 2-1.

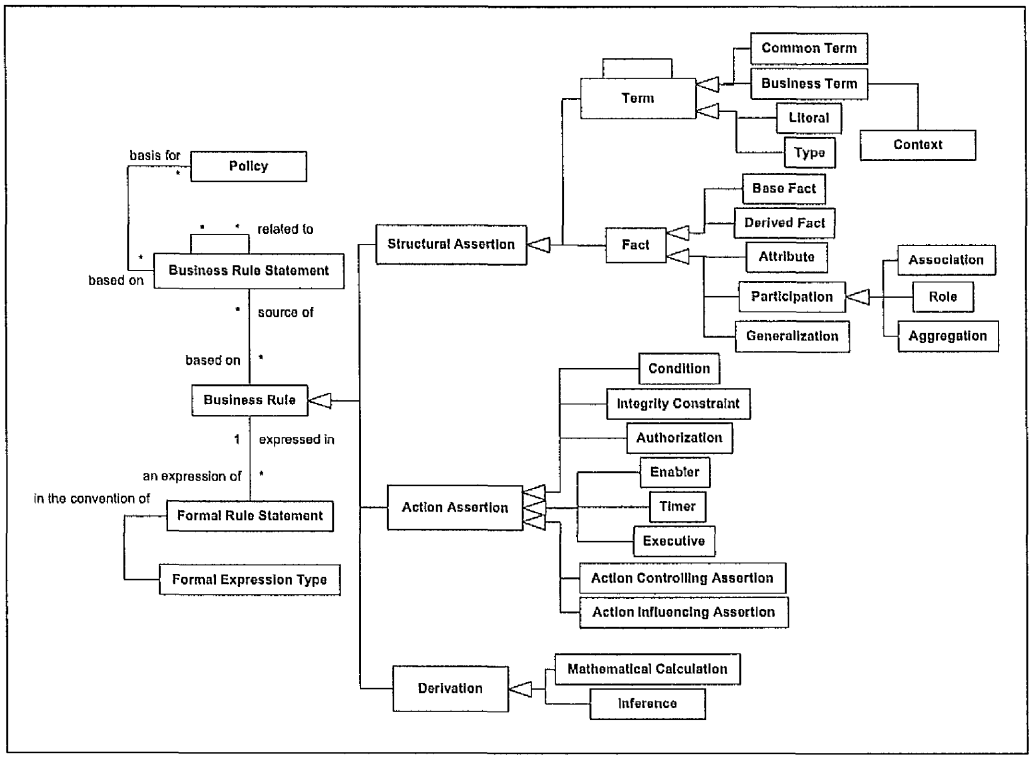

Figure 2-3 BRG's business rule metamodel [Hay and Healy, 2000]

As can be observed in Figure 2-3, business rules are classified into three main types: structural assertions, action assertions, and derivations.

Structural assertion is a statement about concept or relationship of something of importance to the business. There are two kinds of structural assertions i.e. terms and facts. A term is a word or phrase, which has a specific meaning to business. It is either a business term, which has a specific meaning to a business, or common term, which is part of a basic vocabulary. A term can also be categorized into type, which is an abstraction about a group of instances, or literal term, which is an actual value or instance. A fact may be either a base fact, i.e. the originally gathered information, or a derived fact, which is constructed from other assertions. It may also be classified into attribute, participation, and generalization fact.

Action assertion specifies constraints on the results that actions can produce. It consist of two components i.e. anchor object and correspondent object. An anchor object may be any kind of business rule whilst correspondent object may be either another business

rule or action. Often, action assertion is expressed in the form of **`if (anchor object) then (business rule | assertion)`**.

Action assertion may be classified in three ways. First, it can be classified into condition, integrity constraint, and authorization. Condition rule is the basis for executing another business rule based on the truth value of the specified condition, integrity constraint is the expression that must always true at any time, and authorization is a definition of a privilege or permission. Second, action assertion can be classified in terms of its role in controlling the execution of business process into enabler, timer, and executive. Finally, it can be classified to either mandatory (action controlling assertion) or optional (action influencing assertion).

Finally, a derivation is a derived fact that is created by an inference or a mathematical calculation from terms, facts, other derivations, or action assertions. A mathematical calculation derives fact using a specified mathematical algorithm whilst an inference produces a derived fact using logical induction or deduction.

Table 2-1 The examples of the MediNET business rules and origins in BRG

Policy	MediNET must provide a different package to a different type of patients.
Business Rule Statements	Cash patients should pay their bill amount in full. Their employer (panel company) will pay the cost of panel patients' treatment. Partly sponsored panel patients should pay their bills if it exceeds certain limit.
Business Rule	If the bill amount of the partly sponsored patient is more than the limit set by his panel company, the patient should pay the balance.

Formal Rule Statement	IF (bill.amount>limit) AND (patient is a partly panel patient) THEN
	bill.balance = bill.amount – limit
	invoke bill.payment(bill.balance)
	END IF

2.4.1.2 Business Rule-Oriented Conceptual Modelling (BROCOM)

BROCOM is an approach to system analysis that emphasizes business rules in conceptual modelling [Herbst, 1996; Herbst, 1997]. It attempts to solve a problem in the existing system analysis methodologies, which are found insufficient to completely and systematically model the business rules. The detailed comparison of the capability of these methodologies in business rules modelling are discussed in [Herbst et al., 1994].

For this purpose, BROCOM provides a metamodel that formalizes business rules in conceptual modelling, outlines modelling steps that systematically guide the process specification using business rules, and develops repository called BURRO that allows the administration of business rules as metadata.

BROCOM defines business rules as

> ".... statements about how the business is done, i.e., about guidelines and restrictions with respect to states and processes in an organization" [Herbst, 1997].

In his active database research, Herbst expands the scope of business rule definition to all kind of business processes, and not only restricted to data integrity constraints as traditionally defined in the area of database systems. The words 'states' and 'process' implies that business rules are associated to integrity constraints and business processes. Business processes are the dynamic properties of an organization. Integrity constraints are defined as constraints on the possible database states to ensure the database is always in the correct states in the context of the application. In its business rules classification, BROCOM considers integrity constraints as a special type of business rules.

Business rules often consist of three components namely *event* that triggers business rules, *condition* that should be satisfied before an action, and *action* that describes the task to be done. Based on this fact, business rules are specified in ECA structure or ECAA for a less redundant specification. The specification template for ECAA structure and the examples of business rules for the main processes in MediNET case study are given below.

BUSINESS RULE [n] **'BUSINESS-RULE-NAME'**

```
ON     (event)
IF     (condition)
THEN   action
```

'PATIENT-REGISTRATION'

```
ON     (presence of person) OR (request for consultation
        registration)
IF     (person not yet registered)
THEN   begin patient registration;
       raise event 'REGISTRATION-COMPLETE'
ELSE   begin consultation registration;
       raise event 'CONSULTATION-REG-COMPLETE'
```

'BILLING'

```
ON     (consultation completed)
THEN   create new bill;
```

```
            bill items := prescription items;
            calculate bill amount;
            print bill;
            raise event 'BILL-ISSUED'
'INVOICING'
      ON     (bill issued)
      IF    (invoice is not yet created) AND
            (the bill belongs to the panel company's staff)
      THEN   create invoice;
             insert bills;
             print invoice;
             raise event 'INVOICE-COMPLETED'
```

The large number of business rules in a real information system increases the complexity of rule and process specifications. To reduce such complexity, a top-down approach is chosen which provides different level of abstractions in the specification of rules and processes. Initially, the system is decomposed into a few numbers of main processes similar to structural decomposition in traditional software development methods. Next, business rules are used to specify each process. Each process is then refined until it is detailed enough in describing all processes in the system. In the above business rule examples, they are used to specify the higher level processes of MediNET. Each process can be further refined into lower level processes, for example, the INVOICING process may be refined into 'Invoice Item Pre-Processing', 'Invoice Creation and Item Insertion', and 'Invoice Closing'. The business rule specifications for the refined processes are given below:

BUSINESS RULE [3-1] **'PRE-PROCESS-INVOICE-ITEM'**

```
      ON     (bill issued)
      IF     (bill belongs to a panel patient)
      THEN   store bill in a list of panel bills;
             raise event 'INVOICE-ITEM-TO-BE-PROCESS'
```

BUSINESS RULE [3-2] **'CREATE-AND-INSERT-INVOICE'**

```
      ON     (invoice item to process)
      IF     (invoice is not yet created) AND
             (the bill belongs to the panel company's staff)
      THEN   create a monthly invoice for the panel company;
             insert the bill as invoice item;
             raise event 'INVOICE-CREATED'
      ELSE   open the invoice;
             insert the bill as invoice item;
```

BUSINESS RULE [3-3] **'CLOSE-INVOICE'**

```
      ON     (the end of month) AND (invoice created)
      IF     (invoice created)
      THEN   close invoice;
             print invoice;
             send to the panel company;
             raise event 'INVOICE-COMPLETED'
```

In the above examples, we have shown some business rule specifications that are associated to certain business processes. For a development of specifications on integrity constraints, a conceptual data model is used instead of process hierarchy. In the above discussion, we also described the business rule components i.e. event, condition and action. These components, together with business process and conceptual data model components are considered as the core of BROCOM metamodel, which is shown in Figure 2-4.

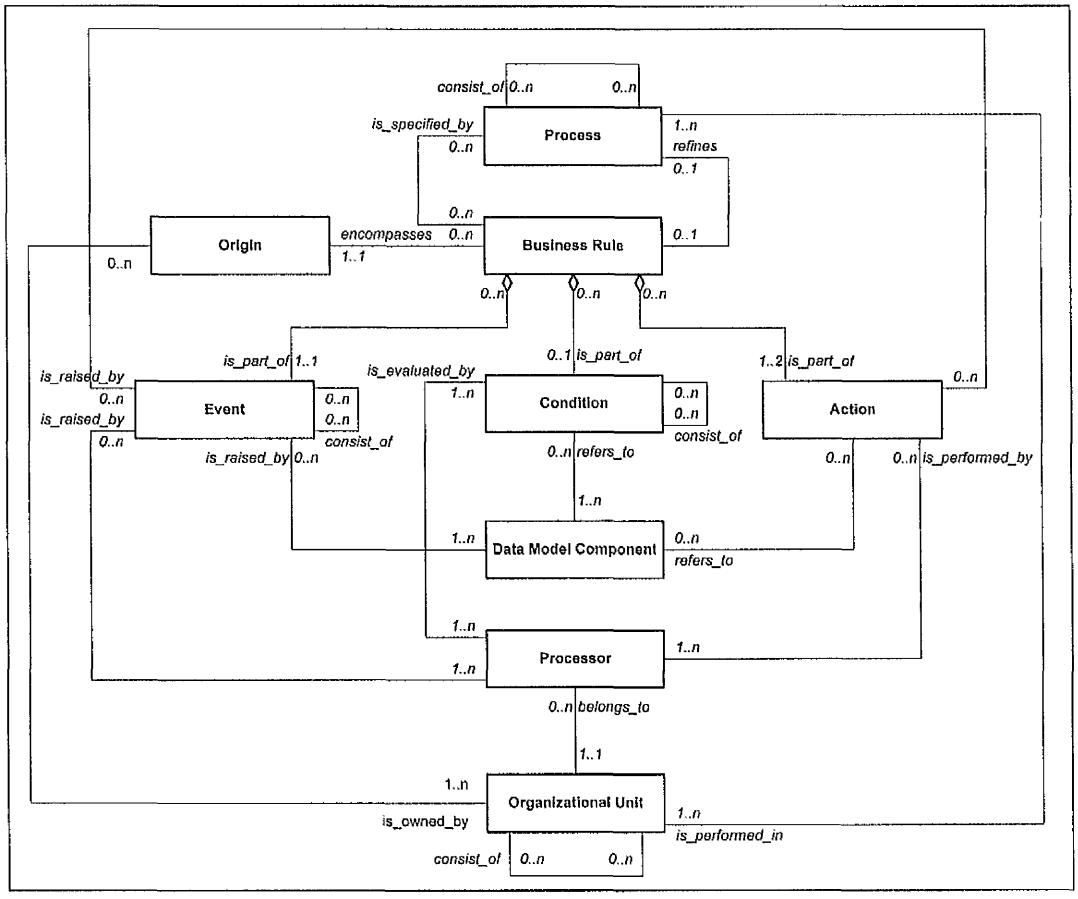

Figure 2-4 BROCOM metamodel [Herbst, 1997]

As shown in Figure 2-4, BROCOM metamodel also includes organizational facts such as origin, processor, and organizational unit. Business rules may be originate from internal origins, which results from internal organizational decisions, or external origins such as natural facts and legal norms. Processor may be manual processor i.e. human actors such as a clinic assistant, or automated processor i.e. software or hardware components.

Processor executes business rules: it detects the occurrence of events, evaluates conditions, and performs actions. Organizational unit is considered in the metamodel in order to provide a comprehensive model of the universe of discourse. Since it owns origin and contains processors, organizational unit helps system analyst to identify origins and processors, which in turn support business rules gathering.

In short, BROCOM is able to deal with formalization, complexity, traceability, and completeness in business rule-oriented conceptual modelling. The case study in business rule implementation found that all rules are adequately formalized in the ECAA structure [Herbst, 1997]. The complexity is reduced by rules classification whilst traceability is achieved via links to organizational facts. By considering the organizational environment in the metamodel, BROCOM achieves the completeness in capturing information about the universe of discourse.

2.4.1.3 BRS Approach

BRS approach is a business rule methodology that assists users in capturing and managing business rules (BRS stands for Business Rule Solutions, LLC). It is supported by a strong theoretical foundation, sentence templates, and its own development process. The metamodel and tool are also provided to facilitate users in managing business rules in their organization. BRS is already applied in a large number of information technology projects, and it is the best example of the combination between theory and practice in business rule methodology.

In BRS, business rule is defined as "*a directive intended to influence or guide business behaviour*" [Ross, 2003]. BRS emphasises that business rule should be regarded as an element of the business system, and not the component of software system, based on the fact that it is not necessarily implemented as software component. Instead, it may be implemented as manual process or using business logic technology such as rule engines, decision management platforms, and business logic servers. Another point highlighted by BRS with regard to business rule definition is the separation between the 'know' and the 'flow'. The 'know' part of business process (i.e. the 'flow') should exist and be managed as a resource in its own right.

Business rules are classified in terms of how they react to events into three fundamental categories: rejectors, producers, and projectors. Rejector rule disallows (or rejects) an event on a violation of the rule to prevent business from incorrect data or state. Producer rule derives a value based on some mathematical functions. It is further divided into computation and derivation rules; the former computes a value using standard arithmetic operations whilst the latter derives a truth value (true or false) based on logical operations.

Projector rule performs the specified action on the occurrence of a relevant event. It may be further divided into enabler, copier and executive rules. Enabler rule asserts that either something is applicable or not based on certain circumstances. It determines the truth value of a fact, the exception in firing of another rule, the creation or deletion of a data item, or the permitted execution of an operation, process, or procedure. Copier rule sets the data item with actual values from something that persists or defines the parameter related to how data is to be presented. Executive rule triggers an operation, process, or procedure to execute, or a rule to fire.

BRS provides sentence templates, which is called BRS RuleSpeak, as the guidelines to develop rule statements. Sentence template is a basic structure or pattern that can be used to express rule in consistent, well-organized manner. The main purpose of using templates is to improve the communication at business level. The use of templates produces easily understood business rule statements with consistent meaning. Sentence template is not a technical or formal language; BRS provide the formal definition of business rules using predicate calculus. Although they are meant for business domain, business rule statements written using sentence templates can be considered as a basis for the implementation level rule structure. The BRS templates organized into their type are presented in Table 2-2.

Regarding the development process, BRS introduces a business rule methodology called BRS Proteus™ Methodology that defines a number of steps for both business and system modelling [Ross and Lam, 2003]. Regarding business modelling, the process starts with determining the scope of the information system. It follows by the development of business tactics, workflows, terms and facts. Next, the business rule specifications are developed based on the information from workflows, terms, and facts.

Business rules can also be captured from the existing documented rules such as procedure manuals, training materials, and existing system documentation. The business product, service, and process are consequently analyzed to identify the decision points. Finally, the existing business rules are further formulated and analyzed for consistency, reducing complexity, validation, and detailing the rule properties.

Table 2-2 BRS business rule templates and examples

Category	Templates	Example
Rejector	<Subject> MUST/should [not] <fact> [if/while <condition>]	Each patient MUST have a patient registration number.
	<Subject> may/should <fact> ONLY if/while <condition>	A patient may register for consultation ONLY if the patient is a registered patient.
Computation	<Subject> must/should [not] BE COMPUTED as <mathematical formula> [if/while <condition>]	The amount of a bill must BE COMPUTED as the total amount of all bill items.
Derivation	<Subject> must/should [not] BE TAKEN TO MEAN <logical expression> [if/while <condition>]	A category one past due paymaster must BE TAKEN TO MEAN a paymaster with more than three months unpaid invoices.
Inference	<Subject> must/should [not] BE CONSIDERED as [a] <term> if/while <condition>	A paymaster must BE CONSIDERED as preferable if its arrears in less than RM 3,000.00 and the total transaction in more than RM 10,000.00 for the last four months.
Rule toggle	<Rule name> must/should [not] BE ENFORCED if/while <condition>	Each patient must be registered prior to consultation UNLESS the patient is an emergency patient.
Process toggle	<Process/procedure> must/should [not] BE ENABLED/DISABLED if/while <condition>	Insert-patient-in-consultation-queue should BE DISABLED if the patient is in blocked status.
Data toggle	<Data> must/should [not] BE CREATED/DELETED if/while <condition>	Each patient record must BE DELETED if the patient is inactive for more than 10 years.
Imprint toggle	<Term> must/should [not] BE SET to <term/value> [when/if <condition>]	The bill status must BE SET to 'unpaid' when it is created.
Presentation rule	<Subject> must/should [not] BE DISPLAYED [to/in/on <media>] <display manner> [if/while <condition>]	An invoice must BE DISPLAYED with the detailed patient information if the paymaster fully pays the patient bill.
Process trigger	<Process/procedure> must/should BE EXECUTED when <condition>	Send-invoice-reminder must BE EXECUTED when the invoice's payment is not received after 30 days from the invoice issue date.
Rule trigger	<Rule name> must/should BE FIRED when <condition>	The print-format-rule must be FIRED when the invoice is printed or displayed.

With reference to system modelling, the first step is the mapping of business rules resulted from business modelling to their respective system components such as devices, hardware/software platforms, and human roles. The next step is the reviewing

of system interfaces to support the workflows and fact models. This step also include the identification of other kind of infrastructure besides IT systems to support business rules. The design of human interfaces, logical data model, and data migration are designed are performed as the following steps. Next, the original business rule statements are rewrite to follow the system naming and formatting standards. Subsequently, the rule is reviewed to ensure the correctness and preciseness in terms of logic flows, timing, exceptions, and triggers. Then, the automated actions are decided by reviewing logic flows to ensure computability and user friendliness, and business rule script is developed to provide executable business logic. Finally, the rules are tested using the scenario test data specific to their type.

In respect of business rule management, BRS introduce metamodel that relates rule to other business components. The metamodel consists of more than thirty rule management elements which relate business rules to its origin and environment such as sources, business components, and software components. The examples of the rule sources are law, regulation, business policy, and procedures manual. Regarding business components, business rules can be linked to business tasks such as operational, decision-making, or creative tasks, and business purpose such as objectives, tactics, and roles. In terms of software components, business rules can be linked to artefacts such as design and implementation components. In the case that a business rule is not automated, the business rule is linked to manual implementation components. Apart from the metamodel, BRS also provides an automated tool for recording and organizing business rules called BRS RuleTrack™.

2.4.2 Business Rules and Evolvable Software Systems

In contrast to the approaches discussed in the previous section that focus on the externalization of business rules at analysis stage, the approaches in the development of evolvable software systems focus on design and implementation stage. The majority of approaches in this category aim to improve the understanding and evolution of a software system by logically and physically separate business rule components from other software components. Most of them also utilize or extend the object-oriented techniques due to the inherent flexible nature of object-oriented concepts. In this section, three leading approaches in the design and implementation of evolvable

software systems that include the benefits of business rules and object-oriented concepts are reviewed. They are Adaptive Object Model (AOM), coordination contract, and Business Rule Beans (BRBeans).

2.4.2.1 Adaptive Object Model (AOM)

Adaptive Object Model (AOM), which is also known as Dynamic Object Model [Riehle et al., 2000], is *"a system that represents classes, attributes, and relationships as metadata"* [Yoder et al., 2001b]. Unlike traditional object-oriented design, AOM is based on objects rather than classes. It provides descriptions (metadata) of objects that exist in the system. In other words, AOM provides a meta-architecture that allows users to manipulate the concrete architectural components of the model such as business objects and business rules. These components are stored as an object model in a database instead of in code. The code is only used to interpret the stored objects. Thus, user only needs to change the metadata instead changing the code to reflect domain changes. AOM is active because system will change immediately after changing metadata.

AOM is a combination of several design patterns. *TypeObject* and *Property* patterns are used in the core of AOM. *TypeObject* provides a way to dynamically define new business entities. In a traditional object-oriented design, business entities are represented by a set of classes or objects. The changes to these classes require code changes since all classes are implemented in codes. Differently, *TypeObject* provides higher abstraction notions by using 'Entity' and 'Entity Type' to represent business entities. Users may extend the model by manipulating instances of Entity and Entity Type, which are stored in a database. While *TypeObject* provides a flexibility to manipulate objects and classes, *Property* pattern provides flexibility to define object attributes. It allows objects of different types to implement attributes differently by creating an instance variable that hold a collection of attributes.

The above *TypeObject* and *Property* patterns can be used together with the existing object-oriented syntax and semantics to define the simple business rules such as types of the entities or attributes, permitted subtypes, permitted type of relationships, cardinality, and attribute's optionality. However, *Strategies* [Gamma et al., 1995] and *RuleObjects*

[Arsanjani, 2000] are used to model complex rules which are *'functional or procedural in nature'* [Yoder and Johnson, 2002] such as permitted types of values for certain attribute, or permitted relationships based on certain conditions. Strategy pattern define a set of algorithm with different interfaces. Strategies implement operations which are called by operations of the fundamental entities. RuleObject, which is a combination of two or more primitive rules (strategies) using *Composite* pattern, is used to represent more complex business rule. In RuleObject, the primitive rules that represent predicates, numerical values, and sets are combined using their respective connectors. *Micro-Workflow* architecture is used to define business rules that describe workflow. This architectural pattern describes the workflow structures including repetition, conditional, sequential, forking, and primitive rules. The combination of *TypeObject*, *Property*, and *RuleObject* design patterns forms the architecture of the AOM, which is shown Figure 2-5.

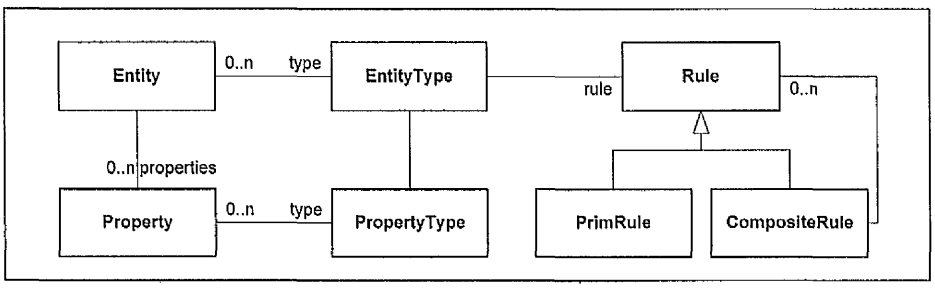

Figure 2-5 The Adaptive Object Model [Yoder et al., 2001b]

With regard to software development process for AOM, the evolutionary process is recommended due to the difficulty in finding the flexible parts of the system during its initial development phase [Yoder et al., 2001a]. In this process, the initial framework is developed and tested with the help of the business users. The feedback is used to identify the required flexibility and consequently evolve the framework. These activities are repeated until the stable framework is achieved. During maintenance, the changes to object model can be easily made since the system is stored as a meta-model.

Clearly, AOM makes a system more flexible and simple. It is flexible since users are allowed to configure and extend a system by manipulating the objects that are stored in database. The low number of classes makes AOM simpler than the traditional object-

oriented design. However, AOM also has several disadvantages [Yoder and Johnson, 2002] such as hard to understand and poor performance execution. The higher level of abstraction (meta-level) leads to difficulty in understanding AOM models. Consequently, the maintenance of such models is harder especially to a team member who does not participate in the development of the models. In terms of performance, the needs to interpret the large number of stored objects reduce the overall speed of system execution.

2.4.2.2 Coordination Contract

Coordination contract aims to separate coordination from computation aspects (or core components) of a software system [Andrade et al., 2002]. It is motivated by the fact that there should be two different kinds of entities in a rapidly changing business environment: core business entities which are relatively stable and volatile business products which keep changing for the business to remain competitive [Andrade and Fiadeiro, 2000]. For example, in the MediNET case study, *Patient*, *Panel Company* and *Bill* are the core business entities whilst *Cash Patient* and *Panel Patient* are the examples of business products. It is desirable to have a separate modelling and implementation of these two entities so that the changes are localized only to the volatile parts, with minimum impact on the core services already implemented in the system.

In coordination contract, volatile business products are implemented as contracts. Contract aims to externalize the interactions between objects (core entities) by explicitly define them in the conceptual model. It extends the concept of association class by adding a coordination role similar to other components in architecture-based software evolution such as architectural connectors [Oreizy et al., 1998], glue [Schneider, 1999] , actor [Astley and Agha, 1998] or change absorbers [Evans and Dickman, 1999]. The interacting objects never aware of contract. The invocation of the relevant method in the registered component triggers the business rule in the coordination contract. Figure 2-6 shows the coordination contract in the architecture of the separated configuration and component layers.

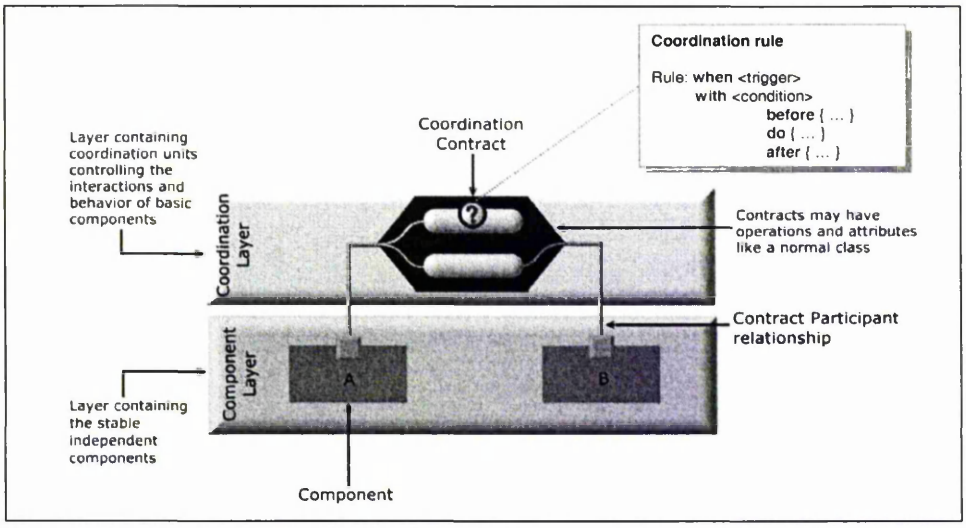

Figure 2-6 The architecture of coordination contract approach.

By separating computation from coordination, the coupling between components is minimize, which in turn localize the business changes [Andrade and Fiadeiro, 2001; Andrade et al., 2002]. If there is any change to business product or its business rules, the relevant contract is '*superposed*' on the existing implementation of core entities, which are considered as black boxes. The template for contract specification in Oblog language is given below:

```
contract <name>
    partners <list-of-partners>
    invariant <the relation between the partners>
    constants
    attributes
    operations
    coordination <interactions-with-partners>
    behaviour < behaviour being superposed>
end contract
```

The instances of contracts coordinate the instances of partners listed in **partners** section that satisfy the relationships specified in **invariant** section. Business rules are specified as interactions under **coordination** section in the following form:

```
<name> : when <condition>
         do <set of actions>
         with <condition>
```

The name of interaction is used to establish the overall coordination among the various interactions and contract own actions. The above interaction is quite similar to ECA

form as discussed in section 2.4.1.2. The condition specified by **when** section triggers interaction; it is typically occurrences of actions in the partners. The **do** section specifies the actions to be performed; they usually exist in the form of the partners' actions or some of the contract's own actions. Finally, the **with** section defines constraints on the action involved in the interaction, typically condition that should be satisfied before the action.

As an example, consider the billing process in the MediNET case study. After the bill is issued to a patient, an action should be taken depending on whether the patient is a cash or panel patient. If he is a panel patient, the bill will be transferred to a temporary list to be verified and subsequently inserted into invoice as an invoice item. Otherwise, he must pay the bill himself. The flexibility in dealing with this kind of billing process is accomplished by implementing the interactions between core entities, i.e. Patient and Bill, in different contracts namely `Cash-patient` and `Panel-patient`. If there is a change in software requirements, for example 'each bill belongs to panel patient should be inserted to invoice', the change can be done only at `Panel-patient` contract without effecting the core entities. The shortened version of the specifications for `Cash-patient` and `Panel-patient` contracts is given below.

```
contract Cash-patient
    partners b : Bill; p : Patient;
    invariant ?owns(b,p)=TRUE;
    coordination
        cashBill :  when b.issued()=TRUE
                    do b.payment()
end contract

contract Panel-patient
    partners b : Bill; p : Patient;
    invariant ?owns(b,p)=TRUE;
    coordination
        panelBill : when b.issued()=TRUE
                    do b.insertToTempInvoiceItem()
end contract
```

Assume that there is a panel company whom only wish to pay its staff's bill up to a certain limit, and the staff has to pay the balance. This requires the HCP to introduce a new package. Again, the package is implemented as a new contract to be added to the system without affecting existing core objects. The contract is specified below.

```
contract Partly-sponsored-panel-patient
    partners b : Bill; p : Patient;
    attributes Limit : Integer;
```

```
    invariant ?owns(b,p)=TRUE;
    coordination
        partlyPBill : when b.issued()=TRUE
                      do
                      {  Bill tb = new tb(b);
                         if (tb.amount <= Limit)
                         {   tb.amount = b.amount;
                             b.amount = 0;
                         }else
                         {   tb.amount = Limit;
                             b.amount = b.amount - Limit;
                         }
                         tb.insertToTempInvoiceItem()
                         payment
                      }
        payment :     do
                      {  b.amount = paymentAmount;
                         b.status = 'paid';
                      }
                      with b.amount > Limit
end contract
```

The concept of coordination contract is implemented as a design pattern [Andrade and Fiadeiro, 2000; Gouveia et al., 2001] which is based on other well-known patterns, namely the Broker and Proxy [Gamma et al., 1995]. This pattern exploits some widely available properties of object-oriented programming languages such as polymorphism and sub-typing. The semantics of contracts is defined by a formal program design language and model called CommUnity. In short, there are three modelling languages for coordination contract: CommUnity, design patterns, and structured specification.

As regards software development process, coordination contract extends the component-based development with a number of steps: context setup, contract development, testing, and deployment. Coordination Development Tool (CDT) is developed to support these activities [Gouveia et al., 2001]. CDT allows users to develop a coordination layer on top the implemented application components. The application components are registered as candidates for coordination. Contracts are defined to connect the registered components. Coordination rules and constraints are defined on those contracts using a mixture of abstract specifications and programming language syntax. The source code necessary to implement the coordinated components and the contract semantics in the final system are produced by generating the necessary parts according to the contract micro-architecture. Finally, the animation tool is used to check the behaviour of the contracts, prior to building an application. The animation

tool uses UML sequence diagrams to demonstrate the run-time behaviour of the component interactions via contracts.

2.4.2.3 Business Rule Beans Framework

Business Rule Beans (BRBeans), formerly known as Accessible Business Rules [Rouvellou et al., 1999; Rouvellou et al., 2000], is a framework that provides guidelines and infrastructures for the externalization of business rules in a distributed business application [IBM, 2003]. Business rules are externally developed, implemented and managed to minimise the impact of their changes on other components such as core business, application, and user interface objects. They are implemented as server objects, which are fired by embedded trigger points in application objects. The rule management facility is provided to help users to understand the existing rules and to locate the rules when changes are required. BRBeans is implemented as a part of WebSphere Application Server by IBM *"to support business applications that externalize their business rules"* [Kovari et al., 2003].

In this work, business rule is defined as "*a statement that defines or constraints some aspect of a business by asserting control over some behaviour of that business*" [Kovari et al., 2003]. It can be originated from within the company or from outside, typically from regulatory agencies. In general, business rules are categorised into two types: base rules and classifier rules [I.B.M., 2003]. Base rules, which are the most common rules, can be further divided into derivation, constraint, invariant, and script. Derivation rules define an algorithm to calculate and return the value. Constraint rules ensures that an operation follow the structural and behavioural constraints. Invariant rules make sure that changes made by an operation are properly related to one another. Script defines a variable portion of business process or workflow. Classifier rules derive a classification of business entity for a particular business situation. For instance, a panel company in the MediNET case study can be classified into active, archived, or blocked based on its request, activities and payment history.

BRBeans approach provides guidelines to identify trigger points during object-oriented analysis and design. Use Cases and Sequence Diagrams are carefully analyzed to find the rules and *"points of variability"* in a business process. An arrow notation is

introduced to specify such points in Sequence Diagram whilst an attached text describes which rules should be fired at that point. In a textual specification, keywords such as 'except when', 'unless', and 'depend on' can be used to indicate the points of rule externalization. Each point of variability is translated into a trigger point namely a set of interfaces in the rule client that locate and pass the information for business rules execution. The architecture of BRBeans framework showing the interaction between its components are illustrated in Figure 2-7.

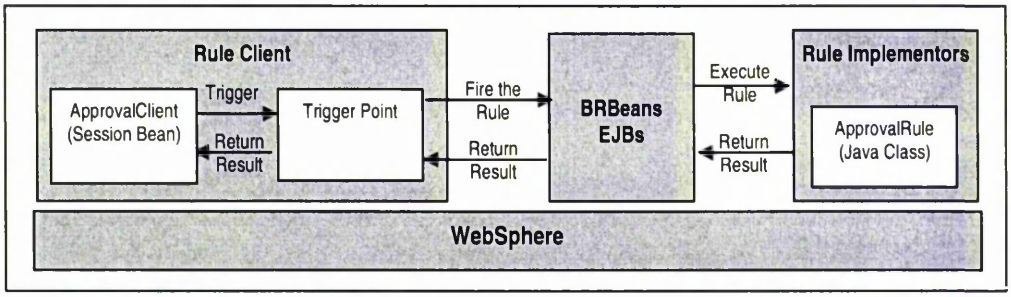

Figure 2-7 The interaction between BRBeans framework components [Kovari et al., 2003]

During development, rule implementors are created to implement the business rules and the common business logic that are going to be used by BRBeans Enterprise Java Beans (BRBeans EJBs). Next, business rules are configured using the stand-alone Rule Management Application and implemented as EJBs. They are arranged in folder and defined by a set of properties such as rule name, folder name, start date, end date, implementor name, firing parameters, and firing location. Finally, trigger points are developed and placed in the rule client implementation to provide an interface to access rules via BRBeans EJBs.

At runtime, the rule client creates an instance of a TriggerPoint object and executes one of the object's trigger methods that satisfy its current context. Next, the TriggerPoint object find the Business Rule Beans rules by names defined using the Rule Management Application. When the rule is found, the TriggerPoint object invokes the rule by calling the fire method on BRBeans EJBs, which in turn executes the rules implemented in the RuleImplementor object. The BRBeans infrastructures are similar to component broker architectures in most distributed object technologies such as OMG's CORBA, Sun's

Java RMI, and Microsoft's DCOM.

2.5 Critique on State-of-the-art Business Rule Approaches

For a systematic comparative review and evaluation of state-of-the-art business rule approaches, an appropriate evaluation framework and technique should be defined. Floyd states that the approaches cannot be described and evaluated without a reference to a specific conceptual framework and perspective for system development [Floyd, 1986].

2.5.1 The Evaluation Framework

Before going further into the description of the evaluation criteria, it is important to decide the goals of the evaluation framework. Although the notion of business rules has been coined in the last two decades, business rule approaches are relatively new compared to software development approaches in other paradigms such as structured and object-oriented. Moreover, the business rule approach is not a new software development paradigm in that it uses or extends the techniques from other paradigms. Therefore, the main goal is not to find the best approach, but to systematically set the right directions of the will be proposed business rule approach to software evolution. The evaluation framework is used to identify the desirable features of business rule approaches as well as to identify the opportunity of improving the existing approaches.

The framework consists of a set of criteria which addresses not only classical software development method attributes but also properties which are uniquely found in business rule modelling. In order to provide a sound evaluation, some criteria of the evaluation framework and techniques are taken and adapted from a set of well defined evaluation criteria of various domains such as system analysis [Yadav et al., 1998], object-oriented [Hong et al., 1993], agent-oriented [Dam and Winikoff, 2003; Sturm and Shehory, 2003], and business rule modelling [Herbst et al., 1994; Kardasis, 2001]. Several criteria are also derived from the common and desirable quality attributes normally found in business rule literatures. The framework is divided into four closely related components i.e. concept, modelling language, process, and pragmatics.

A **concept** is an abstract or general idea that serves to indicate a category or class of instances such as entities, relations, and activities. In this framework, several concepts are derived from business rule definition, metamodel, and taxonomy of the reviewed approaches. The concepts are compared in term of their availability in a particular approach and how detail they are treated. Among the list of relevant concepts to compare are:

- *Business rule definition*: The definition of business rule considered by the approach may reflect the perceived perspective of business rules and type of systems under consideration. It also reflects the scope or limitation of their modelling language and process in a development of a software system.

- *Business rule taxonomy*: The taxonomy of business rules is an appropriate source for deriving concepts in a business rule approach. It may also indicate how complete and detail treatments were made by the approach in dealing with various types of business rules. Although there is no standard taxonomy for business rules, the approaches will be compared based on the following common business rule categories:

    - *Structural rules*: Structural rules constrain the properties of the entities or their relationships such as entity type, relationship type, cardinality, and optionality. Some approaches also include the permitted behaviours of an entity as structural rules.

    - *Behavioural rules*: Behavioural rules constrain the behavioural aspect of the business. It is often written as a statement that determines the reaction of the system against the occurrence of event(s) and/or the satisfaction of condition(s). Other popular examples of behavioural rules are pre- and post-conditions of an operation.

    - *Derivation rules*: Derivation rules produce a new value or fact based on the existing value or fact. It is often further categorised into computation and inference.

- *Business rule management elements*: Business rule management elements refer to modelling or specification components which are used to organize or manage the business rules excluding the typology elements. This type of elements is included for the purpose of improving the traceability and changeability of business rules. For example, rule set is used as a way to organise the closely related business rules. Organization elements also include rule environment components that are related to business rule such as owner and business process.

Modelling language is a set of symbols (either graphical or textual), syntax and semantics which is defined for supporting and representing the specified concepts of an approach. In business rule conceptual modelling, most approaches often define their modelling language using sentence templates or EBNF definitions. A modelling language is compared using the following criteria:

- *Understandability*: The degree of understandability of business rule representation is greatly determined by the language. Natural language is highly understandable compared to formal language or programming language. However, structured natural language or pseudo-code is more understandable to system developers.

- *Expressiveness*: Expressiveness is related to a capability of the expressions produced by the modelling language in completely and correctly presenting the business rule concepts. The modelling language must have enough constructs to represent or specify different types of business rules.

- *Unambiguity:* Unambiguity denotes the capability of modelling language to be interpreted correctly and precisely. There two common factors that cause ambiguity in the language expression: conflict of meaning and redundancy. The former happens when a single business expression can give several meaning whilst the latter occurs if there are more than one expression refer to the same meaning.

- *Formality*: Formality is the measure of rigour in the specification produced by a modelling language. It is important for the implementation, executability,

testability, and preciseness of business rule expressions. At the highest level of rigour, the specification uses the formal mathematical language. The most common formality introduced by business rule approaches is structured language using sentence templates and graphical notations. It is important to note that, in many cases, higher formality may reduce understandability.

- *Evolvability*: Evolvability refers to the flexibility of a software system in dealing with business changes. In other words, evolvability is the ability of an approach to localise rule changes only to the business rule components or minimize the impact of rule changes on other software components. Apart from flexibility, traceability is also important for evolvability since it tracks the dependencies between software artefacts. A modelling language is assumed to be high evolvability if it explicitly provides traceability between software requirements, design, and implementation.

Process is a series of well-defined steps or activities with corresponding input and output products which assist users (such as analysts, developers, and managers) to perform software development tasks. The criteria which are related to process component are:

- *Lifecycle coverage*: Lifecycle coverage is a set of common development phases defined by the evaluated approach. These phases include analysis, design, implementation, and maintenance that are generally found in most software lifecycle models.

- *Process description*: Process description is how the availability of detailed descriptions about steps or activities within the scope of its lifecycle coverage. The description includes deliverables at each stage (documentations or models) and guidelines for quality or project management.

- *Coherence*: Coherence is the degree of logical connection from a flow of one step to another step. The connection can be an activity or deliverable that links the previous step to the next step in software process.

- *Support for evolution*: The availability of process description regarding the maintenance or evolution of business rules and the relevant software design components.

Pragmatics is concerned with the practical aspects of deploying and using the approach. It includes both management and technical issues. The former emphasis on the total cost resulted from using the approach such as consulting services and purchasing of tools. The latter is about the issues in the application of the approach in establishing software development tasks. Among the criteria associated with this component are:

- *Communicability*: In software development, both models and processes must support the communication between various groups such as manager, developers, and end-users. Higher clarity, readability and understandability specifications may enhance the communicability. In contrast, higher complexity of the deliverables and process description may reduce the ability of smooth communication.

- *Usability*: Usability is the easiness of applying the process and syntax. It is hard to use an approach if the modelling language syntax is too rigorous or too vague, or the process is too complex to be followed. Similarly, an approach is less usable if there is not enough description on the process.

- *Resources availability*: The availability of resources such as text book, user's group, and training are important for the users in facing their everyday problem in establishing their software development tasks.

- *Openness*: Openness is either the solution of the approach is independent or dependant of certain implementation platforms such as architectures, paradigms, or programming languages. For business rule modelling, the implementation-independent approach is more acceptable by its users.

The components of the above evaluation framework are closely related to each other since some evaluation criteria of one component have some degree of influence to the criteria of another component. For example, higher understandability and unambiguity

(modelling language criteria) may produce higher usability (pragmatics criterion). The relationships among the framework components, as well as the relationships between the framework and its application domain, are illustrated in Figure 2-8.

As shown in Figure 2-8, concept represents the abstraction of the real world entity, relation, or activity which is related to the system under development. Modelling language and process are developed to support the list of concepts defined by the approach. During its application, process uses the modelling language as an input, output, rules, or guidelines of its activities. Pragmatics acts as a 'filter' that determine the degree of acceptance of the approach by users in the real world or application domain. Most of the features of modelling language and process may influence the criteria defined in pragmatics components.

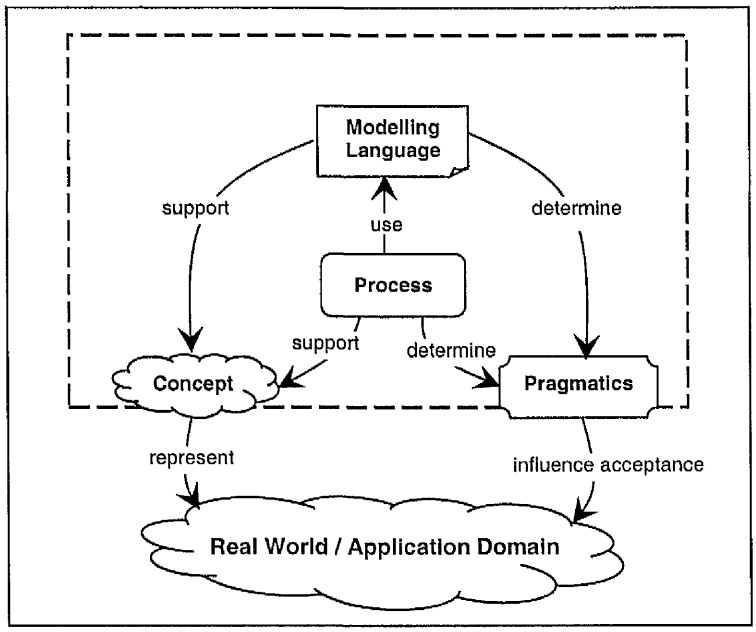

Figure 2-8 The Evaluation Framework

2.5.2 The Comparative Evaluation

The previously described evaluation framework is suitable to be used with any evaluation techniques. For the purpose of this research, feature analysis technique was used to compare the selected business rule approaches. Although feature analysis is a

subjective evaluation technique, it is easy to perform if the criteria are well-defined [Siau and Rossi, 1998]. If a metamodel is available for each compared approach, this evaluation technique can be combined with metamodel analysis technique for a more objective evaluation [Hong et al., 1993].

Since the reviewed business rule approaches inclined toward two different categories i.e. business rule conceptual modelling and evolvable software systems, the evaluation should be done to reflect these two different contexts. However, most of the criteria of the above framework are designed to be as generic as possible for use by both categories. The results of the comparative evaluations of business rules conceptual modelling and evolvable software systems are summarized in Table 2-3 and Table 2-4 respectively.

2.5.2.1 The Comparative Evaluation of Business Rule Conceptual Modelling

In terms of business rule definition, the compared approaches focus their definition in two perspectives: business and information systems. BRS emphasises that business rule should be consider as an element of business whilst BRG and BROCOM emphasize the business rule as a part of information systems. Regarding business rule taxonomy and management elements, BRG provides a high number of elements related to structural rules since it defines the term and fact in details. However, it has an average number of elements for behavioural rules, derivation, and rule management elements. BROCOM is found to have a very low number of structural and derivation rule types in their metamodel. Nevertheless, it provides the highest number of concepts that described behavioural rules. In fact, all BROCOM rules constrain the behaviour of the systems. Similar to BRG, BROCOM also has an average number of business rule management elements. BRS is observed to have a medium count of structural and behavioural rules. Although structural rule count in BRS is only one, but it is considered as medium since it excludes terms and facts in their taxonomy, which are clearly explained. BRS has a very high number of business rule management elements that are included in its RuleTrack metamodel.

With respect to modelling language, the approaches are compared based on their metamodel which is often available in a form of graphical representation, context-free

grammar definition, or formal mathematical logic. Both BRG and BROCOM have a medium level of understandability; BRG provides natural language rule expressions, which is easily comprehended by business users whilst BROCOM provides structured expressions which facilitate the understanding for system developers. However, BRS has a higher level of understandability by providing structured rule expressions which is easily understood by business users and effortlessly linked to information technology components. Both BROCOM and BRS have a high level of expressiveness since they are rich in language construct to the textual representation for all of their business rule types. In contrast, BRG does not define the textual language construct although it specifies some concepts in their graphical metamodel. Regarding unambiguity, BROCOM has a high unambiguity since there is no detected conflict and redundancy in its language definition. Both BRG and BRS have an average level of ambiguity. The former inherits the ambiguity from the natural language flexibility whilst the latter is detected to have several ways to express the same meaning such as the redundancy of derivation and inference rules. In terms of formality, BRS provides the highest level of language formality via its predicate calculus definition of business rules. BRG and BROCOM have an average level of formality by providing graphical and/or context-free grammar representations. In terms of evolvability, both BRG and BROCOM has an average level of evolvability in that it provides a definition for structured rule expression and includes rule management elements in their metamodel. BRS is considered as more highly evolvable than BRG and BROCOM because it has more rule management elements, which in turn improve the traceability during evolution, in addition to very structured expression via its sentence templates.

Regarding modelling process, BRG is unsuitable to participate in the comparison since it does not describe the process. As regards BROCOM and BRS, both of them have a detailed and clearly explained process description. Their steps and deliverables are logically link from one stage to another stage of modelling process. Both BROCOM and BRS support business rule evolution by providing detailed guidelines and automated tools. The only different between BROCOM and BRS is the lifecycle coverage. The former only covers analysis phase whilst the latter embraces analysis and design phases.

Table 2-3 The comparative evaluation of business rule conceptual modelling

Criteria \ BR Approach	BRG	BROCOM	BRS
Concepts			
Business Rule Definition	IS	IS	Business
Business Rule Taxonomy			
- Structural Rules	High (10)	Low (0)	Medium (1)
- Behavioural Rules	Medium (8)	High (>30)	Medium (8)
- Derivation	Medium (2)	Low (0)	Medium (2)
Bus. Rule Management Elements	Medium (5)	Medium (9)	High (>30)
Modelling Language			
Understandability	Medium	Medium	High
Expressiveness (business rules)	Medium	High	High
Unambiguity	Medium	High	Medium
Formality	Medium	Medium	High
Evolvability	Medium	Medium	High
Process			
Lifecycle coverage	A	A	A + D
Process description	N/A	High	High
Coherence	N/A	High	High
Support for evolution	No	Yes	Yes
Pragmatics			
Communicability	Medium	High	High
Usability	Medium	High	High
Resources availability	Low	Medium	High
Openness	High	Medium	High

Lifecycle coverage: A-Analysis, D-Design, I-Implementation, M-Maintenance

In connection with pragmatics criteria, BRS is found as the most practical approach in the capturing and management of business rules. Its RuleSpeak sentence templates may facilitate the communication and usability. The availability of the methodology, rule management metamodel and tool improve the pragmatics aspect of BRS. In terms of openness, BRS implementation is open to any type of implementation including hardware, software, business logic technology, or manual process. Similar to BRS, BROCOM also has a high degree of communicability and usability by providing a structured format of business rules via its EBNF definitions. However, its resources availability is slightly lower that BRS since BRS is supported by commercial organizations. The openness is also lower than BRS since BROCOM is meant for active database technology. As BRG is not supported by any process or tool, it is found the least practical compared to other approaches as far as system development approach is

concerned. It has an average level of communicability and usability based on the availability of its metamodel and plenty of examples. It is very low in resources availability since it has no support on process or tool. However, it is flexible in terms of implementation since it says nothing about implementation.

2.5.2.2 The Comparative Evaluation of Evolvable Software Systems

With regard to modelling concepts, the concepts provided by BRBeans are more exhaustive than those provided by AOM and Coordination Contract. BRBeans is the only approach that gives an explicit business rule definition. Its business rule taxonomy covers a wide range of business rule types found in, or at least easily mapped to conceptual modelling. In contrast, Coordination Contract only deal with ECA rules, and AOM provide the primitive, composite, and workflow rules, which is difficult to map to business rules in business perspective. In terms of business rule management elements, BRBeans provides folder and rule attributes such as business intent, original requirement, and description for the purpose of managing the rule changes. On the contrary, there is no management element provided by the other two approaches.

In terms of modelling language, the current comparison is different from the previous comparison in that the former also considers software modelling aspects in addition to business rules specification. Furthermore, the language redundancy criteria in representing business rules is deemed non-applicable since most design and implementation approaches contain far less rule syntax compared to conceptual modelling.

In the sense of software modelling, AOM is found the most understandable since it provides the graphical representation that uses a set of well-defined and proven design patterns. BRBeans and coordination contract is slightly less understandable given that it relies on the existing object-oriented technology syntax and textual specification of software components. However, the structured textual specifications provided by BRBeans and coordination contract increase their ability in expressing business rules. The structured specifications make them better than AOM in terms of the expressiveness of representing business rules. Although rich in graphical representation, AOM is low expressiveness since the business rule expression must be defined in terms

of rule objects. Regarding the formality of modelling language, AOM has the lowest degree of formality since it only includes the graphical representation. BRBeans formality is slightly higher because it uses programming language as modelling specification. Coordination contract provides the highest formality by introducing highly formal language, i.e. CommUnity, for specifying contract semantics in addition to structured specification and design patterns. In terms of evolvability, there is no doubt that all of the reviewed approaches are highly evolvable with their own styles. AOM, Coordination Contract, and BRBeans respectively introduce rule object, coordination rule, and BRBeans EJB to separate the volatile parts from other stable parts of software system. These components can be independently changed with a minimum impact on other components.

With respect to development process, BRBeans was found superior than other compared approaches. It covers all of common activities in software lifecycle including testing and maintenance. It provides a very detailed process description in a form of technical specification, programmer's handbook, and other publications. Regarding coherence between activities, it offers a smooth link and sequence from one activity to the next activity. In terms of support for evolution, BRBeans provide a rule management tools that manage the rule changes without affecting other components. IBM is currently improving business rule management using Fusion framework [Rouvellou et al., 2004]. Coordination contract lifecycle coverage is slightly narrower than that of BRBeans since it excludes system analysis. The process description of coordination contract is also rather fewer than that of BRBeans. Relating to support for evolution, coordination contract is also supported by tools but with less guidelines. The lifecycle coverage of AOM is harder to be evaluated since it does not employ the classic waterfall lifecycle. Instead, it uses the evolutionary development process which repeats the development activities based on user's feedback. There is no available detailed description of AOM's development process. In connection with support for evolution, there is no an available automated tool or detailed discussions on maintaining software evolution.

Table 2-4 The comparative evaluation of evolvable software systems

BR Approach / Criteria	Adaptive Object Model (AOM)	Coordination Contract	Business Rule Beans (BRBeans)
Concepts			
Business Rule Definition	Implicit	Implicit	Explicit
Business Rule Taxonomy	primitive, composite, workflow	ECA	derivation, constraint, invariant, script, classifier
Business Rule Management Elements	Nil	Nil	Yes
Modelling Language			
Understandability	High	Medium	Medium
Expressiveness (business rules)	Low	Medium	Medium
Formality	Low	High	Medium
Evolvability	High	High	High
Process			
Lifecycle coverage	(Evolutionary)	D + I + T + M	A + D + I + T + M
Process description	Low	Medium	High
Coherence	Medium	Medium	Medium
Support for evolution	Low	Medium	High
Pragmatics			
Communicability	High	Medium	Medium
Usability	Low	Medium	Medium
Resources availability	Medium	Medium	High
Openness	Medium	Medium	Low

Lifecycle coverage: A-Analysis, D-Design, I-Implementation, T-Testing, M-Maintenance

Regarding the pragmatics issue, each approach has its own strength and weaknesses. As far as communication is concerned, all of them have a medium level of communicability. AOM lacks in textual specification for representing business rules although the use of widely known design patterns simplifies the understanding of its software design. Similarly, coordination contract and BRBeans be deficient in graphical representation although it provides an acceptable structure for specifying business rules. In terms of usability, AOM is found to have a low degree of usability in that it requires knowledgeable developers to deal with the meta-level design. Moreover, there is no detailed process description for the development of rule objects. On the contrary, both coordination contract and BRBeans have a slightly higher degree of usability since the process is clearly explained and supported by structured specifications and tools. With respect to resources availability, coordination contract and BRBeans are very much higher than AOM since they are supported by both research and commercial groups.

Their supported resources include research papers, user's manuals, technical specifications, software tools, and support group. Relating to openness, AOM and coordination contract have a medium level of flexibility in choosing the implementation technology; the only possible restriction is that it must use object-oriented technologies. In contrast, BRBeans must be implemented in Java programming language since it is not architecture per se but an application framework.

2.6 Summary and Further Remarks

Many approaches that address various specific problems in software evolution have succeeded in achieving their goals. The majority of them derive their benefits from the outstanding features of object-oriented, distributed system, and component-based technologies such as loosely coupled and highly cohesive components. The three-dimensional view of software evolution approaches highlights the software evolution research trends and pinpoints the desirable features for successful approaches. Based on the reviewed state-of-the-art approaches to software evolution, there are some important points that should be considered by software evolution approaches in order to ensure their success in achieving their goals. In short, software evolution approach should:

- explicitly consider business rules, which were identified as the important sources of changes that bring the highest impact to both business processes and software system;
- utilize or extend the existing software technologies such as software architecture, object-oriented, and component-based;
- at least provide the product and process, which were identified as the compulsory components for a practical solution to evolution problem.

Since business rules are important sources of changes, business rule approaches to software development were reviewed. Although business rule approaches were given different names, for example, business rule-driven approach [Moriarty, 1993], business rule-extended development method [Haggerty et al., 2001], or simply business rules approach [von Halle, 2001b], and they propose diverse development techniques, they share a common goal i.e. to simplify the evolution of a software system due to the

frequently changing business environment. From the reviewed state-of-the-art business rule approaches, they are observed to focus on several techniques in achieving this common goal such as:

- the conceptual externalization of business rules by separating business rule specification from other software models or specifications. For example, they can be captured and represented using their own model and notations during analysis and design. This technique allows users to independently manage the changes of business rule specification.

- the separation of business rule components from other software implementation components. For example, a repository may be used to separately manage and store business rules similar to the concept of database management, or a separate software component may be developed to be triggered by main program in order to execute particular business rules. By externalizing business rules, it is possible to separate the volatile parts from the more stable parts of a software system.

- the establishment of business rule traceability in business and software components. Establishing business rule traceability in a business modelling component supplies the information about the reason of business rules existence whilst linking business rules to software design and implementation components allows the alignment of a business to its supporting software system.

- the use of an appropriate business rule representation. Representing business rules in graphical or textual style may improve the communication between business users and developers. At the same time, the representation should be formal enough to facilitate their implementation in a software system.

- the introduction of various components such as metamodel, process, and tools. These components are important to ensure the applicability of the approach in applying their techniques to the real information systems.

- the utilization of the existing well-proven widely accepted software technologies. Developing solution on top of the existing technologies or proven techniques, such as object-oriented, expert system, distributed object, software

architecture, design patterns, and metamodelling, may increase the chances for the approach to be accepted by software development community.

Nearly all of the business rule approaches mention the above techniques but they lack in detailed information about the way to achieve these objectives. For example many approaches point out that it is important to link business rules to software system but there is no detail explanation on how to perform the linking tasks. Some approaches provide excellent details on certain aspect such as conceptual modelling but failed to provide a detailed treatment on other aspects such as the implementation and evolution of business rules. For these reasons, two groups of business rule approaches were studied for the purpose of investigating a better business rule approach to software evolution i.e. business rule conceptual modelling and the construction of evolvable software systems.

The evaluation framework, which includes concept, language, process, and pragmatics criteria, is developed to systematically carry out a comparative evaluation of the selected business rule approaches. The framework does not only consider the methodological aspect, but it also takes into account the business rule modelling and software evolution issues. There are two main observations which are drawn from the results of the comparative evaluation of the selected approaches.

- With regard to the business rule conceptual modelling approaches, they were found very excellent in the modelling concept and language in representing business rules. The only common drawback of these approaches is their coverage in software lifecycle. There is no detailed information on the implementation of their business rule specification in a software system since they focus on the business rule lifecycle. Some of them suggest the business rules may be implemented using a database (repository) or separate software modules. It is agreed that storing rules in a database or separate software modules makes rule management easier especially for the centralized database. However, simply implementing all business rules in a database may cause poor system performance. Data communication and rule interpreter are among factors that may slow down the system execution. The poor performance system is likely to become unusable.

- Regarding the construction of evolvable software systems, they provide exhaustive information on the software architecture, implementation and process. However, their modelling concepts are limited in capturing and representing business rules. In addition, their business rule specification is too far from the user language, which in turn introduces a gap between analysis and design in terms of business rule components. The conceptual model developed during analysis, as proposed by BROCOM or BRG, is likely to be irrelevant during software design although it is excellent in representing rules in a universe of discourse. The current approaches unable to utilize the rich abstractions in the proposed conceptual model. Due to their modelling constructs, most of them only translate a small part of the conceptual model, thus wasting a lot of useful information in the model such as links to organizational facts.

The analysis of the state-of-the-art business rule and software evolution approaches identifies the opportunities of improvement, which should be done in order to successfully exploit the benefits of business rules in software evolution. The identified desirable features may be used to set the directions of future software evolution approach driven by business rules. The approach that takes into account most of the important issues discussed in this chapter will be explained in the next chapter.

Chapter 3

The BROOD approach

The inherent problems in software evolution and the gap in the prominent business rule approaches, which have been elaborated in the previous chapter, motivate the investigation of a novel approach in software evolution that is driven by business rule changes. The proposed approach, **B**usiness **R**ule-Driven **O**bject-**O**riented **D**esign (BROOD), brings together the benefits from business rule, model-driven and object-oriented approaches to software development in producing a software system that is flexible to business changes.

In this chapter, the overview of BROOD approach and the detailed discussion of its metamodel are discussed. At the outset, this chapter presents the rationale behind the BROOD approach. Next, it provides the overview of two main BROOD components, i.e. the process and metamodel. The detailed discussions on the three main components of the BROOD metamodel i.e. business rules, software design, and rule phrases (linking elements) are made in the following sections. A chapter summary is presented at the end of this chapter.

3.1 The Rationale

In the previous chapter, the in-depth study on software evolution and business rules suggests the main problem areas and gives the ideas about the desirable criteria to improve the existing approach. This study leads to the rationale that motivates the development of the BROOD approach. The rationale ranges from the well-known fundamental evolution problems such as Lehman's Laws and maintenance cost, to specific limitations of the state-of-the-art approaches in business rules and software evolution.

The following rationales are related to the fundamental software evolution problems in general:

- *Software evolution is an inevitable process.*

  The most fundamental reasons that motivate BROOD to focus on software evolution are a number of principles proposed by Lehman's Laws [Lehman and Belady, 1985]. According to Lehman's Laws, a software system that is used in a real-world environment inevitably must change or become progressively less useful in that environment. Software evolution can be thought as the natural characteristic of business software system since today's business environment is frequently and rapidly changing in nature. Lehman's Laws also state that the software structure tends to become more complex due to the implemented changes and its size must continue to grow to accommodate new user requirements. Therefore, there is a need to introduce a method that facilitates the management of the increasingly complex and larger size software system during evolution phase.

- *The cost of modifying software is enormous.*

  Erlikh reported that an informal industrial poll indicates that 85% to 95% of software costs are evolution costs [Erlikh, 2000]. In his survey on software maintenance, Lientz discovered that 50% to 70% of the overall software maintenance efforts is dedicated to fulfil new or changed user requirements [Lientz, 1983]. These facts indicate that there is a crucial need for a better

solution in producing a software system that is more resilient to requirement changes. By focusing on the way of evolving requirement changes, and automatically propagating them to software design, the enormous evolution costs might be reduced.

The following rationales are derived from the facts on software evolution discussion in the previous chapter:

- *The sources of changes are the root of evolution problem.*

  By considering the sources of changes as the root of evolution problem, a more holistic view of the problem can be realized. In other words, the holistic approach to software evolution differs from the conventional approach in that it takes into account the sources of changes in addition to the software technology. This fact is aligned with the views on software evolution discussed in Chapter 2 that suggest software evolution can be viewed in term of the environment and the process or technology. The environment can be interpreted as the sources of software changes, which is often referred to business rules.

- *Product and process should be the compulsory components in a successful software evolution approach.*

  Product and process components should be considered as the compulsory components in software evolution approach. This assumption is based on the review of software evolution approaches made in Chapter 2 that reveals the approaches is more likely applicable if they include both product and process components. Product component should embrace evolvability as one of its quality attributes, while process component should include the detailed description on how to perform the evolution tasks.

The following rationales are deduced from the state-of-the-art business rule approaches in the previous chapter:

- *Business rule externalization is fundamental in separating the volatile part from the stable part of software system.*

  Having recognized that business rules are the most volatile parts of business software system, and business rule changes bring the highest impact on both software system and business processes, many approaches aimed to externalize business rules from other software components. At the conceptual modelling level, they provide separate syntax and semantics for modelling business rules. This effort is not only localize the changes to business rule components, but it also increases the understanding and maintainability of business rule specification itself. At the implementation level, they create separate software components that implement business rules. As a result, the business rule changes will only localize to such components, and reduce the impact of changes to overall software structure.

- *Linking business rules to software system components can improve software evolution.*

  In the previous chapter, the comparative evaluation was made on the prominent software evolution approaches that incorporate business rules as their key elements. It is observed from the evaluation results that there is one group which is very excellent in producing evolvable software systems but they slightly be deficient in representing business rule concepts. In contrast, there is another group that is very excellent in dealing with the concepts related to business rules, but they provide relatively little description on the design and implementation aspect of business rules. Introducing the linking between the conceptual model of business rules to software design and implementation may increase business rule traceability, which in turn improves software evolution. Traceability may be achieved using a proper documentation of business rules specification [von Halle, 2001b]. Traceability is highly desired in software evolution since there is a need to track the related software components in order to implement business rule changes, which is originated by the changes in business environment. The link may also ensure the fulfilment of user requirements since the implemented

software system is linked to business rule specification that represents user requirements.

- *Business rule specification should naturally define business rules from business user's perspectives and, at the same time, be well structured enough to be implemented or linked to software design.*

  Business rule specification should be easily understood by business users to allow their involvement in the construction and management of business rules. This principle motivates many approaches to provide a specification that mimics the natural language. However, the structure of the specification should be formal or structured enough for linking the represented business rules to their implementation within software system. The introduction of sentence templates is a popular example on how most approaches conform to this principle.

There are also other rationales relevant to both software evolution and business rule approaches that influence the BROOD features:

- *Building solutions on top of the proven, widely accepted software technology increases the chances of the solutions to be accepted.*

  The leading approaches in software evolution focus on both business rules and software technology. With regard to software technology, most of them utilize the widely accepted, well proven techniques in the existing areas such as object-oriented, component-based, and architectural connector to name a few.

- *Evolution problem should be tackled at metamodel level.*

  A model is important to provide a blueprint for producing a workable software system. The evolvability of the software model is largely determined by the elements of its metamodel. Therefore, the evolution problem should be tackled at the metamodel level. In terms of business rule-driven software evolution, the business rule components should be linked to software components at their metamodel level to facilitate their modelling and the propagation of business rule changes to software system.

- *User requirement changes may happen prior to implementation.*

  A traditional view on software evolution is closely related to the traditional waterfall software lifecycle model where the evolution happens after the system is delivered i.e. during maintenance phase. However, in nowadays rapidly changing business environment, changes might happen prior to software deployment. Therefore, there is a need to support the evolution during early phase of software development. The best solution is to link business rules, which may be considered as the rapidly changing user requirements, to software design, which represents the software components.

3.2 BROOD Overview

Business rule specification is considered as a central part of the BROOD approach. In this approach, the metamodel is systematically developed to enable the appropriate specification of business rules as well as to link the specification to object-oriented software design. Using BROOD, the implementation of changes is performed at a software design level, which is also known as model-driven software evolution.

In order to be considered as a complete and practical approach in a development of evolvable software system, BROOD considers both *product* and *process* perspectives of the development and evolution a software system.

The *product* is defined using the BROOD metamodel, which specifies the structure for business rule specification, software design, and their linking elements. The BROOD metamodel is considered as the core of this approach since it determines the evolvability feature of the developed software applications. The metamodel is complemented by a language definition based on the context-free grammar EBNF, which is included in appendix B. The language definition defines the allowable sentence patterns for business rule statements and describes the linking elements between business rules and the related software design elements.

The *process* refers to a set of systematic and well-defined steps that should be followed during software development and evolution. BROOD process emphasizes several important activities in a software lifecycle that contribute to a more resilient software

system. The scope of BROOD process is limited to analysis, design, and evolution phases. The flow of activities in each phase, which are arranged according to their process roles, is shown in Figure 3-1. The detailed explanation on the BROOD process is presented in the next chapter. In summary, the BROOD phases are described as follows:

- During analysis, the requirements model produced in the previous phase (typically called requirements phase) is used by software architect and component engineer to produce the analysis model, which consists of architecture description, class diagram, statechart diagram and package diagram. The use of the result from the previous phase ensures faithful translation of the requirements to the design of the software under development. In terms of business rule modelling, the informal business rule statements and other requirement models produced by the previous phase are analyzed to develop an initial business rule specification.

- During design, the analysis model is further refined to produce a suitable structure for implementation. Some implementation-related information is added to the model. For the purpose of future evolution, each business rule statement in the business rule specification is further refined and linked to its respective software design elements. The refinement and linking of the business rule specification may require additional detailed information to be added to the software design.

- During evolution, BROOD facilitates the management of rule changes and the propagation of the changes to software design. By documenting correlated business rules, the software design can be automatically changed according to the rule changes even though the changes take place prior to software implementation.

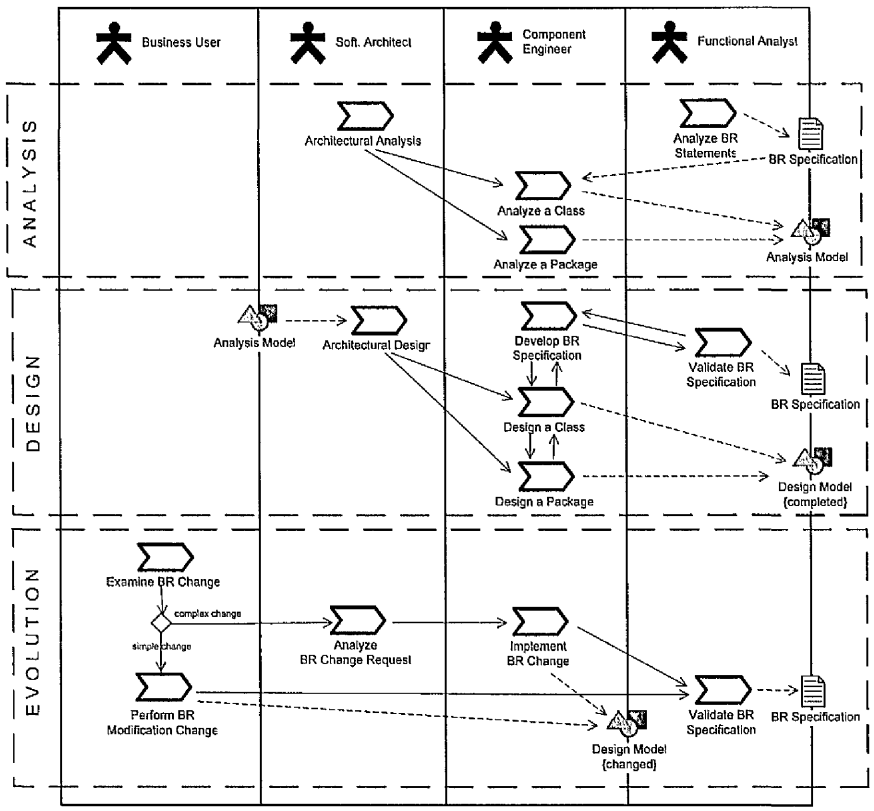

Figure 3-1 The BROOD Process

The prototype of the automated tool that supports the core activities of the BROOD process were developed using the configurable Generic Modelling Environment (GME). It aims to demonstrate the capability of automating the important aspects of the BROOD approach such managing business rule changes and propagating them to software design. The application of the BROOD approach in an industrial-strength MediNET case study is used to experiment, improve, and demonstrate the proposed software development and evolution concepts.

With reference to the BROOD metamodel, there are three main components of the metamodel that are required in order to link and propagate changes of business rules from conceptual domain to software domain i.e. the business rules metamodel, the software design metamodel and the linking elements between the two. The business rules metamodel provides an exhaustive typology and management elements of business rules. The formal language definition is derived from the business rule metamodel to define the syntax of business rule specification. A widely accepted

Unified Modeling Language (UML) is taken as the software design metamodel. Although UML provides various useful models, this research only uses class diagram for modelling the static aspect and statechart diagram for modelling the dynamic aspect of software system. Some approaches introduce the extension to the existing standard UML metamodel by adding elements to cope with new semantic definitions dedicated to business rules implementation. However, BROOD aims to avoid this extension by introducing linking elements in its metamodel. The linking elements are written as phrases that form business rule statements, and they are mapped to their respective software design elements. Most of the business rule examples presented in this chapter are taken from the MediNET case study.

3.3 The Business Rule Metamodel

The ultimate aim of the BROOD metamodel is to support the linking of business rules to software design, which in turn facilitates the traceability and propagation of the rule changes to its related design components. Since business rules are often managed by business users, the metamodel should naturally define business rules from the users' perspectives. At the same time, the definition should be well structured enough to be linked to software design.

The business rule metamodels proposed by most of the existing prominent business rule approaches in conceptual modelling are suitable for business users. Among the important aspects of the metamodels including business rule typology, rule management elements, and sentence templates. However, some templates of these approaches, such as in [Hay and Healy, 2000; von Halle, 2002; Ross, 2003], are inappropriate and redundant for the purpose of downstream development although they provide an exhaustive business rule typology. Other approaches [Herbst, 1997; Morgan, 2002], although they provide a good set of templates, are less comprehensive in their typology. Almost all of them do not have a structure suitable to be linked to software design; they also do not provide a detailed mapping of their metamodel elements to software design. For these reasons, the above business rule metamodels were further improved to achieve a set of appropriate typology and templates for the purpose of linking and propagating business rule changes to software design. The initial concept of the metamodel was

introduced in [Wan Kadir and Loucopoulos, 2003; Wan Kadir and Loucopoulos, 2004b] and the technique for linking and propagating business rule changes to software design were briefly described in [Wan Kadir and Loucopoulos, 2004a].

At the outset, three main desirable characteristics were set for deriving an appropriate business rule metamodel which fit the purpose of the aims of this research. First, it should have an exhaustive and mutual exclusive typology to capture all possible types of business rules. Second, it should have the structured forms of expressions for linking the business rules to software design. Third, it should include rule management elements to improve business rule traceability in business domain, which consequently simplifies business rule management. These three characteristics form the basis for the development of the business rule metamodel, which is shown in Figure 3-2.

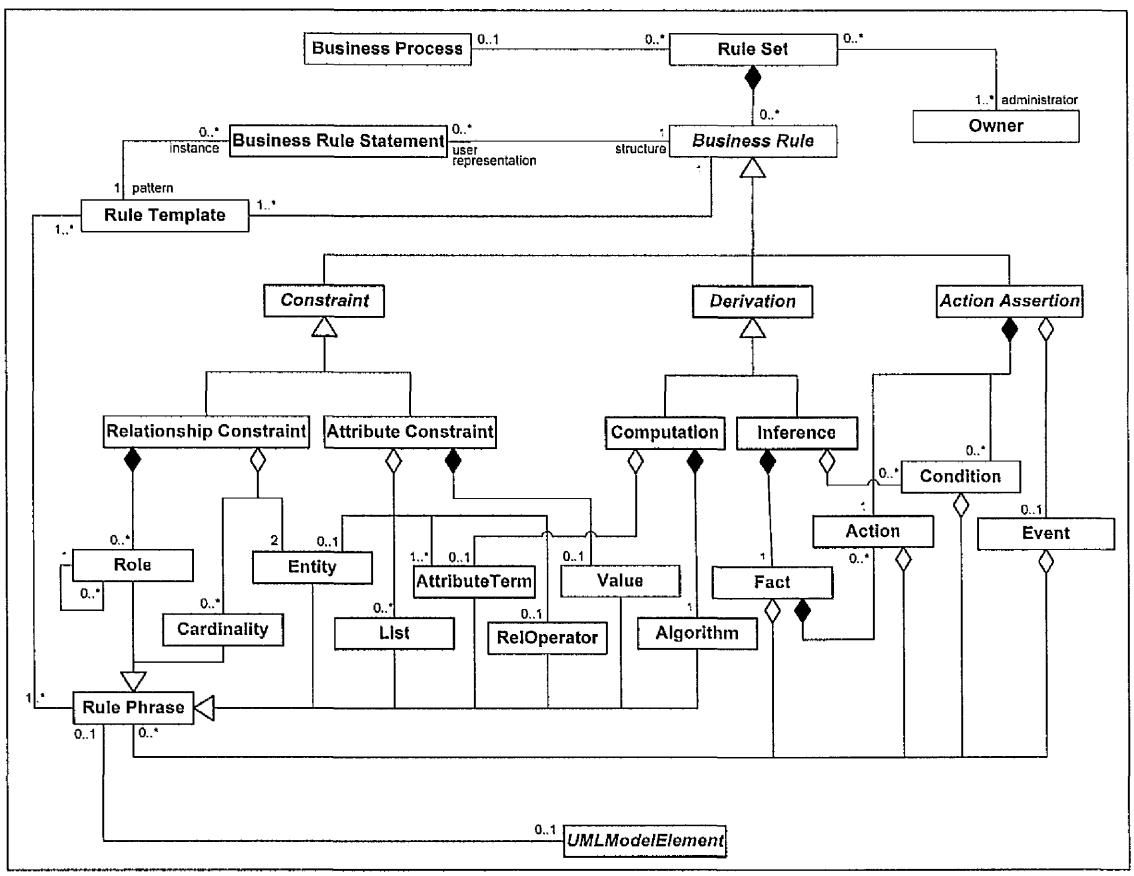

Figure 3-2 The BROOD business rule metamodel

3.3.1 Business Rule Typology

As shown in Figure 3-2, the metamodel classifies business rules into three main types i.e. constraint, action assertion, and derivation.

3.3.1.1 Constraint

Constraint rules specify the static characteristics of business entities, their attributes, and their relationships. They can be further divided into attribute and relationship constraints. The former specifies the uniqueness, optionality (null), and value check of an entity attribute. The later asserts the relationship types as well as the cardinality and roles of each entity participated in a particular relationship. The syntax of constraint rules are further specified by the following EBNF definitions:

```
constraint     = att_constraint | rel_constraint;
att_constraint = entity, ('must have' | 'may have'), ['a unique'],
                 att_term
                 | att_term, ('must be' | 'may be'), relational_op,
                 (value | att_term)
                 | att_term, 'must be in', list;
rel_constraint = ( [cardinality], entity, 'is a/an', role, 'of',
                 [cardinality], entity
                 | [cardinality], entity, 'is associated with',
                 [cardinality], entity
                 | entity, ('must have' | 'may have'), [cardinality],
                 entity
                 | entity, 'is a/an' entity );
```

In the above definitions, the first valid sentence for attribute constraint determines the optionality and uniqueness of an entity attribute. The presence of string literal `'must have'` indicates that the attribute is mandatory whilst the keyword `'may have'` indicates that it is optional for the attribute to have a value. The use of `'a unique'` keyword shows that the attribute must hold a unique value at all times. The examples of attribute constraints in MediNET are shown below:

- Patient must have a unique patient registration number.
- Patient may have a passport number.
- Bill must have a unique bill number.

The remaining two definitions for the attribute constraint refer to the value check of an entity's attribute. The first sentence definition specifies the comparison between a value of an attribute with the value of another attribute or any literal value. The comparison is

made using a set of relational operator such as 'less than', 'equal', 'greater than', 'greater than or equal', 'less than or equal', and 'not equal'. Literal value may be any value from the same type with the type of the compared attribute. The second definition restricts the value of an attribute to be only from the listed values. The listed values are often represented by a variable which contains a set of enumerated values. For example:

- The amount of Bill must be less than the maximum bill amount set by the paymaster.
- An employee level of a Panel Patient must be in {employer, executive, production operator}.

For relationship constraints, the first definition states the role played by an entity in a particular relationship with another entity. The cardinality of each participated entity may also be defined using this sentence pattern. The second definition represents a general association between two entities whilst the third definition specifies the containment of an entity in another entity. The final sentence definition of relationship constraints represents the generalization relationship between sub-type and super-type entities. The examples of relationship constraints in MediNET are given below:

- Clinic item is a/an item type of bill item.
- Bill must have zero or more bill item.
- HCP Service Invoice is a/an Invoice.

It is also important to mention that the use of words `'must'` and `'may'` implies different meaning in a rule statement. The former specifies a mandatory feature that must be satisfied by a business entity, whilst the latter specifies its optional feature. Business events that violate the mandatory constraint should be rejected, whilst violation of guideline rules should only raise a warning to the user.

3.3.1.2 Action Assertion

Action assertion, which is also known as active [Berndtsson and Calestam, 2003] or ECA (Event-Condition-Action) [Herbst, 1997] rule, is a statement that concerns a dynamic or behavioural aspect of the business. Action assertion specifies the *action* that should be activated on the occurrence of a certain *event* and possibly on the satisfaction of certain *conditions*. The number of triggered events or tested conditions can be more than one, and they can be connected using logical connectives AND or OR. The portion

of the BROOD metamodel that further describes the action assertion rule is shown in Figure 3-3. The EBNF definition for action assertion is

```
action_assertion  = 'WHEN', event, ['IF', condition], 'THEN', action;
```

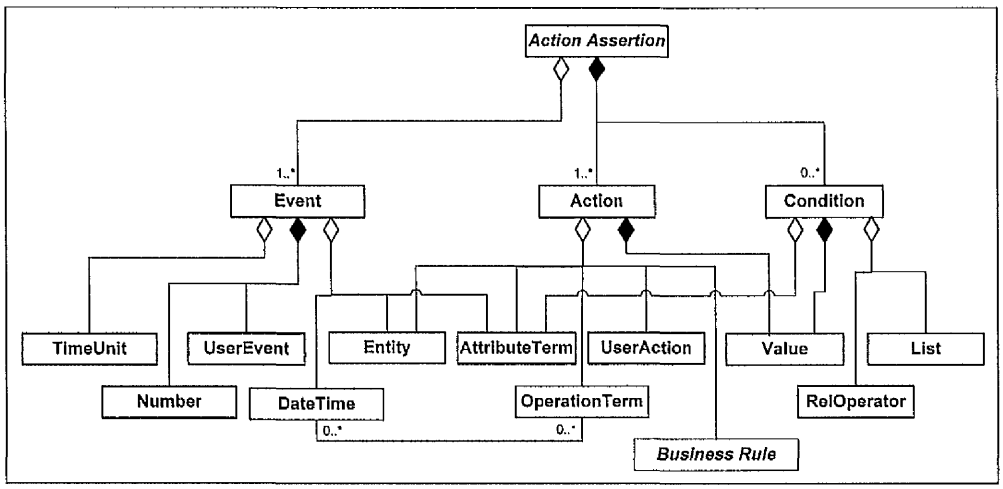

Figure 3-3 Action assertion

Event

In common language, an event is something that happens at a point in time. This research refines the definition of event to a happening or change that is significant to both business and software activities. In other words, the events which are only important to business such as 'customer arrived', and the events which are only important to software such as 'window closed', beyond the scope of this research. The identification of the possible significant events is important for a system to plan ahead the response or action to be taken on the occurrence of the events. For example, 'the receiving of a customer order' and 'the completion of patient consultation' are considered as a significant event since they require a particular response to be taken. The received customer order is needed to be confirmed by checking the condition that the ordered items are available in stock, or rejected if the items are not available. Similarly, the completion of patient consultation indicates that the bill must be created for the patient.

The definition of event is given below:

```
event         = simple_event | complex_event;
simple_event  = change_event | time_event | user_event;
change_event  = att_term, 'is updated' |
                entity, ('is deleted' | 'is created') |
                (operation | business_rule), 'is triggered';
time_event    = date_time |
                n, time_unit, 'time interval from', date_time , 'is
                reached' |
                number, time_unit, 'after', date_time;
user_event    = string;
complex_event = simple_event, {('Or' | 'And'), simple_event {('Or' |
                'And'), simple_event}
```

Event can be either a simple or complex event. Complex event is the combination of two or more simple events using the logical connectives AND or OR. Similar to their usual implication, the use of AND to combine two events in action assertion denotes that both events must occur for the specified action to be triggered, whilst the use of OR only requires the occurrence of either one of the events. Simple event can be categorised into change event, time event, and user event.

Change event includes the updating, deletion, or creation of attributes or instances of an entity, and the invocation of an operation or business rule. In other words, it is concerned with the events generated by the changes of attributes or state of an object. The examples of change event in MediNET include 'updating of the panel status', 'deletion of a patient record', and 'creation of a patient bill'.

Time event is triggered when the specified date and time are reached or certain duration is elapsed. It can be further divided into time point, periodical, and time delay. Time point event is raised when a system clock reach the specified date and time. Periodical event is triggered when a specific event or temporal specification happens for the *nth* time. A delay event happens after a duration which starts at the time point of the occurrence of another event. The examples of time event are '24 Oct 2004 / 18:00', 'every 2 weeks from start date', and '30 days after invoice issue date'.

Each of the above event types describes the valid sentences for specifying business rules. However, it is almost impossible to have a complete set of sentence structure that

capture all type of significant events in a business system. Therefore, user event is introduced to represent any significant event that is not categorised under the above event types. User event is expressed in a natural language. It is often a manual event generated by human activities. The examples of user event are 'patient consultation completed' and 'registration requested'. User event may also be an automated event when it shows the interaction between user and software, for example 'booking confirmed' and 'bill is validated'.

Condition

Similar to event, condition may also be a simple or complex condition. Complex condition is the combination of two or more simple conditions connected with the logical connective AND or OR. Simple condition is a boolean expression which compares a value of an entity attribute with any literal value or the value of another entity attribute using a relational operator. It can also be an inspection of the existence of a value of an entity attribute in a list of values.

The definition of condition is given below:

```
condition         = simple_condition | complex_condition;
simple_condition  = ['Not'], attribute_term, relational_op, (value |
                    attribute_term) | attribute_term, ('in' | 'not
                    in'), list;
complex_condition = simple_condition, ('Or' | 'And'), simple_condition,
                    {('Or' | 'And'), simple_condition};
```

Action

Action is an activity that should be performed by a system in responding to the occurrence of the significant event and the satisfaction of the relevant condition. The execution of action may change the state of an instance of an entity. The enabling and execution of an action is controlled by action assertion rule. Action is defined by the following definitions:

```
action            = simple_action | action_sequence;
simple_action     = trigger_action | object_ manipulation_action |
                    user_action;
trigger_action    = 'trigger', (process | operation | business_rule);
```

```
object_manipulation_action= 'set', att_term, 'to', value |
                            ('create' | 'delete'), object;
action_sequence           = simple_action, {simple_action};
```

As shown in the above definitions, action can be a simple action or a sequence of simple actions. Simple action can be further categorized into three different types i.e. trigger action, object manipulation action, and user action. Trigger action invokes an operation, a process, a procedure, or another rule under certain circumstances. Object manipulation action is a special type of trigger action that sets the value of the attribute or create/delete an instance of an entity. User action is a manual task that is done by system users. During implementation, user action is often implemented as a message displayed to the user.

The examples of action assertion rule with different action types are given below:

- When new invoice created then calculate invoice end date.
- When a patient consultation completed then removes the patient from consultation queue and create bill for the patient.
- When invoice entry updated if stock of drug smaller than re-order threshold then reorder the drug.

3.3.1.3 Derivation

Derivation rule derives a new fact based on the existing facts. It can be categorized into two types i.e. computation, which uses a mathematical calculation or algorithm to derive a new arithmetic value, and inference, which uses logical deduction or induction to derive a new fact. Inference rule is also used to represent permission such as user policy for data security. The definitions for both computation and inference are given below:

```
computation  = attribute_term, 'is computed as', algorithm;
inference    = 'if', condition, 'then', fact;
```

In the above definition, computation rule specifies an algorithm for the calculation of an entity's attribute. The algorithm can be specified using any procedural specification language such as pseudo-code, third generation programming language, scripting language, or natural language. As an example, consider the following rule:

'The amount HCP MediNET usage invoice is *computed as* the amount of transaction fees, which are calculated as the transaction fee multiply by the total number of transactions, plus the monthly fee'

The algorithm for the above rule is defined in pseudo-code as:

```
let a = transaction_fee;
let b = number_of_treated_patient;
transaction_fees = a * b;
invoice_amount = transaction_fees + monthly_fee;
```

Inference rule implies that a fact is true upon finding a true value of a condition. The condition may be a simple or complex condition as defined in section 3.3.1.2. The derived fact is a declarative statement that represents a conclusion from a certain set of circumstances (truth-valued conditions). Facts may represent either a static aspect of an entity property such as the state or value of a particular attribute under certain conditions, or a dynamic aspect such as the permission to perform certain action. The examples of static fact are 'patient is an emergency patient', 'customer is a preferred customer', and 'tenderer is not a qualified contractor', whereas the examples of dynamic fact are 'user may modify bill', 'customer may not cancel order', and 'user may view medical certificate'. Fact is formally defined as follows:

```
fact    = (attribute_term | entity), relational_op, ['a'], value ) |
          entity, ('may' | 'may not'), action;
```

The examples of inference rules are given below:

- If the paymaster's last quarter transaction is more than RM12,000.00 and the paymaster has no past due invoices then the paymaster is a preferred customer.
- If the user type is equal to HR Officer and the user company is equal to patient paymaster then the user may view the patient's medical certificate.

3.3.2 The Rule Templates

In BROOD metamodel, each business rule type is associated with zero or more rule templates. Rule templates are the formal sentence patterns by which business rules should be expressed. They are provided as a guideline to capture and specify business rules as well as a way to structure the business rule statements. Each rule template consists of one or more well-defined rule phrases, which are discussed in section 3.5.

By using the available templates, an inexperienced user may easily produce a consistent business rule statement. Rule templates help users to avoid tedious and repeated editing when creating many similar rules; and ensure the uniformity by restricting the type of rules that can be written by business users. The use of templates also allows the precise linking of business rules to software design elements. The templates can be directly derived from the language definition in Appendix B. Table 3-1 lists the identified templates for each business rule type.

The following convention is used in the definition of the template in Table 3-1:

`[x]`	`x` is an optional variable or value.
`<x>`	`x` is a variable that represents a value or set of values.
`x | y`	either `x` or `y` must exist.
`xyz`	'`xyz`' is a keyword or string literal in a business rule statement.

Table 3-1 Business rule templates

Types	Templates
Attribute Constraint	<entity> must have | may have [a unique] <attributeTerm>. <attributeTerm1> must be | may be <relationalOperator> <value> | <attributeTerm2>. <attributeTerm> must be in <list>.
Relationship Constraint	[<cardinality>] <entity1> is a/an <role> of [<cardinality>]<entity2>. [<cardinality>] <entity1> is associated with [<cardinality>]<entity2>. <entity1> must have | may have [<cardinality>] <entity2>. <entity1> is a/an <entity2>.
Action Assertion	When <event> [if <condition>] then <action>. *The templates of <event> :* <attributeTerm> is updated <entity> is deleted | is created <operation>[<rule> is triggered the current date/time is <dateTime> <number> <timeUnit> time interval from <dateTime> is reached <number> <timeUnit> after <dateTime> <userEvent> *The templates of <condition> :* <attributeTerm1> <relationalOperator> <value | attributeTerm2> <attributeTerm> [not] in <list> *The templates of <action> :* trigger <process> | <operation> | <rule> set <attributeTerm> to <value> create | delete <entity> <userAction>
Computation	<attributeTerm> is computed as <algorithm>
Derivation	if <condition> then <fact>. *The templates of <fact> :* <entity> | <attributeTerm> is [not] a <value> <entity> may [not] <action>

3.3.3 Management Elements

The management elements are also included in the BROOD metamodel for facilitating the organization and management of business rules. These elements include the rule set, business process, and owner. They can be found in the following definitions:

```
business_rule_model = rule_set, owner;
rule_set            = (rule_set | rule_statement), {rule_set |
                      rule_statement}, [owner], [business_process];
```

Rule set is used to simply group business rules into a set of closely interrelated rules. Each business rule model must have a single rule set, which is considered as the root rule set. This rule set must have at least one rule statement or another rule set. One of the popular ways to identify a rule set is through its related business process. For example, the rules 'The bill amount is calculated as the sum of amounts of all bill items' and 'If a patient is a panel patient and his paymaster pays the bill in full, the balance is set to 0 and the bill status is set to unpaid' can be grouped in a rule set which is related to 'bill preparation' process. By properly organizing rules, the complexity of managing a large number of rules can be reduced. In the Appendix A, there are more examples on the business rules from the case study which were naturally and informally grouped into business processes.

Each business rule model must have an owner. An owner may also be defined for a rule set. The owner of a parent rule set is assumed to be the owner of its child rule set if the child does not define its owner. It is important to define the owner information in business rule model to determine the access rights and responsibility to a business rules repository, especially for software systems with multiple user groups that possess different business rules. Owner might be an organization, an individual user, a user group or role that is responsible for the management of the respective business rules. During business rule implementation, each rule set, business process, and owner is given a unique identifier. The BROOD metamodel discussed in this chapter has no intention to include the details of rule management components as it may restrict the flexibility of its implementation.

3.4 Software Design Metamodel

The Unified Modelling Language (UML) [OMG, 2001] metamodel is used to represent software design since it is widely accepted in research and industry communities. In general, the UML metamodel consists of three packages i.e. Foundation, Behavioral Elements and Model Management. These packages define various useful models for the understanding and specification of the system under development. The Foundation and Behavioural Elements packages are the language infrastructures that correspondingly specify the static structure and dynamic behaviour of the UML models. For the purpose and scope of this research, only two models are included in the study namely class diagram and statechart diagram which respectively model the static and dynamic aspects of software systems.

3.4.1 Class Diagram

A static structure shows the kind of things that exist (such as classes), their internal structure, and their relationships to other things. The central concept in a static view is a class diagram. A class in the class diagram can be directly implemented in an object-oriented programming language that has direct support for the class construct. Figure 3-4 shows the appropriate subset from the UML metamodel, which contains all the necessary elements used in practice for the class diagram.

As shown in Figure 3-4, each UML model element is a subclass of the *ModelElement* abstract class. An abstract class is not allowed to have any objects and it is used merely to specify a common set of attributes and operations of its subclasses. Thus, the *name* attribute is used to identify each model element in a UML model. The *Class* class, which is inherited from an abstract class *Classifier*, has zero or more features. *Feature* is an abstract class which is inherited by *Attribute* and *Operation* classes. In other words, a class has zero or more attributes and/or operations. An attribute is a named property of a class that describes a range of values that the instances of the property may hold. It may also describe the multiplicity, visibility and type of its instance. An operation is the implementation of a service that can be requested from any object of a class to affect its behaviour. It is described with a return-type, a name, and zero or more parameters.

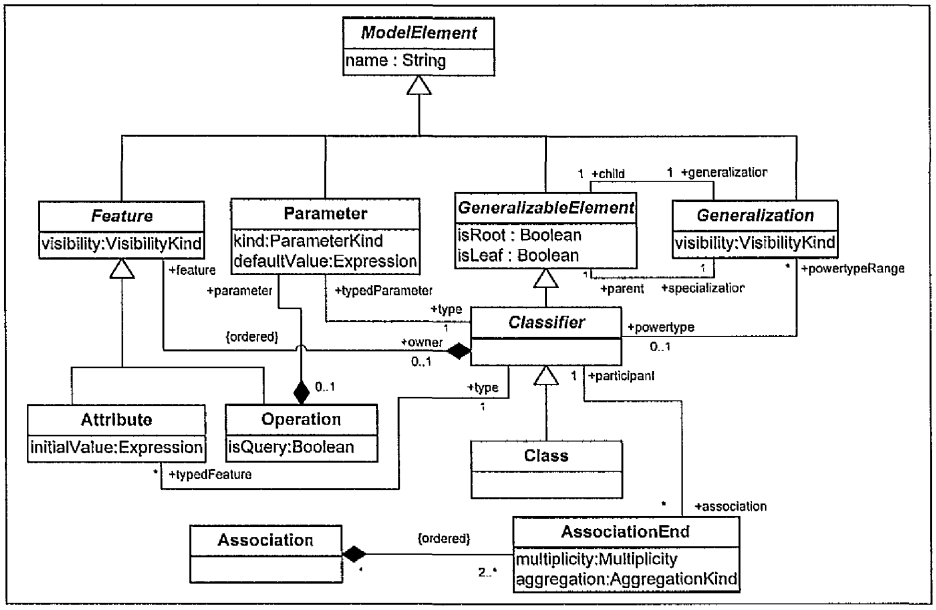

Figure 3-4 A fragment of the Class Diagram metamodel (an excerpt from the UML standard v1.4 [OMG, 2001])

The semantic relationship between classes is defined by an *Association* element. The instances of an association are a set of tuples relating objects of the classes. Each association has at least two *AssociationEnds*. An *AssociationEnd* is an endpoint of an association, which connects the association to a class. The *aggregation* attribute of *AssociationEnd* defines the kind of relationship none, aggregate, or composite. Apart from *Association*, there is also *Generalization* relationship that connects a class to its super-class so that the class inherit all of the features from its super-class.

3.4.2 Statechart Diagram

Statechart diagram is used to model the possible finite states of an object or system as well as the transitions from one state to another. It specifies the sequence of states, the event that cause the transition, the guard (condition) that should be satisfied, and the action triggered by the transition. It is useful for representing and understanding of the object with a complex behaviour.

The syntax and semantics of Statechart diagram is described in the State Machines package, which is a sub-package of Behavioral Elements package. In State Machines package, *State* is a static situation such as waiting for an event, or a dynamic condition

such as performing a process. It has one or more outgoing and incoming *Transitions*. *Transition* shows the change from the first state to the second state when the specified *Event* occurred and the *Guard* satisfied. It also specifies the *Action* triggered by the state change. The UML model elements related to statechart diagram is shown in Figure 3-5.

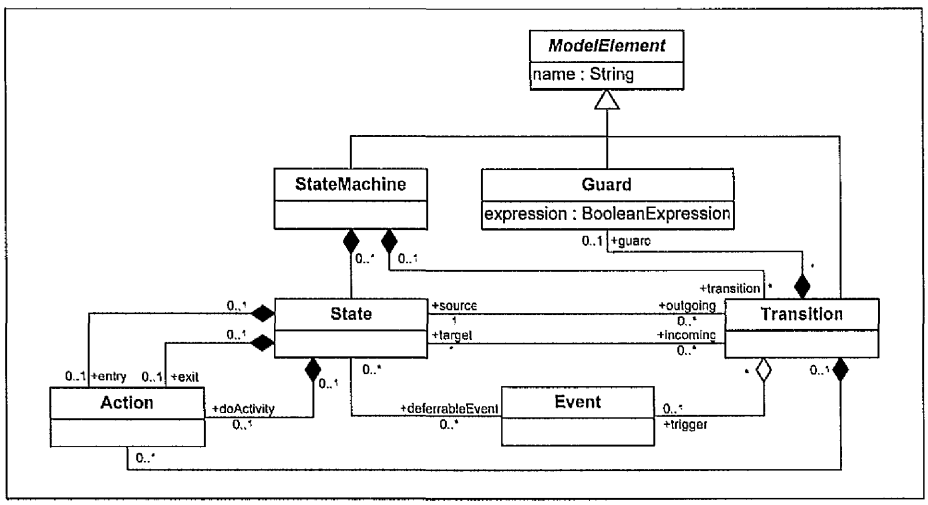

Figure 3-5 State Machine metamodel (excerpt from the UML standard v1.4 [OMG, 2001])

3.5 Rule Phrases and Linking Elements

Business rules can be implemented or linked to software system in many ways such as using rule object, rule engine, UML metamodel extension, or OCL constraints. However, the use of rule object or rule engine increases the execution time overhead during business rule trigger since the rule object or rule engine is located at different address space with the triggering software components. The extension of the well-known UML metamodel should be minimized because it will introduce new modelling elements that need new semantic definitions, which makes understanding of the design model more difficult. The use of constraints expressed using OCL may provide a link between business rule specifications and software design but OCL is still hard to understand by business users although OMG claimed that no mathematical background is required in using OCL. Due to these problems, BROOD introduces rule phrases that link the user-oriented business rule specification to software design.

Rule phrases are considered as the building blocks for the rule statements. They can be maintained independently during implementation, in other words, they are not deleted when a business rule is deleted. However, the modification and deleting of a rule phrase is not recommended since a careful effort is needed in reviewing its aggregated business rules.

In addition to playing a role as the building blocks for business rule statements, rule phrases are also important in linking business rules to software design elements. As shown in the metamodel in Figure 3-2, each rule phrase is linked to zero or more UML model elements. The mappings between rule phrase types and UML model elements are summarized in Table 3-2.

Table 3-2 The associations between rule phrases and design elements

Rule Phrase Type	Software Design Elements
Entity	Class
Attribute Term	Attribute
Operation Term	Operation
Attribute Constraints	Attribute.isUnique, Attribute.notNull
Cardinality	AssociationEnd.multiplicity
Role	AssociationEnd.role
Event	Transition.event → Class.operation
Condition	Transition.guard, Operation.specification
Action	Transition.action → Class.operation
Algorithm	Operation.specification
Value	- (literal value), Operation.
List	- (enumeration), Operation
Relational Operator	- (enumeration)

Most of the rule phrases are directly linked to class diagram model elements. Entity and Attribute Term are directly connected to the respective class and attribute in the class diagram. Cardinality and Role are correspondingly linked to multiplicity and role of an association end of a relationship. Algorithm, which specifies the details of a calculation, is linked to operation specification.

Class diagram is the most important representation of the software implementation components. For example, the information from class diagram may be used for code generation or directly mapped to the constructs of a modern object-oriented programming language. Therefore, it is important to maintain a link between business rule statements and class diagram model elements in order to improve business rule traceability in the actual software components. However, not all rule phrases can be semantically linked to class diagram since it only concerns with the static aspects of a software system. These rule phrases are Event, Condition, and Action, which are the building blocks for action assertion rules. They are naturally linked to statechart diagram since both action assertion and statechart diagram are concerned with the dynamic aspects of a software system. Event, Condition, and Action phrases in action assertion rule are respectively linked to event, guard, and action of a state transition in a statechart diagram. Consequently, event and action may be linked to a class operation, and guard may be linked to an operation specification, in a class diagram.

All of the above rule phrase type must be connected to software design elements. On the contrary, the remaining rule phrase types, i.e. List, Relational Operator, and Value, are not necessarily connected to any UML model elements. List and Relational Operator contain enumerated values whilst Value contains a literal value. However, Value and List can be linked to an operation that return a single and multiple values respectively.

3.6 Summary

The inherent problem of a software system in a dynamic business environment, the recent views on software evolution, and the gaps in business rule approaches to software evolution motivate the introduction of the BROOD approach. BROOD was proposed to tackle the root of software evolution problem by considering business rules as the important components of a software system. It provides the traceability that allows the propagation of business rule changes to software design.

At the heart of BROOD is the metamodel that offers a complete foundation and infrastructure for the development of a software system that is resilient to business rule changes. The metamodel includes the exhaustive rule typology and templates for guiding the process of developing a business rule specification. The widely-accepted

UML metamodel was chosen to represent the software design part of the metamodel. The rule phrases are defined to link business rules to the respective software design elements in order to minimize the extension or modification of the standard UML metamodel in implementing business rules.

The metamodel is capable to enhance the traceability of business rules from the conceptual domain to the design elements of the software domain. If this work is combined with the work on design traceability in source code [Alves-Foss et al., 2002], it would then be possible to achieve total traceability in a software system. The metamodel also contains some elements to facilitate the rule management or maintenance. To further enrich the implementation potential, the metamodel is also supported by EBNF definitions that describe the detailed definitions of the modelling language. Using this metamodel, the evolution of a software system can be automatically driven by the changes in business rule specification.

Although the metamodel and language specifications described in this chapter are considered as the fundamental part of BROOD approach, the approach is not considered complete without the descriptions on process, tools, and application. Therefore, the discussions on the BROOD process, its supported tool, and its application examples will be presented in the subsequent chapters.

Chapter 4

The BROOD Process

This chapter presents the second component of BROOD approach i.e. the software process. It starts with a brief description of SPEM modelling concepts and notations, which is used to model the BROOD process. This is followed by a high level description of the BROOD process in terms of its process metamodel, lifecycle phases, and use-case view of the process roles and main activities. Next, the detailed descriptions on analysis, design, and evolution phases are explained using the activity diagrams. Finally, the chapter summary is presented at the end of this chapter.

4.1 Introduction

As mentioned in the previous chapter, BROOD provides both the required components of a practicable software evolution approach i.e. product and process. The metamodel that represents the product component of the BROOD approach was presented in the previous chapter. The software process that supports the BROOD approach will be discussed in this chapter. In general, a software process is a set of activities that leads to the production and guides the evolution of a software product. It is important to have a detailed software process description as a guideline to validate the correctness or completeness of a process as the process becomes complex. Having a correct process model is like having an engineering blueprint, against which the activities can be designed and validated. A good process model may also facilitate the management of software project, improve communication between team members, and potentially shorten time to market of the software product. The BROOD process is described using the process model based on the syntax and semantics of the OMG Software Process Engineering Metamodel (SPEM). This chapter describes the SPEM modelling concepts and notations, the high level description of BROOD process, and the detailed description of the phases of the BROOD process.

4.2 Software Process Engineering Metamodel

Software Process Engineering Metamodel (SPEM) is developed by the Object Management Group to provide a metamodel and notations for specifying software processes and their components [OMG, 2002]. The original purpose of SPEM is to support the definition of a concrete software development process that suit organization's changing needs and environment based on the fact that the software process is also constantly change and evolve similar to software product. It also aims to provide a unified standard for process modelling techniques since there are many process models and standards which uses different terminology for the same meaning, or sometimes a different meaning for the same word or phrase. SPEM extends the Unified Modeling Language (UML) [OMG, 2001] metamodel with process specific stereotypes. A portion of SPEM that shows most of the important components of a process structure is shown in Figure 4-1.

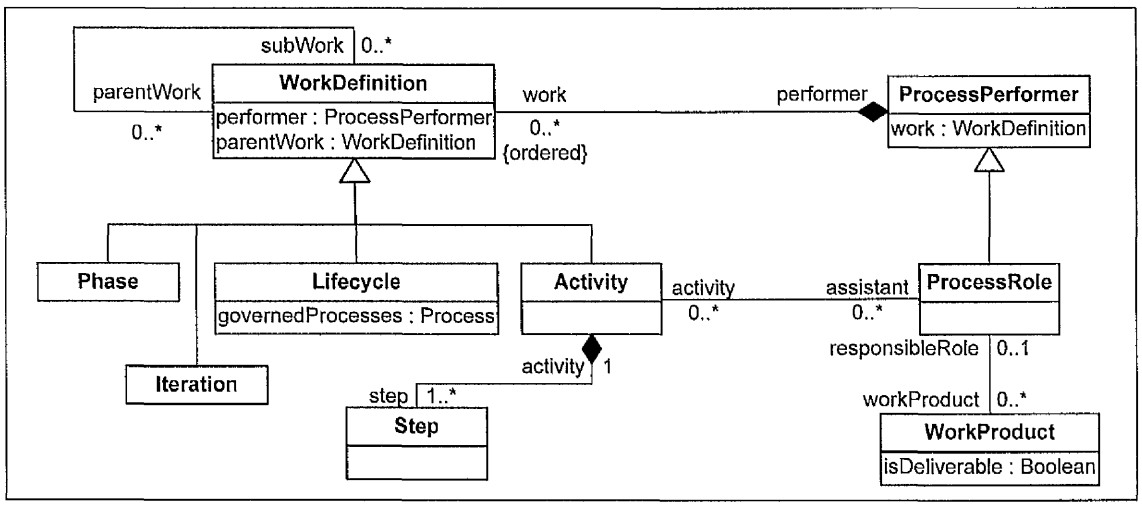

Figure 4-1 An excerpt from OMG Software Process Engineering Metamodel [OMG, 2002]

In SPEM, a *work product* is an artefact produced, consumed, or modified by a process. It may be a piece of information, a document, model, or source code. It is either used as an input by workers to perform an activity, or a result or an output of such activities. A work product is called a deliverable if it is needed to be formally delivered by a process. The examples of work products in BROOD are class diagram, statechart diagram, and business rule specification. Each work product is associated with a process role who is formally responsible for its production.

A *process role* defines the responsibilities of an individual, or a group of individuals working together as a team. Each process role may have a responsibility on a specific work product and role in performing or assisting specific activities. For example, 'functional analyst' and 'business user' are two process roles that respectively play the roles of performing and assisting the production of an 'initial business rule specification' work product. One individual may participate in more than one process role. Each process role is associated with zero or more performed or assisted activities.

An *activity* of a specific process role is a unit of work that an individual in that role may be asked to perform. The activity has a clear purpose, usually expressed in terms of creating or updating some work products. It may consist of one or more atomic components called steps. Activity is the main subtype of work definition, which describes the work performed in the process. Other subtypes of work definition include

lifecycle, *phase*, and *iteration*. An instance of work definition can be used to represent composite pieces of work that can be further decomposed. Figure 4-2 shows the notations based on SPEM that can be used in conjunction with use-case and activity diagrams to describe the above mentioned software process components. Please note that two additional components, i.e. *document* and *UML model*, are added to represent two sub-types of work product.

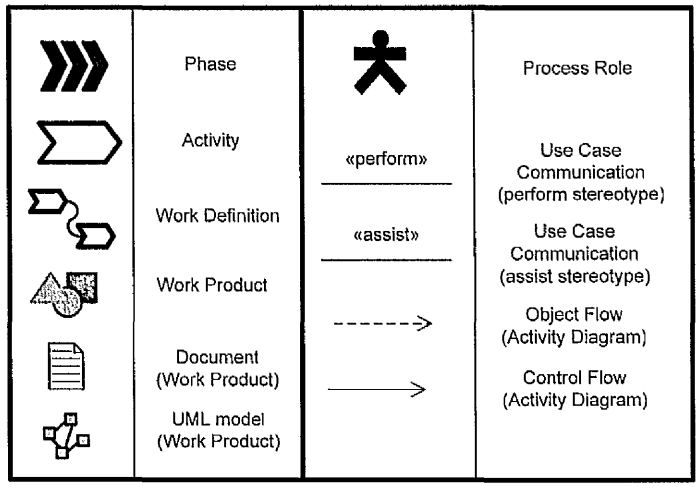

Figure 4-2 Notations used to describe software process

4.3 The Description of the High Level BROOD Process Components

Before going further into the description of the detailed process specification, the high level process components such as the main phases of the BROOD process in software lifecycle and the relationships between process roles and BROOD activities will be described in this section. It is important to mention that the BROOD process only defines the relevant phases in software lifecycle. It does not define any specific lifecycle model to allow flexibility of the BROOD process to be tailored for any type of software lifecycle model. The relationships between process roles and BROOD activities are described using use-case diagrams.

Some concepts in SPEM must be further clarified and adapted since SPEM is too abstract and largely influenced by the modern Rational Unified Process (RUP) lifecycle model. For this purpose, the metamodel that is used to describe the BROOD process components is developed; it is shown in Figure 4-3. The metamodel augments the

semantics of SPEM elements to produce more concrete definitions of work definition elements such as lifecycle, phase, and activity. In this metamodel, software lifecycle contains one or more phases. Each phase produces the tangible work products which are considered as deliverables. A management decision is often made at the end of each phase either to continue to the development of a project, or to drop it. Each phase contains one or more activities, which in turn contain one or more steps. Activity diagram is used to describe the flow of activities in certain phases whilst the detailed description of each phase such as work products, activities, and steps is described using structured process specification.

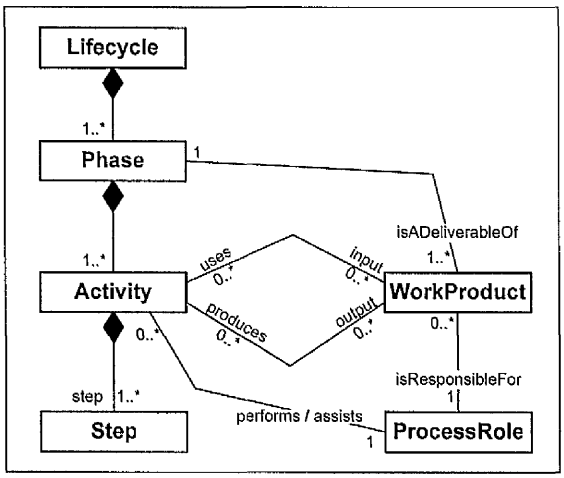

Figure 4-3 The BROOD Process metamodel

4.3.1 The BROOD Phases

The BROOD phases, which is shown in Figure 4-4, aims to give the high level view of a software development process which contains a sequence of possibly iterative phases and their respective activities. It represents a typical software lifecycle that consists of several main phases such as requirement, analysis, design, implementation, and evolution. Although the diagram shows the complete phases in the software lifecycle, the scopes of BROOD process are limited to analysis, design, and evolution phases. The requirements phase, implementation phase, and some activities in evolution phase are outside the scope of BROOD. However, they are briefly explained in order to position the BROOD process in the overall software lifecycle.

The exclusion of the requirement and implementation phases leads to two assumptions: the requirements model produced by requirements phase is always valid, and the presence of a link between software design and implementation. The former implies that the current research does not have to provide a detail treatment on requirements gathering and specification which certainly consumes longer time to complete the current research. The latter supports the claim of this research that the techniques of linking and propagating of business rule changes to software design may improve the evolvability of the implemented software system since there is a link between software design and its implementation.

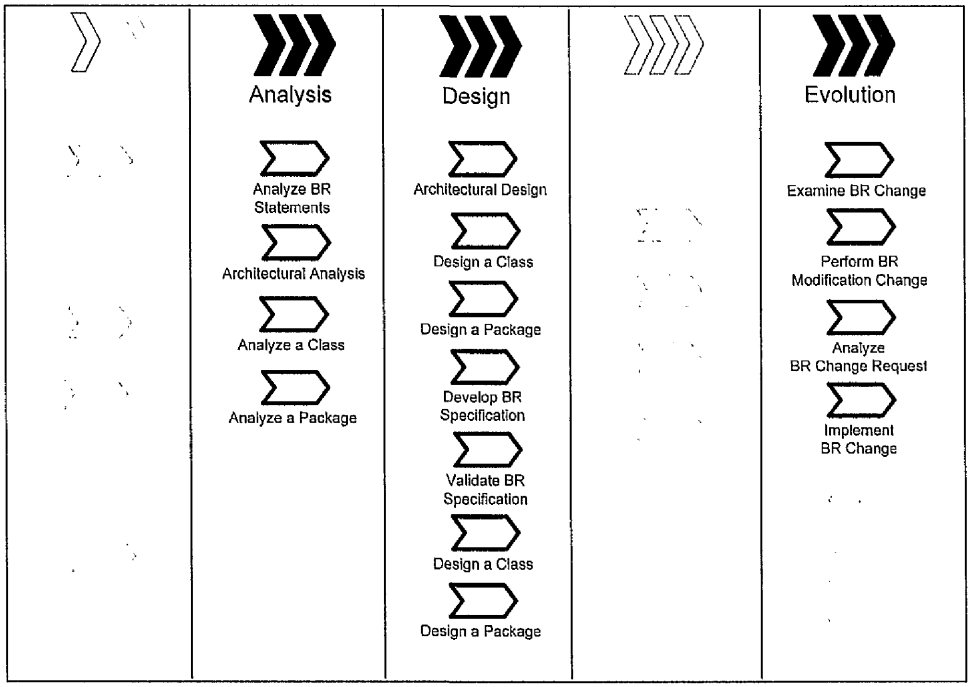

Figure 4-4 The main phases in the BROOD Process

The main purposes of the requirements phase are to understand the needs of business domain on the services provided by software under development and to define the scope of software application by the identification of system boundaries. This phase starts with gathering information from various sources such as interviews, questionnaire, observation, and existing system documentation. It involves a great deal of interaction with the people who will be affected by the system. In certain occasions, system developer may choose to develop business models such as business vision, process,

behaviour, or structure especially for a highly complex business domain. Apart from system understanding, these models also provide useful information for the identification of strategic, tactic, and operational business rules. These types of rules are high level definition of business rules which are outside the scope of this research. Both functional requirements, i.e. the statements of the services or functions the system should provide, and non-functional requirements, i.e. the constraints on the services or functions offered by the system, are captured during this phase. All of the captured requirements are validated at the end of this phase to ensure the completeness, unambiguity, and correctness. The activities in this phase are often iterated until the correct and reasonably complete requirements specification is produced. The requirements specification is often considered as contracts between software developer and clients. Other examples of work products created at the end of this phase include various business models, use-case model, domain model, and business vision documents. They are often collectively referred as requirements model.

During analysis, the requirements model developed in requirements phase is analyzed to achieve more precise understanding of the system. The requirements, including business rules, are transformed into various models or specifications, which are more structured or formal than requirements model. These models are used to detect and remove redundancies and inconsistencies of system requirements. In contrast to the user-oriented language used in requirements phase, the language used for communication in analysis is more technical. A set of work products produced by the activities in analysis phase is called analysis model, which includes class diagram, statechart diagram, package diagram, and business rule specification.

The analysis model is further refined in the design phase to produce the descriptions of the software's internal structure that will serve as a basis for its construction. Among the two main activities in this phase are the development of the detailed business rule specification and the creation of object-oriented software design. These two activities are closely related since there is a need to link business rule specification to software design elements to facilitate future software evolution. They are considered as the core software development activities in BROOD process. Among the work products produced at the end of design phase are the design (or detailed) version of class

diagram, statechart diagram, package diagram (sub-systems), and business rule specification.

The main purpose of the implementation phase is to translate the software design into machine readable form. Most of the contemporary CASE tools are capable to automatically generate code skeleton from the information in software design. The code skeleton is used by the programmers in their coding activity. The individual unit of the implemented software system is often tested prior to their integration. The system testing is done after each unit is integrated. The term unit is loosely used to represent a class, component, or subsystem.

The evolution phase is very crucial in determining the software survival in its business environment owing to the frequent and rapidly changes in business environment. Moreover, the cost of implementing software changes is often several times higher than its development cost. In general, the process in this phase starts with the analysis of change request initiated by business users, newly introduced technologies, or external environment. The types of changes are classified according to the purpose of changes such as to repair software errors, to adapt the software with different operating environment, and to response to organizational or business change. After the type of the required changes is identified, software architect will produce a change plan and pass it to component engineer for the implementation of the changes. As justified in the previous chapter, the scope of BROOD is to deal with the business changes, in particular the business rule changes.

As mentioned earlier, BROOD activities can be applied to any software lifecycle model with different arrangement of the BROOD phases. For example, the analysis, design, and evolution phases can be converted into workflows in RUP software process model [Jacobson et al., 1999]. The workflows may subsequently be fitted into inception, elaboration, construction, and transition phases with appropriate number of iterations. In SPEM terminology, the term workflow is called discipline.

4.3.2 Use-Case Diagram

Use-case diagram is used to show the relationships between process roles and the main activities in software process. There are two possible stereotypes of the relationships: *<<perform>>* and *<<assist>>*. The former indicates that an activity is owned by a process role that is the performer of the described activity. The latter refers to additional process role that is the assistant in the activity. Three use-case diagrams were developed for analysis, design, and evolution phases in BROOD process. They are shown in Figure 4-5.

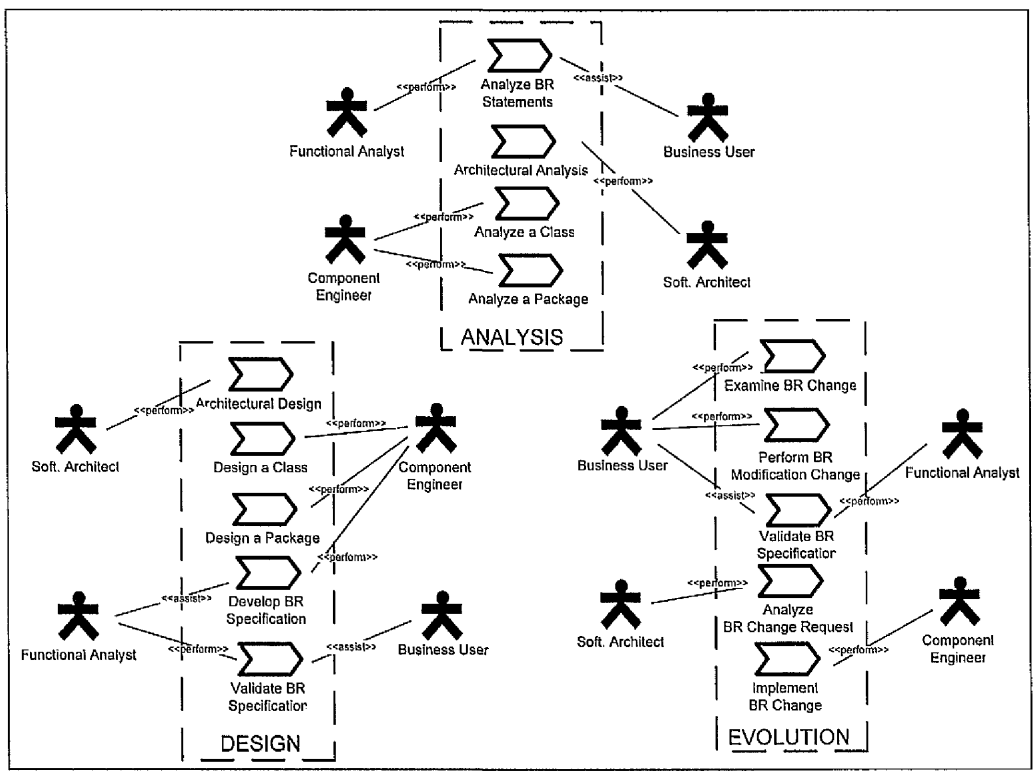

Figure 4-5 Use-Case Diagram for BROOD Process

As shown in Figure 4-5, there are four process roles that involve in BROOD process i.e. functional analyst, business user, software architect, and component engineer. Each process role is responsible for the work products produced by his activities.

Functional analyst involves in the activities related to requirements elicitation, analysis, and validation. He is responsible to ensure that the system fulfil the user requirements. With regard to business rule modelling, functional analyst analyzes the initial business

rule statements during analysis phase. This activity produces the initial business rule specification that contains more structured rule statements. Functional analyst also validates the developed or changed business rule specification during design and evolution phases. In a team of software developers, functional analyst is the most familiar individual about domain or business knowledge.

Business user may be an individual who use the software system, perform some business processes, or anybody who is directly responsible in the operation of the system under development. During analysis and design, business user only assist functional analyst in analyzing and validating business rule specification. However, business user plays important roles during evolution phase. He is responsible to examine the initial business rule change request from the business policy maker. If the change is classified as simple change, he will perform the change himself. On the other hand, if the change is complex, he will refine the initial change request and pass it to software architect.

The remaining process roles, i.e. software architect and component engineer, are mainly responsible with the activities and work products related to software system. During analysis, software architect performs architectural analysis activity that identifies the main classes and packages. Component engineer further analyzes the outlines of classes and packages to develop the analysis versions of class and package diagram. During design, they perform the activities that transform the analysis model into design model. With reference to business rule modelling, component engineer performs the development of business rule specification activity. Since this activity requires the creation of rule phrases, which are needed to be linked to software design components, the person with the detail knowledge on software design components is needed to perform this activity. Functional analyst assists the development of business rule specification using his business knowledge. During evolution, software architect analyze the business rule change request to produce a more detailed change request document. The document is passed to component engineer for the implementation of the requested change.

4.4 The Specification of BROOD Process

The core activities of BROOD process are resided in analysis, design, and evolution phases. Analysis phase produces analysis model that contains two main work products: the initial business rule specification and preliminary software design models. Both work products are refined and linked during design phase to produce a more traceable and consequently evolvable software system. The detailed activities on how to perform the analysis and design based on BROOD approach, as well as to implement the business rule changes during evolution, will be described in this section. Activity diagrams were used to show the flows of activities and their associated work products. The detailed structured specification of these activities is included in Appendix C.

4.4.1 Analysis Phase

The activities in the analysis phase aim to produce the analysis model that possesses more expressive power and formalism than the requirements model. It uses a more formal language to represent the details of system requirements. Analysis model is structured in a way that facilitates the process of understanding, using, and maintaining of the requirements. It also acts as an input to the design and implementation phases. However, solving problems and handling requirements that are better postponed to design phase should be avoided. The flow of activities during analysis phase is shown in Figure 4-6.

As shown in Figure 4-6, analysis phase starts with architectural analysis activity that analyzes the work products from requirements phase such as use-case model, business model, initial architecture descriptions, and supplementary requirements. Software architect performs architectural analysis by identifying the analysis packages based on the functional requirements and the knowledge of the application domain. Each package realizes a set of closely related use cases and business processes to minimise the coupling between packages, which in turn localizing business changes. Next, this activity identifies analysis classes and outlines their names, responsibilities, attributes, and relationships. In order to extract more information about the behaviour of the classes, collaboration or interaction diagram can be developed based on the process

flows (scenario) in the use case models. The main work products produced by this activity are analysis class diagrams and packages in their outline version.

The outline of analysis class diagrams and packages are further refined by class analysis and package analysis activities, respectively. Component engineer identifies more detailed information about responsibilities and attributes of each class. Different types of relationships between classes such as association, aggregation, and inheritance are also identified. The possible states and their transitions were identified to understand the behaviour of objects from certain classes. These steps are repeated until a complete analysis class diagram, statechart diagram and package are achieved.

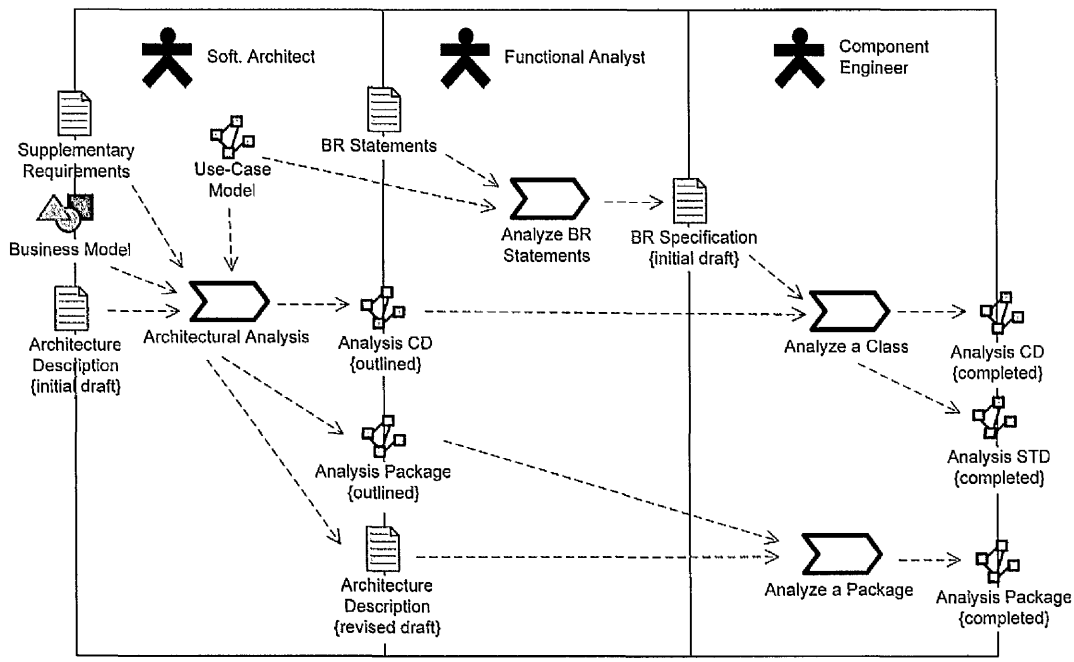

Figure 4-6 The flow of the analysis activities

With regard to business rule modelling, the business rule statements produced in the requirements phase are analyzed by a functional analyst during this phase. This activity starts with identifying the types for each business rule statement based on the typology proposed by BROOD. Next, the business rule statements are transformed into more structured business rule specifications according to the available templates of each type. The identified type and structured representation of business rules facilitate the next steps in this activity i.e. conflict resolution and redundancy elimination. At the end of

business rule statements analysis, the initial draft of business rule specification is produced. The specification is also useful for modelling of the class diagram in class analysis activity.

4.4.2 Design Phase

The goal of the design phase is to transform the analysis model to the design model that provides blueprint for implementation. Apart from functional requirements, the activities in this phase should acquire in-depth understanding of non-functional requirements and implementation constraints. They should produce a seamless abstraction for system's implementation, which permits straightforward refinement by completing the skeleton without changing the software structure. This abstraction permits the use of techniques like code generation and round-trip engineering between design and implementation.

The flow of activities in design phase is shown in Figure 4-7. It begins with the architectural design activity performed by software architect that consider the analysis model as an input. The first step in this activity is the identification of application-specific and application-general subsystems. The application-specific subsystems are related to packages that group a set of closely related services in application domain. They are mostly defined during analysis phase. The application-general subsystem is more related to implementation technology decisions such as the introduction of user interface and database connectivity layers. Then, the architectural significant design classes are identified. These classes include analysis classes, which represent the entities in application domain, and possibly active classes, which are possibly needed to satisfy the concurrency requirements. Finally, generic design mechanisms are identified to handle special requirements such as persistency, security, and transaction management. The design class diagram, package, and revised architecture description are produced at the end of this activity.

The class diagram is used by class design activity to further elaborate the static and dynamic information about the outlined classes. Additional information on the operations, attributes, and relationships are added to each class. The specification of operations and attributes is made using the syntax of the chosen programming language.

Regarding relationships, different types of relationship such as association, aggregation, and inheritance are included to provide more maintainable abstractions. If necessary, the methods that specify the algorithm for the implementation of operations are specified. The statechart diagrams are developed for certain classes to provide more understanding on the events and conditions that trigger their transition from one state to another. For traceability, an event or action in statechart diagram is linked to its respective operation of a class in a class diagram. Similar to class diagram, sub-systems are also designed in terms their dependencies and interfaces.

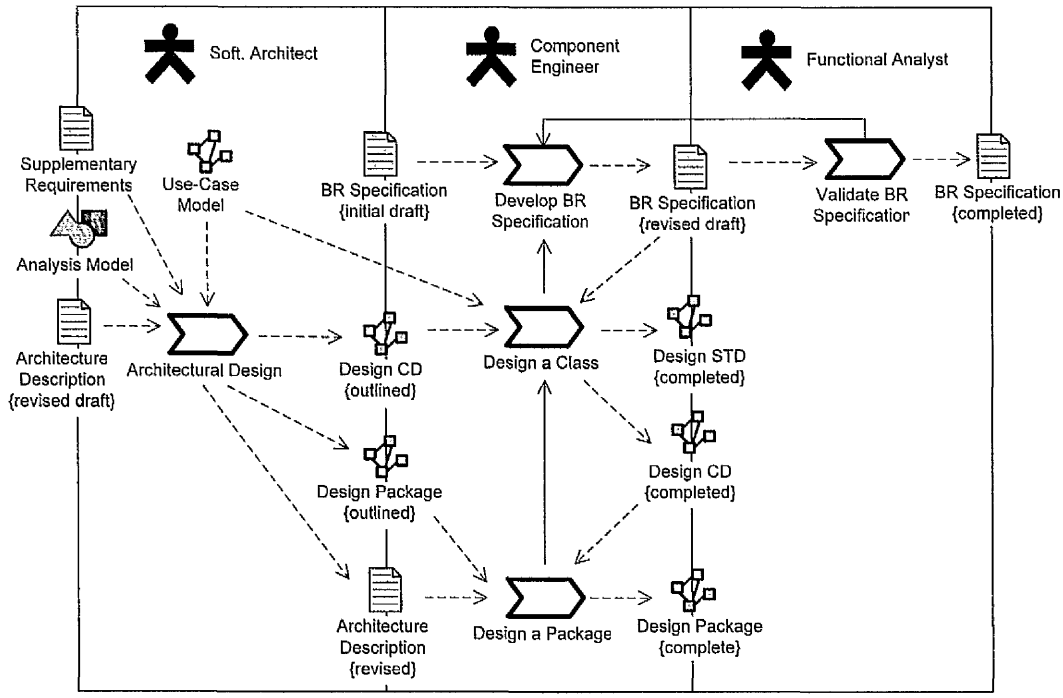

Figure 4-7 The flow of the design activities

In terms of business rule modelling, component engineer is responsible to develop business rule specification based on its initial specification produced at the end of analysis phase. The rule phrases should be firstly developed since they act as the building blocks of the business rule statements in a business rule specification. Each rule phrase definition is stored in the repository called rule phrase entries. The possible values for rule phrase may be a set of enumerated values or the values of the linked software design element. The rules to determine the properties of rule phrases are described in BROOD metamodel. Having populated the rule phrases, the component

engineer is now ready to compose the business rule specification. Component engineer may define certain attributes for each business rule specification such as rule priority, owner, and business process. Each business rule statement can also be arranged in an appropriate rule set to assist the future management of the business rules.

Finally, each business rule change is validated by functional analyst or possibly business user. The validation activity aims to ensure the correctness, understandability, and consistency of the newly introduced or modified business rules.

4.4.3 Evolution Phase

Evolution phase deals with the implementation of different types of changes. As mentioned in section 4.3, BROOD process is only concerned with the business rule evolution or rather, the changes of software design caused by business rule changes.

In a dynamic business environment, software evolution is unavoidable due to the continuously, frequently changed business rules. The initial business rule change request is often generated by business management team in a high level description since they have little knowledge on the detailed specification of business rules. The initial business rule change request is examined by a business user, i.e. the application user who is responsible to manage the business rule specification during software operation. The first step in examining the initial change request is the determination of the type of business rule change.

In general, business rule changes may be classified into simple and complex changes with regard to the required efforts for their implementation. Simple change is concerned with the modification, addition, or deletion of business rules that do not need to introduce new rule phrases or design elements. On the contrary, complex change involves the addition or deletion of rule phrases or design elements. Simple change is more frequently happen, easier to be automated, and often performed by business users. In contrast, the implementation of complex change is often very hard to be fully automated; it needs to be intervened by the technical skill and creativity of a person with software engineering knowledge such as software architect or component engineer. If the change request was identified as a simple change, the business user will perform

the change himself. Otherwise, he will produce a relatively detailed business rule change request to be passed to software architect for further actions.

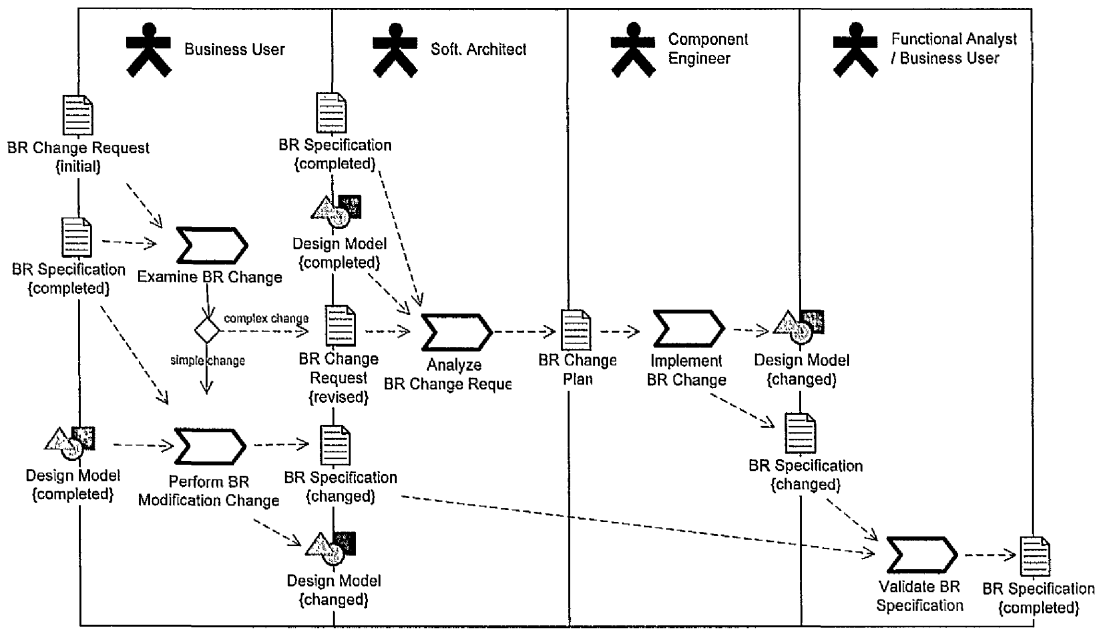

Figure 4-8 The flow of evolution activities

In the case of simple change, business user starts performing business rule change by locating the relevant business rule specifications. The appropriate rule organization such as arranging business rules in their respective rule set according to business processes makes it easy to locate business rules. Having located the relevant business rules, the business user may change the value of rule phrases. In the case of adding a new business rule, the user may compose the business rule statement using the available rule phrases in the rule phrase entries. The committed change is automatically propagated to the linked design elements.

For a complex change, larger efforts are needed since it involves the addition or deletion of certain design elements or rule phrases. The revised business rule change request is analyzed by software architect to produce a change plan. The change plan contains the detailed information about the effect of the proposed change to both the existing business rule specification and software design. Component engineer uses the change plan to implement the changes. Similar to the design process, the changed business rule specification is validated by functional analyst or business user.

4.5 Summary

This chapter describes the BROOD process using UML-based SPEM metamodel, which provides a set of concepts and notations to describe various software process components such as lifecycle phases, activities, process roles, and work products. The scope of the BROOD process is limited to analysis, design, and evolution phases with the emphasis on the role of business rules in producing an evolvable software system and reducing evolution efforts.

During analysis, the requirements model is refined using a more structured or formal language. The initial or informal business rule statements are transformed to the initial business rule specification using the guidelines provided by the BROOD metamodel and templates. The purpose of the analysis phase is to further understand the user and system requirements by producing a set of analysis model using the developer's language. During design phase, the architectural and detailed designs of the software system are developed based on the information from the analysis model. The initial business rule specification is refined and linked to software design. Throughout its operation, the delivered software application must undergo evolution phase due to the unavoidable changes in its business environment. BROOD provides a set of activities to deal with business rule changes including examining the initial change request, performing simple business rule changes, planning for complex changes, and implementing complex changes.

Chapter 5

Using BROOD in an Industrial-Strength Application

The purpose of this chapter is to provide an example of using the BROOD approach in an industrial-strength application. This chapter starts with a brief overview of the chosen application i.e. MediNET system. It is followed by the explanation on analyzing business rule statements and the overview of the produced MediNET classes and packages. The description of MediNET software design and the discussion on linking business rule statements to the software design are made in the subsequent section. Next, the explanation on how to implement the simple and complex changes based on the selected scenarios is presented in the following section. Finally, the summary and discussion on the lessons learnt from the BROOD application are made at the end of this chapter. It is important to mention that the discussion on this chapter emphasizes on the role of business rules in the development and evolution of MediNET. The detailed discussion on common object-oriented software design techniques are only discussed as necessary.

5.1 MediNET Overview

MediNET, which is formerly known as WebCare [Wan Kadir et al., 2000], was chosen as the case study due to its flexible nature for use by various businesses with different business rules. It was also chosen because of the availability of the detailed system description and the author's experience in the development and deployment of its initial version. Throughout this research, MediNET was also repeatedly used to experiment and consequently improve the proposed BROOD product and process discussed in Chapter 3 and 4 respectively.

MediNET is an internet-based application that allows various components of the healthcare industry to exchange business data instantaneously and automates their routine administrative tasks. In general, MediNET users can be divided into three categories: *paymasters*, *healthcare providers*, and a *supplier*. Paymasters are those who pay for medical services. They use MediNET to maintain the basic parts of the patient records. Healthcare providers (HCPs) are the professionals who dispense medical treatment. HCPs perform patient records management, patient billing and paymaster invoicing. The supplier is the company that owns and maintains the MediNET applications. The supplier lets users to access MediNET applications and charges them based on the number of performed transactions.

In MediNET, there are three main business processes: patient registration, billing, and invoicing. Patient registration can be done by the HCP or the paymaster. For each visit to HCP, each patient must be registered for consultation. The consultation registration is used to verify the patient eligibility and to prepare necessary information prior to the consultation. Next, the verified patient is put in a queue. After consultation is completed, a bill is issued to the patient based on the doctor's prescriptions. Cash patients must pay their bills whilst the bills for panel patients are sorted and verified before they are inserted into an invoice as invoice items. Finally, the invoice is sent to the paymaster. The more detailed descriptions of MediNET business processes, entities, and rules can be found in Appendix A.

5.2 Analysis Phase

In the analysis phase, the initial set of MediNET business rule statements were extracted from the available documentation of the current system such as the textual system description and system models. The examples of the identified business rules are listed at the end of Appendix A. The business rule statements were further analyzed and refined to produce the initial business rules specification. The specification consists of more structured business rule statements which were written in accordance with the available rule templates. It may be used in the identification of MediNET packages and classes. In this section, the overview of the main activities in the analysis phase, i.e. the analysis of business rule statements, packages, and classes, are discussed by providing the examples of their work products.

5.2.1 Business Rule Statements Analysis

Prior to linking business rules to software design, the initial set of informal rule statements must be transformed into more structured business rule statements. The rule type for each business rule statement must be identified using BROOD typology to simplify the selection of the relevant template. As a suitable template is selected, the statements can be rewritten according to the selected template. During this stage, some rules may be added or modified to produce a more precise and complete business rules specification. Some examples of the MediNET business rule statements, which were refined using the BROOD templates, are shown in Table 5-1.

The organization of business rule statements into business processes and the identification of rule type of each rule statement are important in producing the initial business rule specification. It can be observed from the above examples that business rules are more organized when they are arranged in a set of interrelated rules (ruleset) according to the business processes. Each ruleset may be further divided into the smaller rulesets to reduce the complexity in rule management.

The identification of the rule type allows the right template is chosen in writing a business rule statement. It is important for the rules to be specified in the templates for future specification and implementation. The rule statements which are written according to templates may also eliminate the overlapping or missing rules. However,

the use of a specific rule phrases such as 'zero or more' and 'is greater than' is not necessary at this stage as the statements will be refined with such rule phrases during the design phase.

Table 5-1 The examples of business rule statements in the MediNET initial business rule specification

Business Process	Business Rule Example	Rule Type
Registration	A patient must have a unique registration number.	Att. Constraint
	A patient may have more than one paymaster.	Rel. Constraint
	If a patient has an outstanding balance, then the patient should be banned from consultation registration	Action Assertion
	When consultation registration is successfully completed, then put the patient into the consultation queue.	Action Assertion
	If a patient's condition is critical then the patient is an emergency patient.	Inference
Billing	The amount of a panel patient's bill must not exceed the maximum bill amount set by the paymaster.	Att. Constraint
	Each bill item is associated with an item from the clinic transaction items	Rel. Constraint
	When consultation is completed then create bill.	Action Assertion
	If the bill is a panel patient's bill then create panel transaction item.	Action Assertion
	The amount of a bill is computed as the sum of all amounts of bill items.	Computation
	The amount of bill item is computed as the unit amount multiply by the quantity.	Computation
	A bill can be modified only if the user role is Chief Clinic Assistant.	Inference
Invoicing	One or more invoices must have zero or more payments.	Rel. Constraint
	When a payment is not received within 30 days from the invoice date, then the first reminder will be sent.	Action Assertion
	The amount of HCP MediNET usage invoice is computed as the sum of monthly subscription fee plus transaction fees.	Computation
	A paymaster (panel company) is under probation if the paymaster has an invoice with category 1 past due and the current balance is more than RM 5,000.00.	Inference

5.2.2 Packages and Classes Analysis

During the architectural analysis activity, the MediNET classes were identified based on the description on business processes and business entities. From the identified MediNET classes, it was found that they can be naturally and logically organized into three packages according to the main MediNET business processes i.e. registration, billing, and invoicing. The registration package groups all classes related to patient registration such as `Patient`, `Paymaster`, `HCProvider`, `Clinic`, `User`, and `RegLocation`. Billing package contains classes related to billing and drugs inventory such as `Bill`, `BillPayment`, `Bill_Item`, `TransType`, `TransItem`, and `ExpenseItem`. Invoicing package

includes classes related to invoicing and invoice payment for example `Invoice`, `InvoiceItem`, `Payment`, and `PaymentAllocation`.

Each package was further analyzed to refine its classes and to define its dependencies to other packages. Some classes from the registration package are used by the billing package for example, `Patient` class is associated with the `Bill` class as the receiver of the issued bill. Invoicing package has the connections with some classes from both registration and billing packages for instance, the `Payment` class is associated with the `Paymaster` class from registration package and the `InvoiceItem` class is associated with the `Bill` class form the billing package. The MediNET packages and their dependencies are shown in Figure 5-1.

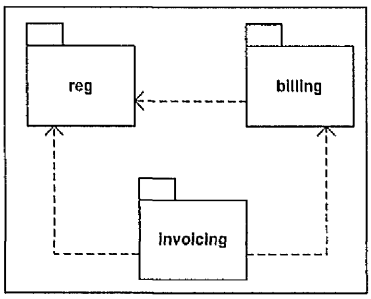

Figure 5-1 MediNET packages

Based on the experience in MediNET, there is no clear separation between the activities of package analysis and class analysis. They are closely related and frequently repeated in a software development process. It is also found that some types of business rules, such as attribute and relationship constraints, directly assisted the development of the class diagrams. Similarly, the action assertion rules supported the development of statechart diagram for the classes with complex object behaviour. At the end of the class analysis, three class diagrams were developed that respectively represents the above mentioned classes found in registration, billing, and invoicing packages. In addition, the statechart diagrams for invoice, bill, and paymaster were also developed. The detailed discussion on MediNET classes will be made in section 5.3.2 to avoid repetition of discussing the same classes since the same set of analysis classes were expanded during design phase.

5.3 Design Phase

During design phase, the analysis models such as package diagram, class diagrams, and statechart diagrams were refined with more detailed information. The initial business rules specification from the analysis phase was also refined for linking to the software design. Each business rule statement was studied to derive the rule phrases and consequently link to its respective software design element. If necessary, the statement can be rewritten to match the available templates, to derive more suitable rule phrases, or to achieve completeness and correctness of the business rules. However, this task must be carefully done to preserve the original meaning of the rules.

5.3.1 MediNET Sub-systems

During the architectural design activity, the package diagram produced at the end of analysis phase was refined with further design decisions. The first step in this activity is to decide the MediNET sub-systems. MediNET is divided into three subsystems based on their target users: *myPeople*, *myClinic*, and *myMediNET*. *myPeople* is used by paymasters to maintain their payee records. *myClinic* is used by HCPs to deal with their patient transaction, paymaster invoicing, and other clinic management activities. *myMediNET* is used by supplier to administer the MediNET application. Each subsystem is represented by its corresponding package and it is further divided into three layers: user interface, business objects and database layers. The MediNET software architecture is illustrated in Figure 5-2.

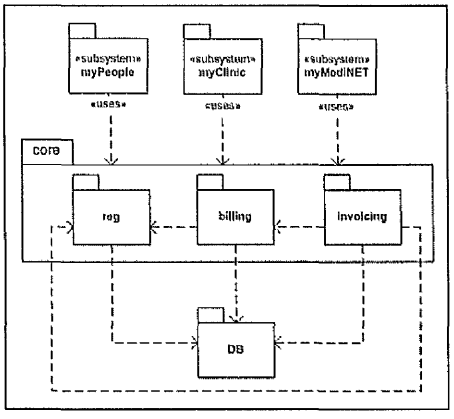

Figure 5-2 MediNET software architecture

5.3.2 Class Diagrams

Although there are many types of design classes in MediNET systems, such as user interface, control, and data access classes, this thesis will only focus on the classes that are directly related to the implementation of business rules in order to simplify the discussion. These classes are located at the business objects layer, which is represented by the *core* package. In the MediNET software architecture, the *core* package is commonly used by all subsystems. It consists of three sub-packages that already mentioned in section 5.2.2 i.e. registration, billing, and invoicing.

5.3.2.1 Registration Package

Registration package models the properties and relationships of persistent classes such as `Patient`, `Paymaster`, and `HCProvider` as well as transient classes such as `ConsultationQueue` and `PatientInQueue`. Most of these classes were identified from the detailed description on the patient registration process. Their attributes, operations, and relationships are shown by the `Registration` package class diagram in Figure 5-3. However, all class diagrams presented in this section hide the parameters and the return values of the class operations for the purpose of reducing the presentation complexity.

`Patient` class consist of the information on a patient such as patient registration number, personal details, and contact details. A patient can be a guardian of zero or more other patients. If a patient is a dependant of another patient, the guardian registration number must be defined. It allows both HCP and paymaster to trace the patient claims. `Patient` class has one subclass, i.e. `PanelPatient`, that represents a panel patient.

`Paymaster` class attributes include the paymaster ID, contact details, and status. The `status` attribute is an enumerated attribute which can be set to '*active*', '*blocked*', or '*archived*'. The value is set to '*blocked*' by HCP or MediNET System Administrator based on the defined business rules, which are related to the outstanding balance of the paymaster's invoices. The reason for being blocked is stored in the remark attribute. If the paymaster wishes to stop from being a MediNET customer, its status is changed to '*archived*'. The paymaster record is then deleted after certain archived period elapsed and the MediNET usage outstanding balance was settled by the paymaster.

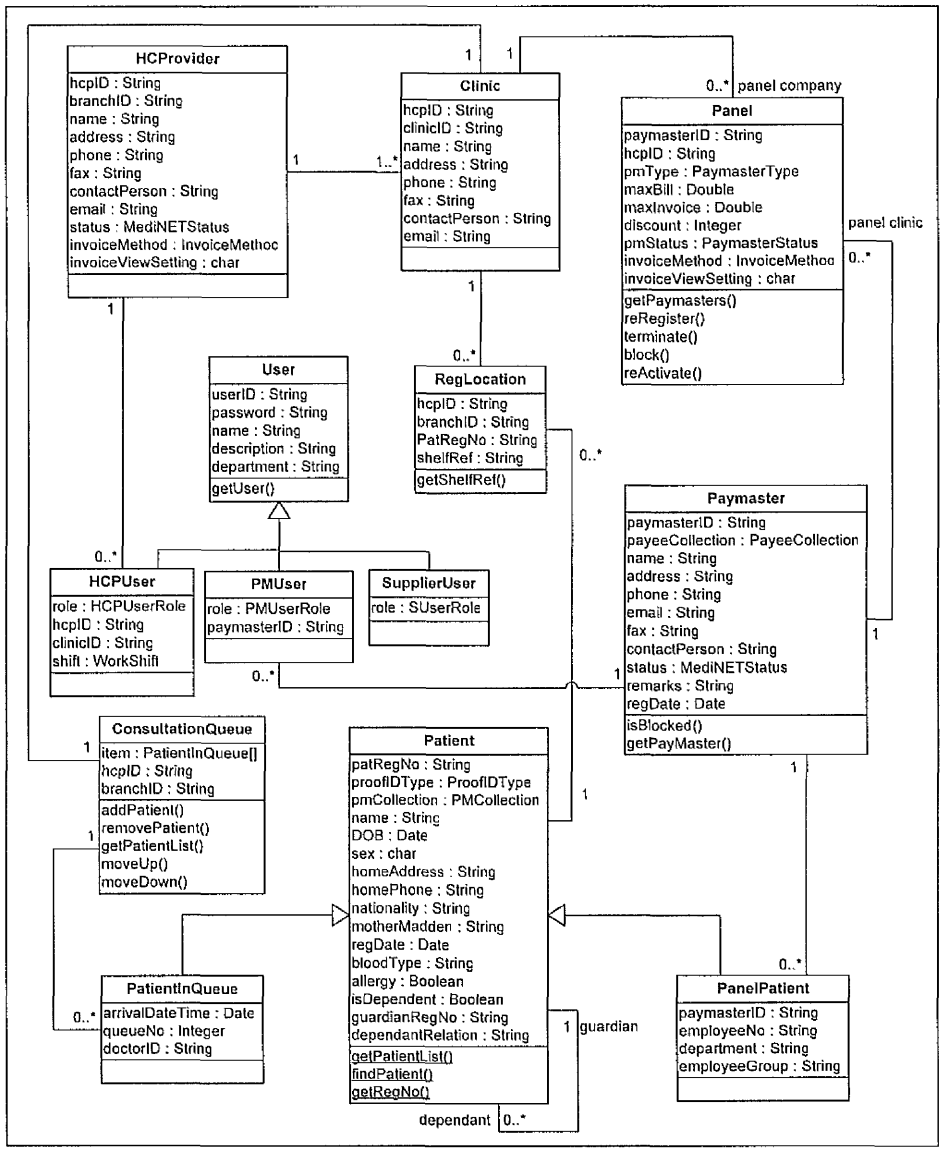

Figure 5-3 Registration package

`HCProvider` class contains the information on HCP such as `hcpID`, contact details, and status. The possible values and rules for the status attribute are similar to the above mentioned `Paymaster`'s status. The HCP status is only maintained by MediNET System Administrator.

`RegLocation` and `Panel` classes were introduced to simplify the problem of many-to-many relationships. `RegLocation` represents the relationship between patient and HCP. It is also used to store the shelf reference number of a patient record. `Panel` represents the relationship between HCP and paymaster. It consists of both `hcpID` and `paymasterID`

attributes which respectively link to HCProvider and Paymaster objects. The Panel class allows different paymaster account settings for different HCP. Paymaster can set the maximum bill and invoice to be paid, and HCP may set the discount and status for the paymaster. HCP may set the paymaster status to *'active'* or *'blocked'* according to its payment records and the defined business rules. The invoiceViewSetting attribute of the Panel class allows paymaster to determine which fields to be included in its printed invoice.

ConsultationQueue and PatientInQueue are two transient classes that deal with the implementation of a queue for patient consultation. ConsultationQueue is a queue type data collection that allows the insertion, removal, and reposition of its items. Its items are the objects instantiated from PatientInQueue class, which is an inheritance from the Patient class. PatientInQueue class has three additional attributes i.e. arrivalDateTime, queueNumber, and doctorID. As implied by their names, arrivalDateTime captures the patient arrival date and time information, queueNumber assigns a position in a queue to a patient, and doctorID is used for removing patient from the queue if the patient is allocated to a particular doctor.

5.3.2.2 Billing Package

In the Billing package, which is shown in Figure 5-4, HCProvider class from Registration package has several additional relationships. First, HCP may employ zero or more doctors. Each doctor is registered to a particular HCP and is assigned a unique ID. If the doctor is a part-timer (locum tenens) then isLocum attribute is set to 'true'. Otherwise, isLocum value is set to 'false'.

Second, HCP has zero or more transaction types. Transaction types allow the grouping of service and drug items provided by HCP. The examples of transaction types include consultation, medication, ward charges, and X-Ray. Each transaction type has zero or more transaction items. For example, consultation type possibly has doctor consultation, nurse consultation, and specialist consultation as its items. The consultation item is implemented as TransItem class. TransItem class plays an important role in patient billing. It contains general information about the bill item such as name, description, and measure unit. For bill item calculation purpose, the unit amount and service tax can

be set. The former is a current price for a unit of item whilst the latter is a service tax normally imposed by the government.

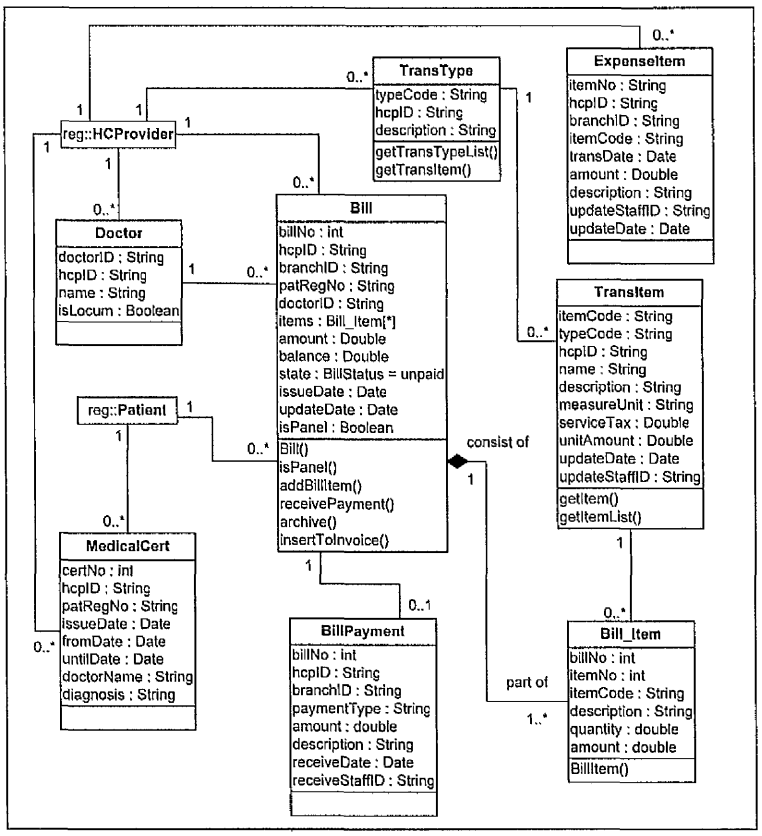

Figure 5-4 Billing package

Third, HCP may issue zero or more patient bills, which is represented by the `Bill` class. Each `Bill` object is uniquely identified by a combination of a bill number and an `hcpID`. The `Bill` class contains `patRegNo` and `doctorID` attributes, which are used to respectively identify the patient who receives the bill and the doctor who issue the prescriptions. The bill items are implemented as an array of `Bill_Item` objects. The `Bill_Item` object has an item number that is sequentially assigned when the item is inserted into the bill. It also has the `itemCode` attribute to reference to a `TransItem` object. The amount of each item is automatically calculated by multiplying the unit amount with the item's quantity. If the value of unit amount is not defined, `Bill_Item` class will allow a value for the bill item amount to be directly set without above calculation. The amount of bill is then calculated using the total of all bill item amounts.

`Bill` class also holds the bill status, which is an enumerated attribute which has three possible values: '*unpaid*', '*paid*', and '*invoiced*'. The bill status is initialized to '*unpaid*' during bill creation. It is changed to '*paid*' after a cash patient bill is settled and '*invoiced*' after a panel patient bill is inserted into invoice. `isPanel` attribute is used to check whether the bill is belonging to a panel or cash patient.

Fourth, if a patient is granted a medical certificate (MC), a new `MedicalCert` object will be created during billing process. The MC contains certificate number, originating HCP ID, and patient registration number. It also contains the start and end dates for the purpose of medical leave. The name of the doctor who issues the MC and the medical reason or diagnosis that leading to MC to be issued is also recorded in `MedicalCert` object.

5.3.2.3 Invoicing Package

In Invoicing package, the Invoice class contains three sub-types i.e. `HCServiceInvoice`, `PMMediNETUsageInvoice`, and `HCPMediNETUsageInvoice`. The invoice object only can be instantiated from one of these sub-types since the Invoice class is an abstract class. Each invoice object is uniquely identified by a combination of `invoiceNo` and `isuerID` attributes. The possible value for the `issuerID` attribute depends on the invoice subtype, for example the value of `issuerID` for `HCServiceInvoice` object is taken from `hcpID` of the HCP that issue the invoice. For `PMMediNETUsageInvoice` and `HCPMediNETUsageInvoice` objects, the value of `issuerID` is always set to 'MEDINET', which is a reserved ID value for MediNET supplier, since it is always issued by the supplier.

The amount of the invoice is stored in the `amount` attribute, which calculated using the abstract `calculateAmount` operation. The calculation of invoice amount is different for different types of invoices. For MediNET usage invoices, the amount is calculated according to the tables included in Appendix A. The amount for healthcare service invoice is calculated as the total of its item amounts after applying additional computation rules such as bill limit, invoice limit and discount. MediNET uses the open item invoicing method that allows an invoice issuer to track each unpaid invoice as an individual item for aging purposes.

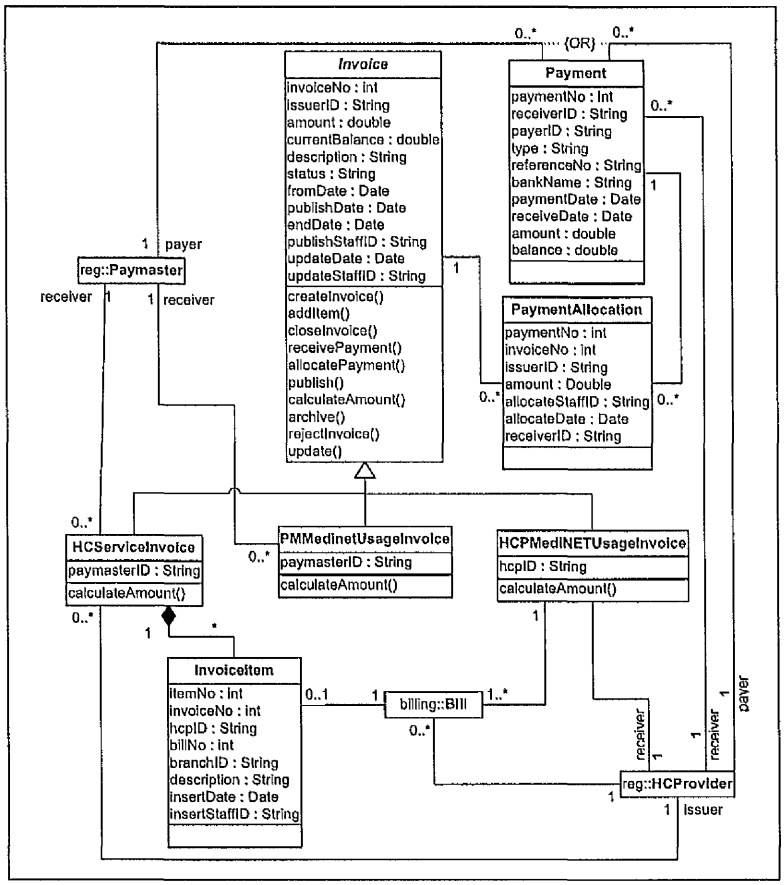

Figure 5-5 Invoicing package

The `state` attribute of the `Invoice` class stores the current state of the invoice. The `state` value can be set to one of the pre-defined states. The explanation on the possible states of an `HCServiceInvoice` object and their transitions are presented in section 5.3.3. The date and user information of the invoice publishing and modification is stored in `publishDate`, `publishStaffID`, `updateDate`, and `updateStaffID` attributes.

With regard to invoice items, panel patient bills are considered as the items for HCP MediNET usage and HCP service usage invoices. For HCP MediNET usage invoice, the number of bills issued by a particular HCP is counted as the number of transactions, which is later used in the invoice amount calculation. A relationship that shows one and only one `HCPMediNETUsageInvoice` object is associated with zero or more bills is enough since no additional information is needed for invoice calculation or maintenance. However, there is a requirement to explicitly introduce a class for the items of HCP

service invoice because additional information is needed to keep track the insertion of each item into the invoice. Thus, the `InvoiceItem` class, which contain the invoice reference, bill reference, update date, and update staff ID information, is used to represent the bill items of HCP service invoice. For paymaster MediNET usage invoice, the invoice item class is not needed since the invoice amount for is calculated based on the number of patients sponsored by the paymaster.

In terms of payment, the `Payment` class is included to represent each payment received for the issued invoice. The payment number and receiver ID are used to uniquely identify the payment object. The `payerID` attribute is used to identify the payer, which is either a paymaster or an HCP. Since MediNET also allows balance forward invoicing method in addition to open item method, zero or more payments must be allowed to be allocated to one or more invoices. This situation causes a problem of many-to-many relationship. The problem is solved by introducing the `PaymentAllocation` class to represent the allocation of each payment. The `PaymentAllocation` object can be allocated to one or more `Invoice` objects. The procedure to allocate the invoice is depending upon the chosen invoicing method. Each time a payment is allocated, the amount of payment allocation is deducted from the balance of the `Payment` object. Again, the calculation is depending on the chosen invoicing method.

5.3.3 Statechart Diagram (STD)

In practice, STD is not developed for every class in the class diagram. It is only developed to provide more understanding on certain classes with complex object behaviour. In BROOD, STD is used to improve both the understanding and maintainability of a software system. The system maintainability is improved by providing links between action assertion rules and STD. In MediNET, the classes such as `Paymaster`, `Bill`, and `Invoice` are important to be modeled using STD since their behaviours are associated with a number of action assertion rules. As an example, the statechart diagram that is developed for an `HCServiceInvoice` object is shown in Figure 5-6.

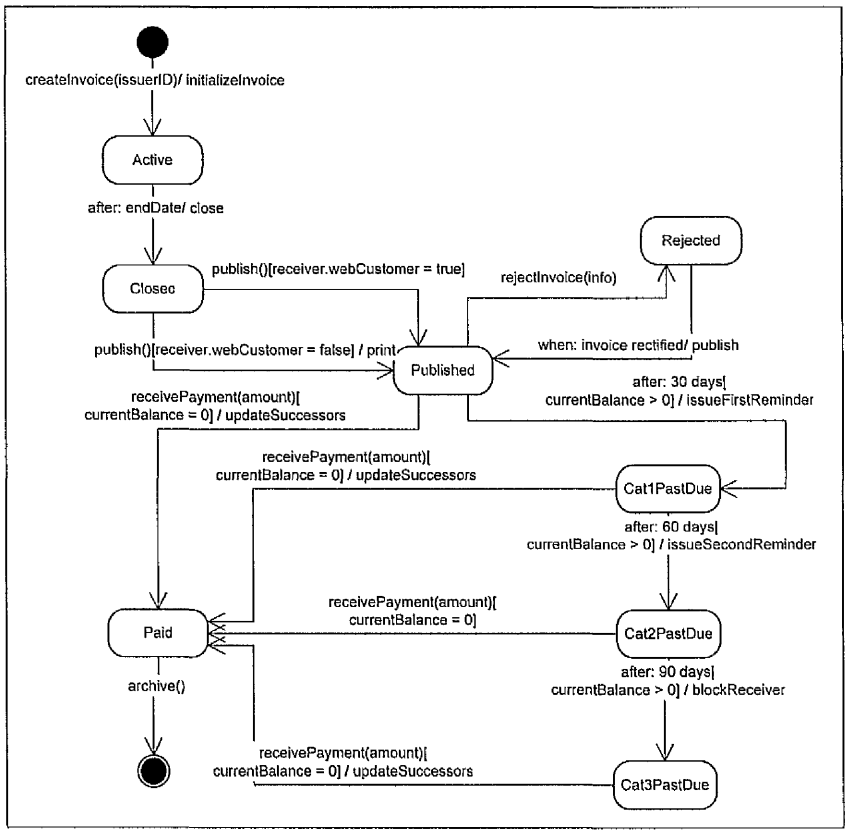

Figure 5-6 STD for `HCServiceInvoice` object

An invoice object will be created as it receives the `createInvoice(issuerID)` message as a triggering event. It consequently triggers the `initializeInvoice()` action and enter the `Active` state. During active state, the invoice items can be inserted into invoice until the invoice is closed. The invoice changes its state from `Active` to `Close` after its end date, which is specified during invoice initialization. After the invoice is printed or published on the internet, its state changed to `Published`. When the payment is received and the current balance value is equal to zero, the state is changed to `Paid`. When the payment is not received after 30 days and there is a current balance, then first reminder will be issued. If there is still no payment after 60 days, the invoice receiver will be issued a second reminder and consequently blocked from receiving services from the issuer after 90 days. The example of linking action assertion rule to STD will be presented in the next section.

In addition to the links with action assertion rules, it is also important to link an STD to a class diagram since a class diagram is the source of code generation. In the above example, most of the events and actions such as `createInvoice()`, `intializeInvoice()`, `publish()`, and `rejectInvoice()` are linked to their respective operations in the `HCServiceInvoice` class itself. Some events or actions such as `receivePayment()` and `blockReceiver()` are implemented as the operations in other classes. For example, `receivePayment()` is implemented in event class, which is often modelled using UML `<<signal>>` stereotype whilst `blockReceiver()` is implemented as an operation of the `Paymaster` class. With the latest advances in event-handling mechanisms, this research assumes that the link between STD and class diagram is seamlessly and easily established. Therefore, it only focuses on the links between action assertion rules and STD.

5.3.4 The Development of Business Rules Specification

The development of business rule specification is the most important activity in ensuring business rule traceability in software design. As its first step, each business rule statement in the MediNET initial business rule specification, which was produced at the end of the analysis phase, was further analyzed to identify the candidates for rule phrases. The available templates were used as the guidelines to identify the rule phrases and to determine the type of each rule phrase. The examples of rule phrases, which were derived from business rule statements in Table 5-1, are shown in Table 5-2.

The first example in Table 5-2 shows the rule phrases derived from the attribute constraint rule. The rule phrases 'a patient' and 'registration number' are directly respectively linked to `Patient` class and `patRegNo` attribute. The keywords 'must have' and 'a unique' are not statically linked to any design element. Instead, they are used to dynamically respectively toggle the optionality and uniqueness values of `patRegNo` attribute during the creation or modification of the business rule statement. In other words, they are used to enable the automated change propagation to software design.

In the second example, the rule phrases 'clinic item' and 'bill item' are respectively linked to `TransItem` class and `Bill_Item` class. The rule phrases 'one and only one' and 'clinic item' play a similar role to keywords in attribute constraint rule i.e. to propagate

business changes to design elements. The former specifies the multiplicity of an association end whilst the later specifies the role of an association end. From the first and second examples, it is found that constraint rules can be directly linked to class diagrams and their changes can be easily propagated.

It is also observed from the above example that the previous business rule statements is further refined with the more technical rule phrases or keywords such as 'one and only one', 'must have', and 'a unique'. Such phrases may improve the preciseness of the business rule statements. For this purpose, some rule phrase types such as cardinality, operator and several BROOD keywords are considered as the common phrases – they have their own pre-defined rule phrases that are unlikely to change. For other rule phrases, they are required to be created and defined by software engineers.

Table 5-2 The examples of the rule phrases and the linked software design elements

B Rule Category	Business Rule Phrases	Software Design Elements
Attribute Constraint	<entity> = 'a patient'	Patient (class)
	'must have'	- (patRegNo.optionality)
	'a unique'	- (patRegNo.uniqueness)
	<attributeTerm> = 'registration number'	Patient.patRegNo (attribute)
Relationship Constraint	<cardinality> = 'one and only one'	- (AssociationEnd.multiplicity)
	<entity> = 'transaction item'	TransItem (class)
	<role> = 'item type'	- (AssociationEnd.name)
	<entity> = 'bill item'	Bill_Item (class)
Action Assertion	<event> = '30 day after the creation date of the invoice'	- (Trans1.event.spec)
	<condition> = 'current balance of the invoice is greater than 0'	- (Trans1.guard.body)
	<action> = 'trigger issue the first reminder'	- (Trans1.action.initialiseInvoice().spec)
Computation	<attributeTerm> = 'the amount of HCP MediNET Usage invoice'	HCPMediNETUsageInvoice.amount
	<algorithm> = 'the sum of monthly subscription fee plus transaction fee'	HCPMediNETUsageInvoice. calculateAmount().specification
Inference	<attributeTerm> = 'a paymaster status'	Paymaster.status
	<value> = 'under probation'	- (literal value)
	<condition> = 'the paymaster has an invoice with category 1 past due' AND 'the current balance is greater than RM 5,000.00'	Paymaster.getStatus().specification

The third example shows the derived rule phrases from an action assertion rule. The rule phrases that represent the event, condition, and action are not directly linked to any design element but they are respectively used to generate the specifications of the

transition's event, guard, and action in the HCP service usage invoice STD. Since event, condition, and action rule phrases are themselves composed by other rule phrases, they may be indirectly linked to the related design components via these rule phrases.

The fourth and fifth examples illustrate the rule phrases derived from computation and inference rules. Both business rules from these examples are linked to the operation specification – the computation rule is linked to the specification of `calculateAmount()` operation in `HCPMediNETUsageInvoice` class and the inference rule is linked to `getStatus()` operation from `Paymaster` class. During the development of an inference rule, a new operation is often needed to be added in its associated class to perform the derivation and return the inferred value.

The defined rule phrases are stored in the MediNET rule phrase entries. They are later used to compose a traceable business rule statements. In order to reduce the number of rule phrases in the rule phrase entries, BROOD allows components engineer to rewrite the business rule statements to match with the existing rule phrases. However, this task should be carefully done without changing the meaning of the original business rules.

5.4 Evolution Phase

The ultimate aim of the BROOD approach is to effectively manage the implementation of business rule changes caused by the changes of business policy. These changes include the modification or deletion of existing business rules and the addition of a new business rule. They may be classified into simple and complex changes according to the efforts needed in the implementation of changes.

5.4.1 Simple Business Rule Change

Business rule change is considered as a simple change if it does not involve the introduction of new rule phrases or design elements. A simple change includes a modification of the existing business rule statements, an introduction of a new business rule using the existing rule phrases, and a deletion of business rule statements. It is often performed by business users.

Modification of the existing business rules was found to be the most frequent type of changes in MediNET. It is also the most simple since it does not involve the introduction of new business rules and rule phrases. These changes normally involve the change of the values in the rule phrases, and each change is often affects only a single business rule or relatively small number of related rules. The examples of five change scenarios that require simple business changes in MediNET system are shown in Table 5-3.

Table 5-3 The examples of change scenarios for simple business rule change

Change Scenarios	Changed Business Rules
1. HCP allows patients to make 'more than one payment for their bills' instead of the previously set 'single payment for each bill'.	One patient bill is associated with zero or more payments.
2. HCP makes small changes on the conditions to issue the reminder and block paymaster.	WHEN *15 days* from the invoice date IF a payment is not received THEN issue the first reminder. WHEN *30 days* from the invoice date IF the payment is not received THEN issue the second reminder. WHEN *45 days* from the invoice date IF the payment is not received THEN block the paymaster.
3. The MediNET supplier offers a more attractive usage charge to HCPs. They are charged based on the number of treated patients regardless the number of patient visits.	The amount of HCP usage invoice IS CALCULATED AS if (opt new package) then the transaction fee multiply by the number of registered patients, else, the transaction fee multiply by the number of treated patients, plus the monthly fee.
4. HCP introduces 5% discount to its internet customer.	If the paymaster is an internet customer, then give 5% discount to their invoices.
5. The HCP decides that each expense item must belong to one of the pre-defined types.	Zero or more expense item is associated with one and only one transaction item.

In the first scenario, the HCP would like to introduce a more flexible way for patients to pay their bills. Instead of a single payment, the HCP is willing to accept more than one payment from patient. Obviously, this change only involves a single business rule. The cardinality rule phrase of payment entity is changed from 'one and only one' to 'zero or more'. This change is easily propagated to the cardinality of the relationship between bill and payment. The propagation is also easily automated using an automated tool since it does not involve any creative human activity.

The second scenario leads to simple changes although it involves more than one business rules. As shown in the modified business rules, it only changes the number of

days (value changes) in the condition part of the existing business rule statements. The change is then propagated to its respective guard in STD diagram for invoice object; in particular the guard in the transition from 'Published' state to 'Cat1PastDue' state is changed from *30* days to *15* days. The similar changes are made on the guards of the transitions to 'Cat2PastDue' and 'Cat3PastDue' with the new value of 30 and 45 days respectively.

In the third scenario, the change in an algorithm to calculate the amount of HCP usage invoice is propagated to the specification of the `calculateInvoice` operation in the `HCPMediNETUsageInvoice` class. Similar change is also required by the fourth scenario.

The final scenario in Table 5-3 is different from the previous scenarios in that it requires the addition of a new business rule. Although it adds a new business rule, which consequently introduces a new relationship between `ExpenseItem` class and `TransItem` class, this type of change is considered as a simple change since there is no new rule phrase introduced and the change can be easily propagated possibly without creative human activity.

In certain occasions, the changes in business policy may require some business rules to be deleted. In this case, only the specific business rule statements are deleted from the business rules specification. Their aggregated rule phrases and related design elements are not deleted since they may also be used by other business rules.

5.4.2 Complex Business Rule Change

The implementation of complex business rule change requires more efforts than that of simple change. It involves the introduction of new rule phrases or design elements, which is needed to be performed by an individual with the knowledge of software design. In addition to technical skills, it often requires creative skills in making a design decision. As the examples, three change scenarios that lead to complex business rule changes will be discussed in this section. They are shown in Table 5-4.

The first scenario initiates the modification of two existing business rule statements namely the calculation of bill and the calculation of invoice amount. These business rule changes consequently lead to a minor change in software design i.e. the introduction of

`hasMaxBill` attribute in the `Paymaster` class. Although this scenario causes minor software change, the knowledge of software design is needed to perform the change.

Table 5-4 The examples of change scenarios for complex business rule change

Change Scenarios	Changed Business Rules

1. HCP introduces new package for paymaster. In this package, the paymaster may limit the maximum amount of each patient bill to RM 20.00, and the excessive cost is absorbed by HCP. However, the paymaster must pay a monthly fee of RM5.00 for each patient.	The amount of a bill is computed as
	let amount = the sum of all amounts of bill items
	if (patient is a panel patient) AND (paymaster has maximum bill amount) AND (amount > RM 20.00)
	amount = 20
	The amount of HCP service invoice is computed as
	let amount = the total of the invoice items
	if (paymaster has maximum bill amount)
	amount = amount + 5 * the number of paymaster's patients

2. Paymaster wishes to provide different healthcare benefit coverage for different groups of its payees.	If (the patient is a panel patient) AND (the patient is an executive staff) then the patient is entitled to any type of treatments and medical procedures. If (the patient is a panel patient) AND (the patient is a production staff) then the patient is entitled for an outpatient treatment.

3. HCP would like to introduce a 5% discount on the invoices to preferred paymasters as a way to express gratitude to the loyal, potential, and good paying paymasters.	If (a paymaster has been a paymaster panel for more than 5 years) then (the customer is a 'loyal' customer).
	If (a paymaster has an average of at least RM24000.00 for the invoices over the last five years) then (the paymaster is considered as a 'potential' customer).
	If (a paymaster never has a past due invoice for the last two years) then (the paymaster is considered as a good paying paymaster).
	When (the invoice in created) if (the paymaster is a loyal, potential and good paying customer) then (set the discount of the invoice to 5%)

The second scenario is used to explain a more complex business rule change that requires major changes in software design. In this scenario, the paymaster decided to introduce different healthcare benefit coverage to different levels of their payees. For example, the executive staff are entitled to any medical treatment and medical procedures whilst the production staff are only paid for outpatient treatments. It is obvious that simply implementing this new requirement into the existing `Paymaster` or `PanelPatient` class may increase the complexity of these classes. Therefore, additional classes that are responsible to manage the healthcare benefit coverage are required to be added to the existing software design. The possible candidates for these classes include `BenefitCoverage`, `SelectedClinic`, `MedicalProcedure`, and `Entitlement`. The introduction of new classes in software design is considered as a major complex change since it requires large efforts in the implementation of such change.

In contrast to the above two scenarios that are used to represent different level of impacts of business rule changes, the third scenario is used to explain the importance of creative skills in the implementation of business changes during evolution phase. Consider the first solution to the third scenario that requires a number of new inference rules to be added to define a loyal, potential, and good paying customer. In addition to these business rules, an action assertion rule that initializes the value of the invoice discount during invoice creation should also be added. The new business rules are shown in Table 5-2. The introduction of the new inference rules consequently requires `isLoyal()`, `isPotential()`, and `isGoodPaying()` operations to be added to the `Paymaster` class. Similarly, the newly introduced action assertion rule requires component engineer to modify the action component of the transition from the initial state to 'Active' state in the STD for `HCServiceInvoice` object.

However, the above solution was found far complex than what it should be. A simpler solution might be to modify the algorithm part in the computation rule that calculates the amount of healthcare service invoice. To be precise, the algorithm should firstly check for the loyal, potential, and good paying conditions, and consequently deduct 5% discount from the amount of this invoice if the conditions are true. In this case, the creative skills of component engineers in performing complex changes are as important as their technical skills.

5.5 Discussion

This chapter has provided an example of using BROOD in the development and evolution of the MediNET application. During development, it demonstrated how to perform the BROOD activities that explicitly specify business rules and link them to software design. Although there are additional tasks that should be performed during development, these tasks have provided benefits in performing changes in MediNET. During evolution, this chapter discussed the implementation of the simple and complex changes using the selected scenarios. Contrary to the traditional approaches, the availability of the BROOD metamodel, templates, and process description have simplified the MediNET evolution tasks in most cases.

Using the experiences in using both traditional (object-oriented) and BROOD approaches in the development and evolution of MediNET, the author observes several common quality attributes that are maintained or improved by BROOD. By using the standard UML for software design, BROOD maintains the well-known object-oriented design quality attributes such as modularity, high cohesion, low coupling, efficiency, and portability. It also improves the traditional approaches in terms of other quality attributes such as *requirements traceability*, *software evolvability*, and *approach usability*.

Regarding *requirements traceability*, the employed traditional approach does not directly explicitly link the MediNET software design to its user requirements. Instead, it provides a so-called 'seamless transition' from the use case models that document the user requirements to the analysis and design models. The requirement structure is not suitable to be directly linked to software design, and the business rules are embedded in both requirements specification and software design models. Consequently, user requirements are less traceable in MediNET using the traditional approach. In contrast, BROOD smoothly transformed the MediNET requirements into the structured business rules specification. The introduced structures allow the linking of business rules to the related software design components.

Concerning *software evolvability*, it is slightly hard to implement the changes of MediNET business policy in software design using the traditional approach. All types of changes must be performed by software engineers with the knowledge of MediNET software design. Since software engineers do not initiate the business changes, they often have to repeat all phases in MediNET development lifecycle especially requirements and analysis phases. It is also hard to locate the related software design components since there is no explicit link between the MediNET design models and its user requirements. Therefore, the traditional approach requires more software evolution efforts than those of the BROOD approach. There is also no detailed process description to guide software engineers in performing the changes. On the other hand, BROOD provides the documented links between MediNET business rules specification and its software design. The links allow the MediNET evolution to be driven by its business rule changes. The business rule-driven software evolution may ensure the alignment of

the user requirements with the implemented MediNET application. However, there is one drawback identified with regard to software evolution. As mentioned in section 5.4.2, it is possible for software engineers to make a mistake in making an appropriate design decision during the implementation of a complex change as shown in the last example of the change scenarios. It is learnt from the example that, although some of the evolution tasks may be automated, both technical and creative skills are required to implement the complex change.

In relation to *approach usability*, the traditional approach is easier to be used during development phase since it does not have to deal with additional steps that were added to explicitly specify, document, and link business rules specification to software design. These steps were found to increase the complexity and duration of software development process. However, the availability of the business rule typology and templates, which provide the guidelines for the analysis of business rule statements and the identification of rule phrases, were found useful in minimizing the above problems. The business rule templates have improved the MediNET system understandability and increased the involvement of business users in the MediNET development. During evolution, BROOD was found easier to be used than the traditional approach. Using BROOD, business users may instantly perform the simple business rule changes as demonstrated in the MediNET application. Rapid change implementation is important especially in business critical application with intolerable downtime. The detailed process description facilitated the implementation of complex changes in MediNET. The linking between the MediNET business rules and its software design provide business rule traceability, which consequently facilitates the propagation of the business rule changes to their respective changes in software design. The BROOD metamodel language definition using EBNF also allows the realization of the automated change propagation. The design and implementation of the automated tool prototype will be discussed in the next chapter.

Chapter 6

The Design and Implementation of the BROOD Tool Prototype

This chapter presents the design and implementation of the BROOD tool prototype that implements the modelling concepts defined in Chapter 3 and demonstrates the possible automation of the BROOD process described in Chapter 4. First, it presents the overview of the architecture and roles of the BROOD tool. Second, it explains the Generic Modelling Environment (GME) modelling concepts and architecture to establish the modelling vocabularies that are used in the later sections of this chapter. It is important to understand the GME concepts since the BROOD tool is developed on top of the GME environment. Third, it presents the physical metamodels that implements the BROOD metamodel discussed in Chapter 3. Fourth, it explains the functionalities and inner-working of the BROOD tool in terms of graphical model editing, rule phrase entries management, business rule composition, and business rule modification. Finally, the summary is presented at the end of this chapter.

6.1 Introduction

In Chapter 4, the software process was proposed to utilize the BROOD metamodel in developing a more practical approach to software evolution. The process simplifies the evolution activities by providing business rule traceability in object-oriented software design which in turn facilitating business rule-driven evolution. However, the process introduces several additional activities to the traditional object-oriented software development. These additional activities include the documentation of business rules and linking them to software design. Although these activities bring great benefit in the evolution of a business rule intensive software system, but they make a software development process too complicated without an automated tool support.

In this chapter, the author describes the design and implementation of the BROOD tool prototype that supports the BROOD process. This tool assists users in performing the development and evolution activities such as business rule creation and management, software design editing, and business rule change propagation. It was developed on top of the Generic Modelling Environment (GME), which is a configurable modelling environment. The metamodel and templates, which were discussed in Chapter 3, were used to generate the BROOD tool environment. The environment may be used to visually edit the software design models, business rule specification, and rule phrase entries. Three main modules (i.e. interpreters in GME terms) were developed to simplify the rule phrase management, business rule composition, and business rule modification. These modules also perform the automated propagation of business rule changes to the respective software design elements, which is impractical to be performed manually. Since some features in the BROOD tool are also used by business users, the friendly user interfaces are provided to ease the management and traceability of business rules by this type of users. The metamodel, the graphical model editor, and the above three modules are located at the core component and user application layer in the BROOD tool architecture. The rule phrase entries, business rule specification, and software design models are considered as the stroge layer. The overview of the BROOD tool architecture is shown in Figure 6-1.

Figure 6-1 also illustrates the informal overview of the roles performed by the BROOD tool in the BROOD software lifecycle. During development, software engineers use the BROOD tool to develop software design models, populate the rule phrase entries, and develop a business rule specification. They also use the tool to provide business rules traceability by linking their specification to software design components. The BROOD tool maintains the consistencies between business rule and the linked software design each time the business rule is created or modified. The completed software design models are assumed to be manually or automatically used in generating the application software.

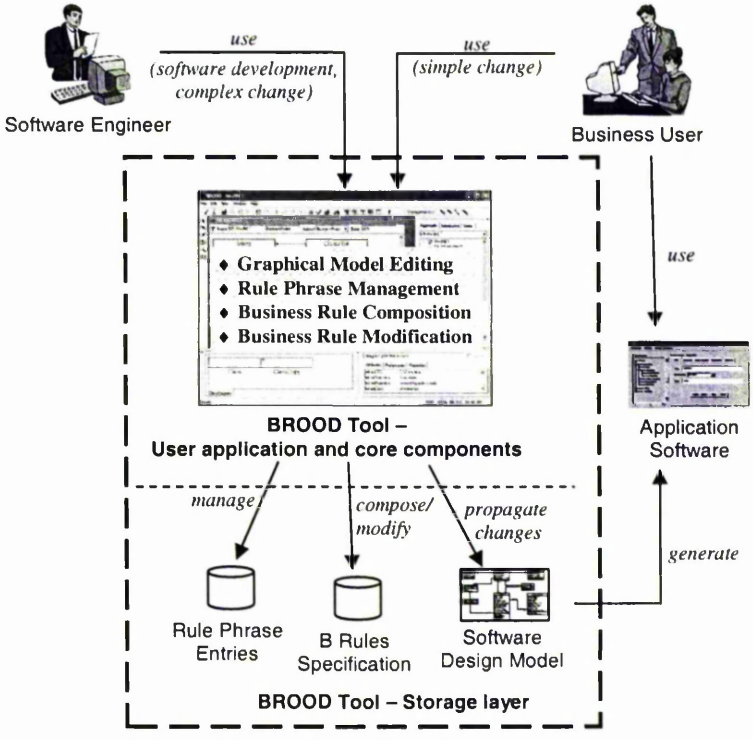

Figure 6-1 The illustration of the roles and architecture of the BROOD tool

During its operation (or evolution), the software application may need to be changed according to the changes of business policies. The changes include the modification or deletion of existing business rules, or the addition of new business rules, in accordance with the changes in business environment. As explained in Chapter 5, the business rule-related software changes may be divided into to types: simple and complex changes.

The simple changes such as the modification of the rule phrases can be done by an authorised application user whilst complex changes should be done by a software engineer. The BROOD tool is capable to provide a full support in performing simple changes whilst partial support in performing the complex changes. The fully automated support to perform complex changes is almost impossible since it requires creative skills of software engineers in making a design decision.

6.2 Generic Modeling Environment (GME)

The Generic Modeling Environment (GME) [Ledeczi et al., 2001a; VU, 2003], which was developed by the Vanderbilt University in Nashville, is a configurable toolset that supports the easy creation of modelling environments. The created modelling environment can be subsequently used for building large scale, complex models. The powerful modelling concepts such as model hierarchy, multiple aspects, sets, references and constraints are integrated in the GME tool. GME also contains integrated model interpreters for translating and analysing the models under construction. The configurable feature of GME provides great flexibility for methodologist especially for the frequently evolved modelling paradigm. This feature was also considered as the main reason for this research to choose GME in experimenting and demonstrating its proposed concepts.

In GME, the modelling configuration is accomplished through a metamodel that specifies the modelling paradigm (or modelling language) of the application domain. The metamodel defines the syntactic, semantic and presentation information of the domain for example, the concepts that are used to construct the models, the relationships that may exist among those concepts, the organization and view of the concepts by the modeller, and the rules governing the construction of models. The metamodel is composed using different combinations of the GME modelling concepts such as model, atom, reference, connection, and aspect.

After the completion of the metamodel, GME requires an interpretation process to be invoked in order to register the paradigm in the GME database. Once the interpreters are created, environment users can create domain models and perform analysis on those

models. Models are organised in a project and stored in a model database. A project is a group of models created using a particular modelling paradigm.

GME provides a constraint manager as a general mechanism for representing modelling rules and constraints. In GME, constraints are expressed using the MultiGraph Constraint Language (MCL), a predicate logic language based on the Object Constraint Language (OCL). The constraint manager is fully compliant with the standard OCL 1.4 specification [OMG, 2001].

GME also incorporates a number of other significant features which make it a very powerful and competitive tool. These features include 'interpreter' and 'add-on' components for programmability, XML format for model portability, and 'decorator' for enhancing model drawing capability. It also provides a development kit that is based on the Microsoft's Visual C++ programming environment for the purpose of component development based on the Builder Object Network (BON) framework. This section presents an overview of the modelling concepts and architecture of the GME.

6.2.1 GME Modelling Concepts

Prior to going into the detailed discussion on the development and implementation of the BROOD tool prototype, it is important to understand the modelling concepts that build the GME modelling (or metamodelling) vocabulary. The modelling concepts are used in model composition. Figure 6-2 shows the UML class diagram of the GME modelling concepts and their relationships. This diagram is also considered as the meta-metamodel that is used to define the implementation metamodel of the BROOD tool.

As shown in Figure 6-2, a *project* contains one root folder, which may contain zero or more other folders. In BROOD tool, only a single root folder is allowed in a project. A *folder* acts as a container to organise a number of models. *Model* is an abstract object that represents something in the real world. It typically has parts i.e. other objects contained within the model such as atoms, references, sets, and connections. It also may contain other models as part-models of the same or different kinds. In this case, it will be a parent node to the part-models in the model hierarchy. All GME models are

composed by different combinations of the GME first class objects (FCOs, i.e. the generic entities at the lowest level of design in the GME paradigm).

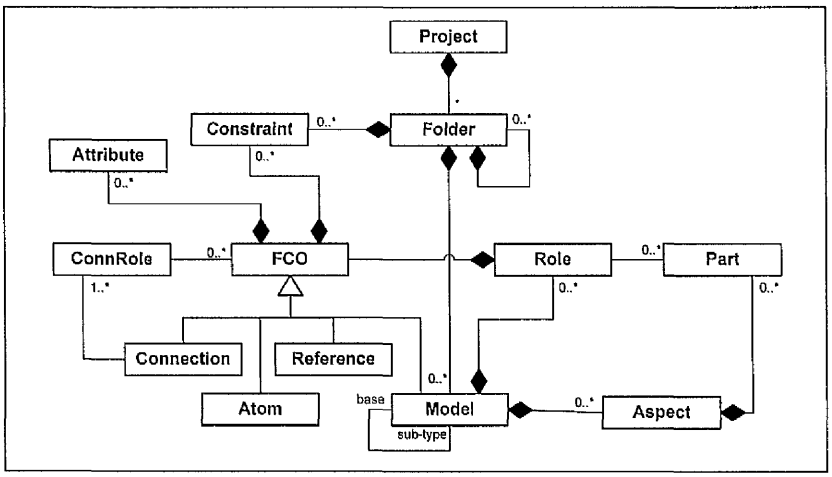

Figure 6-2 GME modelling concepts [Ledeczi et al., 2001a]

Atoms are single modelling objects that do not have internal structure, although they can have attributes. Atoms can be used to represent entities, which are indivisible and exist in the context of their parent model.

References are parts that are similar in concept to pointers found in various programming languages. Reference parts are objects that refer to other modelling objects. A reference part can point to a model, an atomic part of a model, a model embedded in another model, or even another reference part. A UML class from one class diagram that is replicated in another class diagram is an example of a reference that is used in the BROOD tool.

A *connection* is used to express a relationship between two objects. The modelling paradigm may specify the kind of the relationship to be included in a model and the kind of object that may participate in the relationship. A connection may only express the relationship between objects contained by the same model. A reference object can be used to create a relationship between objects from different models.

All FCOs such as models, atoms, references, and connections may have the attributes. An *attribute* is a property of an object that is best expressed textually. The modelling

paradigm may define the number of attributes for a particular object in their metamodel. The attributes are often set using a form-based graphical user interface model editor.

The visibility of model components during visual model editing is controlled by the defined *aspect*. Each model may have more than one aspect that may hide or show a particular part contained by the model. The existence of visibility of a part within a particular aspect is determined by the modelling paradigm.

6.2.2 GME Architecture

GME uses modular component architecture. The components provide services via the interfaces based on Microsoft's Component Object Model (COM) technology. In general, the components of GME architecture can be conceptually divided into three layers i.e. storage, core components, and user application layers. They are shown in Figure 6-3.

The *storage layer* supports different storage formats including relational databases, proprietary file formats, and XML. However, it currently only supports proprietary binary file format and Microsoft SQL Server 7.0 via ODBC. Using GME binary file format (project file) is faster than ODBC. XML is supported using the import/export facilities. Supporting an additional format (e.g. Oracle) requires the implementation of a single, well-defined, small interface component.

Core component layer is the most important component of GME. It consists of the core component that implements the two fundamental building blocks of a modeling environment i.e. objects and relations. The core component provides distributed access and undo/redo services. This layer also contains two components that use the services of the core i.e. GMeta and GModel.

The GMeta defines the modeling paradigm and configures itself by reading the meta-specifications, while the GModel implements the GME modeling concepts for the given paradigm. The GModel uses the GMeta services for self-configuration. During execution, the GModel uses the GMeta component extensively through its public interfaces. Both components expose their services through a set of COM public interfaces.

The GModel component exposes a set of events such as "object deleted" and "attribute changed". External components can register to receive the messages from these events. It is often used by Add-ons (event-based components), which are useful for extending GME user interface capabilities or for executing domain-specific operations.

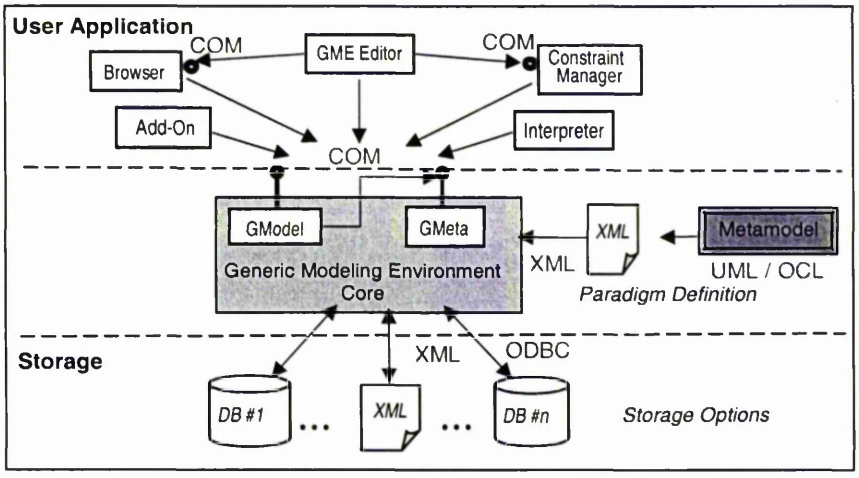

Figure 6-3 GME Architecture [Ledeczi et al., 2001b]

The components at the *user application layer* are provided to allow GME user or application program to interact with the GModel or GMeta components. They include the Browser, Editor, Add-ons, Interpreters and Constraint Manager. The Browser and Editor components provide the graphical user interface (GUI) for visual model editing by GME users. Any operation that can be accomplished through the GUIs can also be done programmatically through other interfaces. This architecture is very flexible and it supports extensibility of the whole environment.

The extensibility is achieved by writing programs that access or modifies model information using several techniques such as GME interpreter, stand-alone program, and Add-Ons. GME interpreter is a component that is loaded and executed using GME environment whilst stand-alone program can be executed without the GME GUI. Typically, Builder Object Network (BON) framework is used by GME interpreter to represent an object in GME model database. BON is a network of C++ objects that

provide read/write access to the objects' properties such as their attributes and relationships. The classes in BON framework is shown in Figure 6-4.

Apart from BON, there are also other technologies which are available for developing the external programs to access GME data for example, COM interface and GME Unified Data Model (UDM-GME). COM interface is the most efficient way to access GME data. However, COM programming and interface protocol handling are more difficult compare to BON framework. UDM-GME is currently an experimental technology that creates interpreters that use an automatically generated paradigm-dependent programmatic interface.

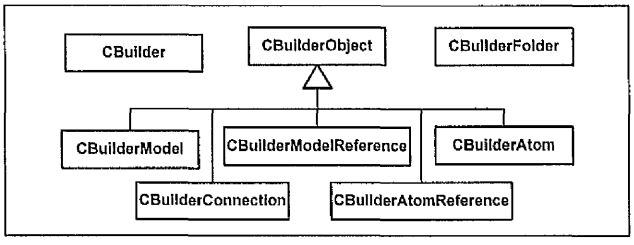

Figure 6-4 The classes in Builder Object Network (BON) framework [VU, 2003]

Add-On is an event-driven model interpreter that is invoked on the occurrence of the registered events. The events, such as "Object Deleted," "Set Member Added," and "Attribute Changed", are managed by the GME core components. Add-On components may register to receive some or all of these events. They are automatically invoked by the core components when the events occur. Add-On is extremely useful for extending the capabilities of the GME User Interface or for reacting to a call of some special operations.

The Constraint Manager evaluates the constraint expressions to verify the constrained model components against their attached constraint. GME may also specify constraints to react to the occurrence of the particular events. The Constrain Manager can be invoked either using the command menu or by the occurrence of the registered events. Therefore, the constraint manager has both features of an Interpreter and an Add-On. Depending on the priority of a constraint, the operation that caused constraint violation

can be aborted. For a less serious violation, the Constraint Manager only sends a warning message.

6.3 BROOD Physical Metamodels

The GME modelling vocabulary that is described in the previous section facilitates the understanding of the BROOD physical metamodels. The physical metamodels are the implementation version of their respective conceptual metamodels discussed earlier in Chapter 3. It follows the semantics of the logical metamodels but slightly differs in structure to comply with GME meta-language as well as to assist the programming of model interpreters.

6.3.1 Business Rule Metamodel

In the business rule metamodel, which is shown in Figure 6-5, each business rule model may contain zero or more rulesets. A ruleset is used to group business rules into a set of closely interrelated rules according to their business process or sub-process. The names or IDs of the business process and ruleset owner are the ruleset's attributes that are used for the purpose of business rule organization. The ruleset ID is used by the model interpreters to uniquely identify the ruleset. A ruleset can be connected to another ruleset using SubSetOf connection to show that the source ruleset is a subset of the destination ruleset. Each business rule must be connected to one and only one ruleset in order to simplify the business rule model traversal by the interpreters.

In the business rule physical metamodel, there are five types of business rules as described in its conceptual metamodel: attribute constraint, relationship constraint, action assertion, computation, and derivation. All types of business rules have two common attributes i.e. template ID, which is used to identify the rule, and statement, which is a complete sentence that describe the business rule.

There are two categories of attributes in each business rule type. First, the attributes that are derived from the rule phrases defined in its sentence templates. The BROOD tool will ensure that all of the necessary rule phrases are assigned the values according to the selected template (stored as the template ID) during the creation of a business rule

instance. For example, to create an attribute term using the template <attributeTerm1> must be | may be <relationalOperator> <value>, the `attributeTerm1`, `relOperator`, and `value` must be assigned the values. The business rule statement can be easily constructed using the values of the attributes (rule phrases).

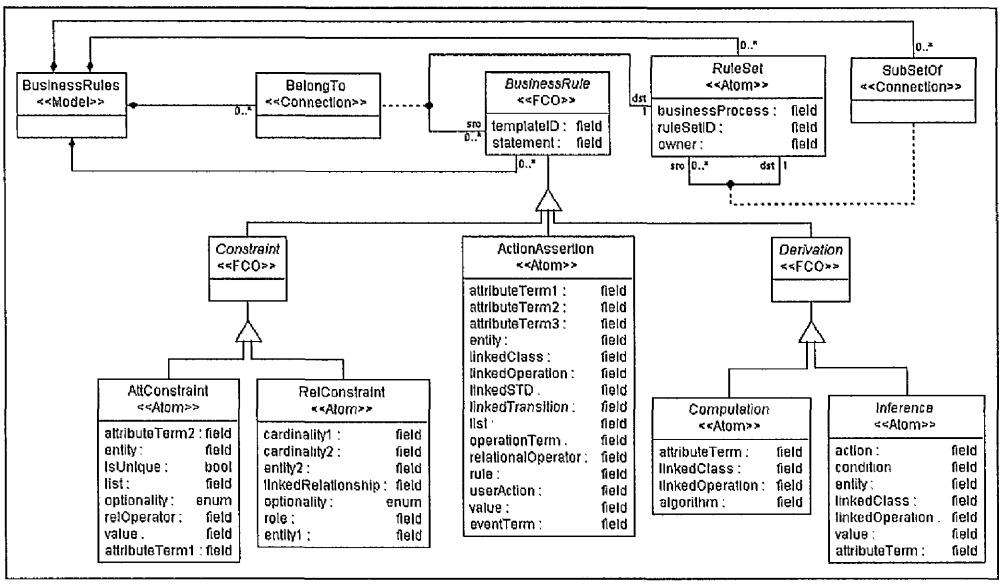

Figure 6-5 Business rule metamodel

The second category of attributes is those used to link or propagate business rule information or changes to software design components. If the linking attributes are defined, the specification of the related design components will be changed is time the rule is created or modified. Table 6-1 shows the attributes of software design components that possibly changed by the introduction of a new business rule.

Table 6-1 Linking the business rules to the software design components

Types	Software Design Components	
Attribute Constraint	Attribute.isUnique	Attribute.isMandatory
Relationship Constraint	Association.dstCardinality Association.dstRoleName Composition.partCardinality Composition.partRole	Association.srcCardinality Association.srcRoleName Composition.wholeRole
Action Assertion	Transition.actionName Transition.eventName Transition.eventOperation	Transition.actionSpec Transition.eventSpec Transition.guardSpec
Computation	Operation.specification	
Derivation	Operation.specification	

6.3.2 Software Design Metamodel

In BROOD approach, there are two software design models: class diagram and statechart diagram. These models were extracted from the UML Specification version 1.4 - they are respectively shown in Figure 6-6 and Figure 6-7.

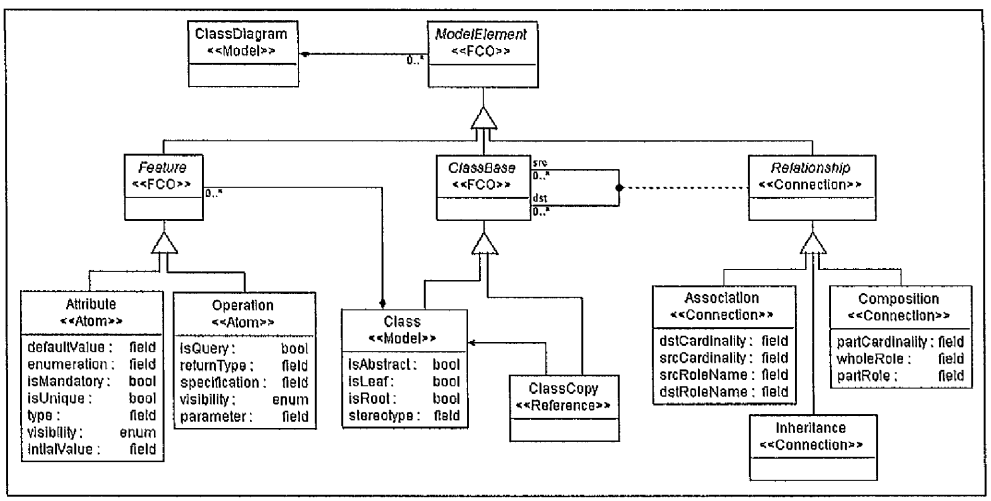

Figure 6-6 Class diagram metamodel

In the class diagram metamodel, a class diagram contains zero or more model elements. A model element has three sub-types: class base, feature, and relationship. Class base is an abstract object that is used to introduce a special type of a class i.e. a class copy. A class copy is a reference to a class object. A class is a model that has zero or more

attributes and operations, similar to the concept of a UML class. Two additional attributes were included in the attribute atom, i.e. `isMandatory` and `isUnique`, to respectively store the optionality and uniqueness values in accordance with the attribute constraint business rule specification. A class or class copy can be connected using one of the three relationship kinds: association, composition or inheritance.

Regarding the statechart diagram, it may have zero or more states and transitions. Each state has three attributes namely entry action, do action, and exit action which respectively specify the action when an object enter, stay, and leave the state. Each state can be connected using a transition. A transition has the attributes to store the specification of the event that generates the transition, guard (condition) that must be satisfied for the transition to happen, and action that is triggered as the side effect of the transition.

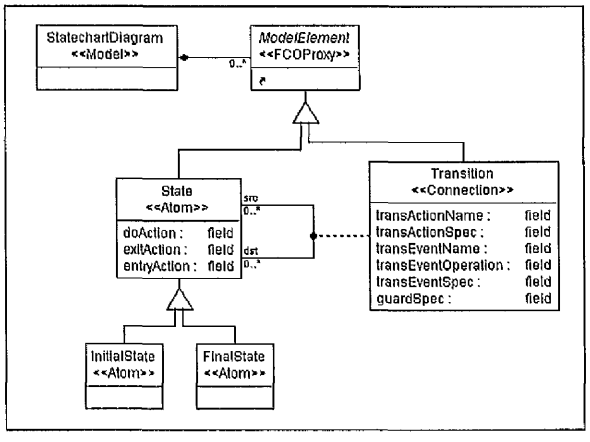

Figure 6-7 Statechart diagram metamodel

6.3.3 Rule Phrase Entries

In terms of the BROOD tool prototype, the rule phrase types may be categorized in three category namely 'literal', reusable (not directly linked), and linked rule phrases. The literal rule phrases are not stored in the rule phrase entries since they are too simple. Instead, they are only stored as the attributes of the business rule instance. The literal rule phrases have a single value or a very small number of possible values. The examples of the literal rule phrases are value, optionality, and uniqueness.

The reusable and linked rule phrases are stored in the rule phrase entries. They are shown in the rule phrase metamodel in Figure 6-8. The reusable rule phrases are not directly linked to the software design components, but their information is used by the interpreters to propagate the business rule changes to the related software design component. They are stored in business rule specification as the common rule phrases that may be used during business rule composition. The types of the reusable rule phrases include cardinality, relational operator, time unit, and list. The linked rule phrases are also stored and reused similar to reusable rule phrases, but it also directly linked to their respective software design components. For example, entity is linked to a class, attribute term is linked to an attribute, operation term is linked to operation, event is linked the operation that generate the event, and date-time is linked to the operation that return the particular date and time. The linked operation of an event and date-time rule phrases is not mandatory. The rule phrase entries are often maintained software engineer or business user with software design knowledge. Once the rule phrase entries is completely populated, business user may create or modify business rules based on the existing rule phrases in the rule phrase entries.

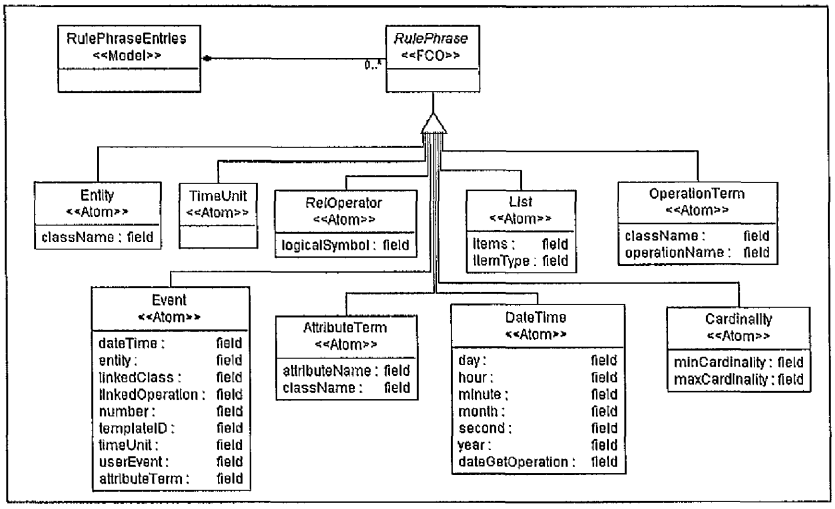

Figure 6-8 The metamodel of rule phrase entries

6.3.4 Modelling Constraints

Most of the BROOD modelling rules and constraints, such as the cardinalities and permitted types of a relationship, are controlled by its metamodels. Such rules and constraints are important in maintaining the correctness of the syntax and semantics of the developed models. However, the explicit constraints are needed for the additional modelling restrictions that ensure the consistency of the model or its components. These explicit constraints were included as the constraint objects in the BROOD metamodel. They are specified using the MCL, which is an extended version of the OCL specification. Each constraint object was connected to the metamodel element of its constrained object. It may be invoked either manually using the check constraint command or automatically upon the occurrence of the registered events. The registered events may be the object, model, or project events. They are generated for example, when the project is close, model is saved, or the object created. The respond to the violation of the specified event depends on the constraint priority – the high priority constraint will abort the current operation whilst the low priority constraints will only display the warning message. In the current version of this prototype, only the simple constraints were implemented. The examples of these constraints and their OCL specifications are given below:

- RPEntriesSingleton: `project.allInstancesOf(RulePhraseEntries)->size <2`
- NotEmptyTemplateID: `self.templateID.trim() <> ""`
- UniqueRulePhrase:
```
project.allInstancesOf(Cardinality) ->
                    select( c | c.name = self.name ) -> size = 1
```

In the above examples, RPEntriesSingleton controls the project to only have a single rule phrase entries model. This restriction may simplify the maintenance of a large number of unique rule phrases. The second constraint, i.e. NotEmptyTemplateID, checks the `templateID` attribute of the connected business rule instance to ensure that the value is not empty. It is important in ensuring that `templateID` always has a value since it is frequently used in the developed interpreters for instance the generated business rule statement depends on the selected `templateID`. The last constraint in the above examples, namely UniqueRulePhrase, was attached to all rule phrase atoms or objects to ensure that each rule phrase is unique. A unique rule phrase name may avoid the

business user confusion on the meaning of the selected rule phrases in composing a business rule.

6.4 BROOD Tool Features

Having completed and interpreted the physical metamodels discussed in the previous section, the newly generated tool environment of the BROOD modelling paradigm that is based on the GME meta-paradigm is now ready to be used. In its main window, which is shown in Figure 6-9, the name of the currently opened project is displayed on the Title bar. Menu bar and tool bar allow access to certain commands provided by the environment. The buttons to execute the interpreters are located at the right-end of the tool bar. Mode bar contains selection and connection buttons. Model editing window is used to visually construct and edit the models. Users may add the model component, which is called part in GME, by selecting it from part browser window and dragging it to the model editing window. Each model editing window has its own title bar that displays the name of the currently edited model. The attributes of the model can be added or modified using the form-based attribute browser window. Users may navigate the models in the current project using the tree structure view provided by model browser window.

6.4.1 Model Editing

There are four main types of models can be created using the BROOD tool: rule phrase entries, business rule, class diagram, and statechart diagram. The simplest way to create a new model is to right-click the project name in the model browser window and select Insert Model command from the pop-up menu. Users may select the type of the model to be created from the displayed choices. As the type is selected, a new item with the default model name is inserted in the tree view of the current project. By double-clicking on the model name, the model editing window is displayed and the model is now ready for editing.

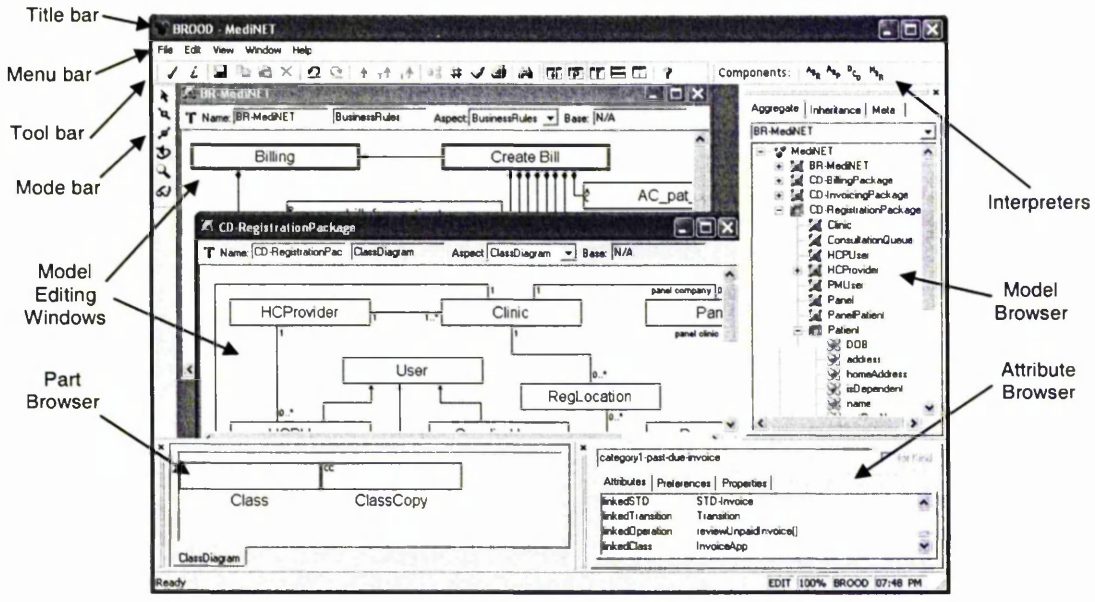

Figure 6-9 The generated BROOD tool environment

The model component (which is called part in GME terms) may be inserted into the model by dragging its type symbol from the part browser and dropping it on the model editing window. The contents of the part browser are changed each time users change the active model editing window. Different symbols are used to distinguish different component types for example, the ruleset is displayed as a double outline box whilst a business rule is represented as a single outline box.

The BROOD tool also provides a convenient way to create a connection between to model components. Users may click the connect mode button on the mode bar and consequently click on the source and destination component on the model editing window. If there is more than one type of connections defined for the source and destination types, the pop-up menu will appear to allow user to select the right type of connection. In certain situations, BROOD allows users to make a connection from a component in one diagram to another component in another diagram. For example, the `Bill` class in the Billing class diagram can be connected to the `Patient` class in the Registration class diagram by creating a copy of the `Patient` class in the Billing class diagram. The `Bill` class is then connected to the `Patient` class copy.

The above graphical model editing is convenient to be used in visual modelling such as the development of the class diagram and statechart diagram. However, it is less convenient in the development of business rule specification. For instance, the addition of a new business rule requires users to link it to several rule phrases and software design components. Using the graphical model editing alone requires user to manually search for the related rule phrases in the rule phrase entries and the related components in the class or statechart diagram. This is an inconvenient, time-consuming, and error-prone process.

Therefore, several modules were developed to simplify certain modelling tasks that are impractical to be performed using the design editors. These modelling tasks include the population of rule phrase entries, the addition of a new business rule, and the modification of the existing business rules. In the developed modules, Builder Object Network (BON) was used as a high level programming library to access and modify the BROOD models and their components. These modules are actually interpreters in GME terms - they are called modules since they provide much functionality beyond interpreting the models. The inner-working of these modules and the way on how to use them in assisting the BROOD process will be discussed in the following sections.

6.4.2 Populating the Rule Phrase Entries

The rule phrase entries must be populated with an adequate number of rule phrases since they are needed to compose a new business rule statement. Model editing window can be used to add the simple rule phrases that are not directly linked to software design components such as cardinality, relational operator, list, and optionality. For the rule phrases of other types such as entity, attribute term, operation term, and event, the developed Add Rule Phrase (ARP) module was developed to simplify the task of creating and adding them to the rule phrase entries.

As the module is invoked, the ARP module displays a window that allow user to choose the rule phrase type. Next, the module will display the respective window according to the selected rule phrase type. As an example, Figure 6-10 shows the window that is displayed when attribute term is selected as the rule phrase type. The first combo box in the window is automatically filled with the names of the existing classes in the current

project. When a user selects the class in the first combo box, the attributes of the selected class are inserted into the second combo box. User may select the attribute from the second combo box and linked it to the newly created rule phrase, which is entered in the new rule phrase text box.

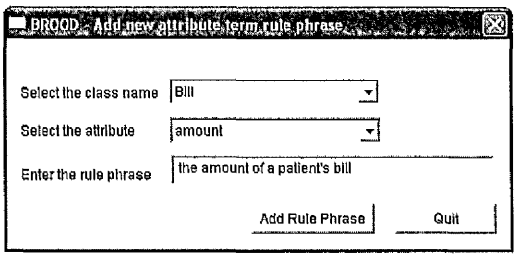

Figure 6-10 Adding a new attribute term rule phrase

In the ARP module, the programming task of populating the class and attribute combo boxes with the available classes and respective attributes was simplified by using the BON framework. For example, the process of populating attributes starts with the traversal of all models in the root folder returned by the BON CBuilder object. Upon finding a class diagram model, it traverse all classes in that class diagram to look for the currently selected class. As the selected class is found, it retrieves the name of each attribute of the selected class, and subsequently adds the name to `m_AttributeComboBox` combo box. The following source code snippet shows the extensive use of BON components in retrieving the class diagram information.

```
const CBuilderModelList * models = builder->GetRootFolder()->GetRootModels();
POSITION pos = models->GetHeadPosition();
while (pos)
{    CBuilderModel *model = models->GetNext(pos);
     if (model->GetKindName() == "ClassDiagram")
     {    //Get the classes in the model
          const CBuilderModelList *classes = model->GetModels();
          POSITION pos = classes->GetHeadPosition();
          while(pos)
          {    CBuilderModel *desClass = classes->GetNext(pos);
               if (desClass->GetName().Compare(className) == 0)
               {    const CBuilderAtomList *attributes = desClass->GetAtoms("Attribute");
                    POSITION pos = attributes->GetHeadPosition();
                    while(pos)
                    {      const CBuilderAtom *attribute = attributes->GetNext(pos);
                           m_AttributeComboBox.AddString(attribute->GetName());
                    } //end while
               }//end if(desClass..) - if the selected class
          }//end while - individual class traversal
     }//end if(model..) - if a class diagram
```

```
}//end while(pos) - models traversal
```

After the new rule phrase is entered and the linked attribute and class are specified, user may create a new attribute term rule phrase by clicking the Add Rule Phrase button. The rule phrase entries model that acts as the container to the new rule phrase is then obtained via traversing the root models similar to the above class diagram traversal. As the model is found, the new atom for the new rule phrase is created, and its name and attributes is assigned to the atom. The example of source codes for creating a new attribute term is shown in the following code snippet:

```
if (model->GetKindName() == "RulePhraseEntries")
{    //--Create new AttributeTerm and set its attributes
     CBuilderAtom *newAttTerm = model->CreateNewAtom("AttributeTerm");
     CString attCName = "className", attAName = "attributeName";
     newAttTerm->SetName(rulePhrase);
     newAttTerm->SetAttribute(attCName,className);
     newAttTerm->SetAttribute(attAName,attributeName);
}//end if
```

The above implementation provides a fairly simple example of using the BON components in the BROOD tool. Model traversal, atom creation, reading/writing the values of the atom's attributes are the examples of the most frequently used BON components. The programming task is more complicated for a more complex rule phrase. For example, in adding an event rule phrase, the writing of the rule phrase must be done according to the selected template. If the event is linked to any class operation, the event specification must be propagated to the specification of the linked class operation.

6.4.3 Adding a New Business Rule

The Add Business Rule (ABR) module was developed to assist user in adding a new business rule to the selected business rule model. The ABR module performs two main tasks: business rule composition and software design updating. In business rule composition, rule phrases are used as the building blocks to construct a new business rule statement. Majority of the rule phrases are available from the rule phrase entries. For certain types of rule phrases, such as value and number, they are not stored in the rule phrase entries. Instead, their values are entered during rule composition and stored as the rule attributes. With regard to updating the software design, the ABR module

automatically updates the software design information to correspond with the newly composed rule. The action taken depends on the selected rule type and template. For example, the attribute and relationship constraints are directly linked to the attribute and relationship in the class diagram.

As the ABR is invoked, it shows the selection window that allows a user to specify the type of the business rule to be added. A business rule composition window is displayed after the user made the choice. As the example, the window that is displayed when the user chooses a relationship constraint is shown in Figure 6-11. User may enter the name and choose the ruleset for the new business rule. The rule template must be chosen from the listed choices. As the template is selected, the rule phrase type combo box is populated according to the selected template. The available rule phrases are displayed in the rule phrase list box. User only needs to double-click the desired rule phrase to select it. The selected rule phrase is inserted in the selected phrases list box. Use construct rule button to display the composed business rule statement.

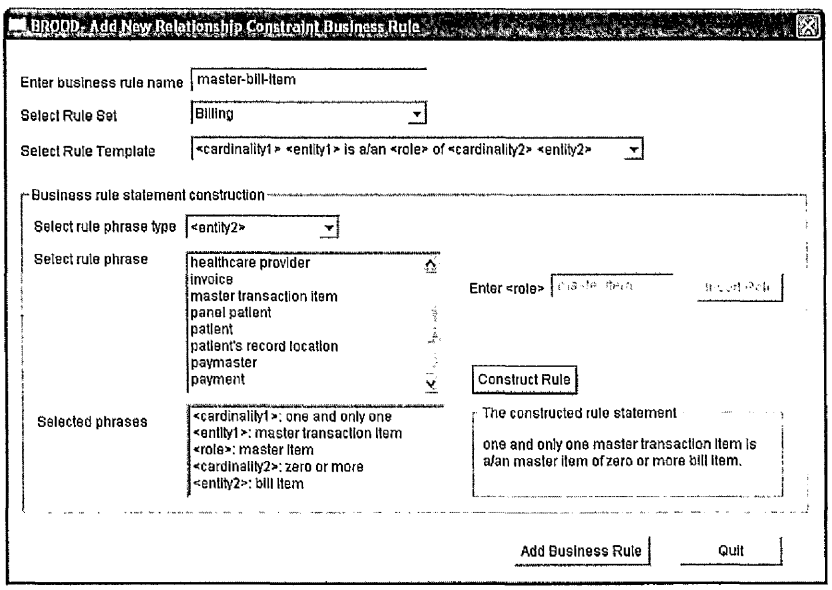

Figure 6-11 Adding a new relationship constraint

Having composed the business rule statement, the business rule is now ready to be added to the currently opened business rule model. As mentioned above, the ABR module does not only compose business rule but it also performs a trickier task i.e.

updating the design components according to the newly composed business rule. In the above relationship constraint example, the relationship constraint atom is firstly created for the new business rule and it is consequently added to the current model after a user clicked the Add Business Rule button. Next, it searches for the existing association relationships between the first and second entities. Then, it displays the pop-up window that allow user to link business rule to one of the existing associations or create a new association. Once the association is selected or created, the cardinalities and role information form the business rule is transformed to their respective attributes of the selected association.

The creation of a new connection is trickier when the source and destination classes are not located in the same diagram or the destination class is a class copy. To solve this problem, the ABR module traverse all models and checks both classes and class copies to find the source and destination classes. If the classes are located in the same class diagram, it will create a connection from the source class to the destination class or class copy. However, if the classes are located in different diagram, the ABR module must create a new class copy as the destination for the new connection. The source codes that implement this task is shown in the following code snippet:

```
//--Create a new Association connection where both classes in the same class diagram
if(classDiagram1 == classDiagram2)
{    if (!dstClass->GetName().Compare("NULL") == 0)
         conn = classDiagram1->CreateNewConnection("Association",srcClass,dstClass);
     else
         conn = classDiagram1->CreateNewConnection("Association",srcClass,dstClassRef);
     conn->SetName(linkedRelationship);
     conn->SetAttribute(srcCardAtt,designCard1);
     conn->SetAttribute(dstCardAtt,designCard2);
     conn->SetAttribute(srcRoleAtt,rolePhrase);
} else
// Create a new Association where srcClass and dstClass resided in different class diagram
{    CBuilderModelReference *newClassRef =
                          classDiagram1->CreateNewModelReference("ClassCopy",dstClass);
     newClassRef->SetName(dstClass->GetName());
     conn = classDiagram1->CreateNewConnection("Association",srcClass,newClassRef);
     conn->SetName(linkedRelationship);
     conn->SetAttribute(srcCardAtt,designCard1);
     conn->SetAttribute(dstCardAtt,designCard2);
     conn->SetAttribute(srcRoleAtt,rolePhrase);
}
```

6.4.4 Performing Business Rule Changes

The ultimate aim of the BROOD tool is to simplify the implementation of business rule changes. The Modify Business Rule (MBR) module was developed to assist tool users

in performing this task. The MBR module starts with displaying a tree structure of the rulesets and business rule statements of the currently opened business rule model. User may browse the business rule statement and select the statement to be modified. After the statement is selected, the MBR module traverse the business rule model to obtain the business rule object (atom), create a new window according to the type of the selected rule, and populating the window with the current business rule information. Figure 6-12 shows the window when a user select 'WHEN 30 day after the creation date of the invoice IF current balance of the invoice is greater than 0 THEN trigger issue the first reminder' rule statement from the tree view.

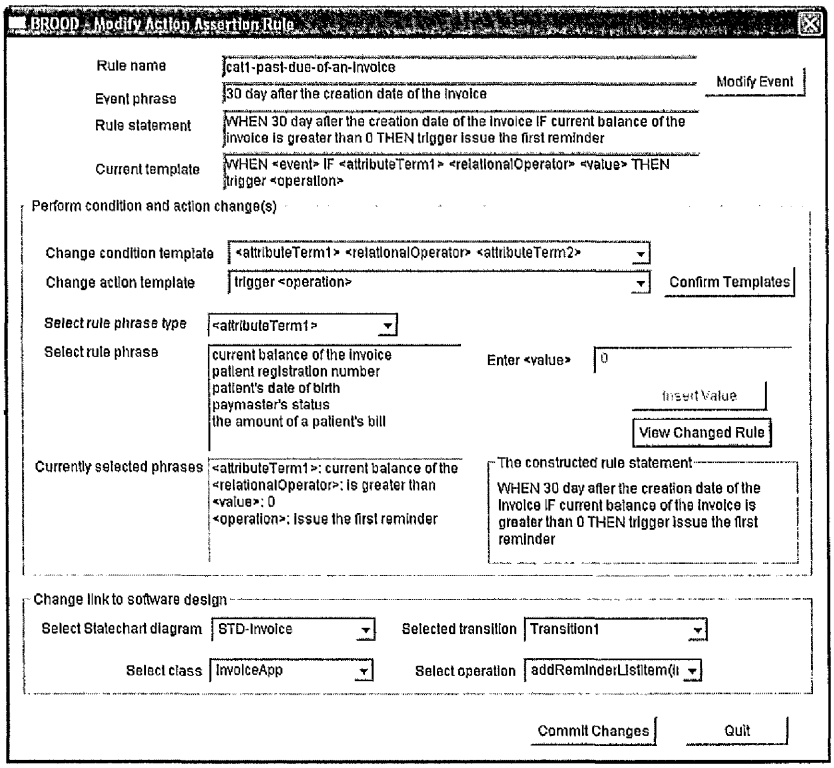

Figure 6-12 Modifying an action assertion business rule

As shown in Figure 6-12, the window is initially populated with the existing rule phrases and templates of the selected business rule statement. User may modify the event, condition, action, and linked software design components using this window. With regard to the condition and action modification, user may change their templates

by selecting one of the listed choices in the provided combo boxes. After the Confirm Template button is clicked, the rule phrase type combo box is populated with the names of the rule phrase types found in the selected templates. User may select a rule phrase type to change from rule phrase type combo box. As the rule phrase type is selected, its instances from rule phrase entries are listed in the rule phrase list box. User may select the item by double-clicking. If the user select <*value*> from the rule phrase type combo box, the edit box and Insert Value button will be activated to allow the user to enter the value of <*value*>. The selected rule phrases are displayed in the selected rule phrases list box. User may view the changed rule statement using the View Changed Rule button.

Upon clicking the Commit Change button, the business rule changes are automatically propagated to the linked software design components. The specification of event, condition, and action of the business rule are transformed to event, guard, and action of the linked state transition in the selected statechart diagram. The business rule may also be linked to the operation that performs the specified action on the occurrence of the event and the satisfaction of the condition. For example, this action assertion is implemented in `addReminderListItem()` operation of the `InvoiceApp` class that adds the paymaster into the list of the category 1 past due paymasters when the payment is not received within 15 days from the invoice date. The list is subsequently used to manually issue the first reminder letters. In this example, the changed business rule specification is transformed to the design specification and automatically inserted in the specification of the `addReminderListItem()` operation. In certain occasions, user may need to change the linked design components. However, changing the linked software design components is infrequently happened during the software operation.

User is also allowed to change the event specification by clicking the Change Event button. As the button is clicked, the window shown in Figure 6-13 is displayed.

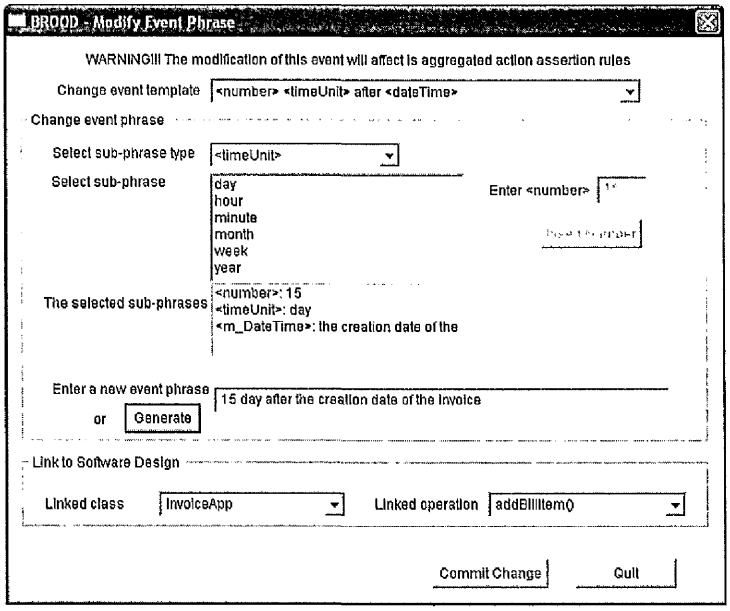

Figure 6-13 Modifying an event rule phrase

As shown in Figure 6-13, the currently selected template and rule phrases are displayed on the window. User is allowed to change the template, however it must be carefully done since the event might be used by other action assertion rules. The most frequent change to an event specification is to change the value of its rule phrase. For example, as described in the second change scenario in section 5.4.1 (in Chapter 5), a user may wish to change the number of days of a category 1 past due invoice from 30 to 15 days. This can be done by very easily by selecting <number> from the sub-phrase type combo box, enter the value in the <number> edit box, and press the Insert Number button. Next, user may choose either to type in a new event phrase or automatically generate based on the modified event specification. Finally, user may click Commit Change button to save the event changes and return to the caller window. The event phrase and rule statement in the caller window will be updated accordingly.

6.5 Summary

The design and implementation of the tool prototype that supports the BROOD approach was described in this chapter. The tool prototype served two main purposes: it demonstrated the possible automation of the critical activities in the BROOD process

and it provided a means of experimenting and improving the BROOD concepts i.e. the metamodel and templates. The automation of the critical development and evolution activities was particularly achieved by the development of the modules that manage composition, linking, and propagation of business rules to their respective software design components. With regard to the improvement of the BROOD concepts, the developed BROOD tool was found very useful in giving the technical feedbacks that improved the metamodel and templates.

The BROOD tool prototype was developed using the configurable GME modelling environment. The physical metamodels, which is the implementation version of the BROOD metamodel discussed in Chapter 3, were developed for all BROOD models such as rule phrase entries, business rule specification, class diagram, and statechart diagram. A number of simple modelling constraints were also created and attached to the particular metamodel components. The defined metamodels were used in generating the BROOD tool environment. The chosen GME was found very convenient in experimenting with the implementation feasibility and technical aspect of the BROOD metamodel since it is highly configurable in generating a new modelling paradigm.

The BROOD tool prototype provides a number of functionalities in assisting the development and evolution activities in the BROOD process. These functionalities are provided using the GME-generated graphical model editor and the developed form-based modules. Regarding the graphical model editor, the model can be developed by composing parts and connections using the mouse operations. The attributes of the parts and connections can be entered using the provided attribute browser. The graphical model editing is particularly useful for the development of a class diagram and statechart diagram.

With regard to the developed modules, they are provided to maintain rule phrase entries, compose new business rules, and perform business rule changes. The form-based graphical user interfaces of these modules facilitate the composition of a new business rule statement using the existing rule phrases in the rule phrase entries. They also assist the linking of business rules to their related software design components. These modules automate the propagation of the business rule changes to software design via the extensive use of the provided BON framework.

As a conclusion, the BROOD tool simplifies the tedious, error-prone, and time-consuming task of linking and propagating the business rule changes to software design components. A business rule can be changed by changing its rule phrases and the changes are automatically propagated to the related software design components. Apart from that, the BROOD prototype provided useful feedbacks in the improvement of the technical aspect of the BROOD metamodel and templates.

Chapter 7

Conclusions and Further Work

The Business Rule-Driven Object-Oriented Design (BROOD) approach, which is proposed by this research, is evaluated in this chapter. This chapter starts with the summary and the explanation on how this research achieves each objective set at the outset of this research. It is followed by the evaluation of the research results using the comparative evaluation framework introduced in Chapter 2. Subsequently, it summarizes the main contributions of this research to various areas of software development and evolution. Finally, it establishes the areas of future research.

7.1 Research Summary and Achievements

The purpose of this research was to propose a software evolution approach that explicitly considers business rules in the development and evolution of a business software system. This research employed the engineering research method [Basili, 1992] that consists of several steps: analyse the existing approaches, propose a better approach, develop the proposed approach, improve the approach based on the feedbacks from the application of the approach in the selected case study, and repeat the process until no more improvements appear possible. Chapter 2 analysed the existing software evolution and business rule approaches using the developed comparative evaluation framework. The analysis indicated that there is a gap between the approaches in business rule conceptual modelling and the evolvable software systems. Chapter 3 introduced an arguably better approach i.e. the BROOD approach. It explained the product component of the BROOD approach, namely the metamodel that specifies the structure of business rules, software design, and their linking elements. Chapter 4 explained the process component of the approach. The applicability of the BROOD approach in an industrial strength application was demonstrated in Chapter 5 using the MediNET case study. Chapter 6 presented the BROOD tool prototype that demonstrates the capability of automating certain important development and evolution activities of the proposed BROOD approach. The remaining discussion in this chapter explains how the research objectives were tackled by this research.

BROOD has been motivated by the inherent problems in software evolution and the improvement opportunities of the state-of-the-art business rule approaches in software evolution. As mentioned in the first chapter, the hypothesis of this research is restated below:

> The evolution of a business software system may be simplified by a practical holistic approach that (i) explicitly considers business rules in software modelling in addition to the adopted software technologies and (ii) provides a process and tool that facilitate the development and evolution activities.

The above hypothesis is subsequently transformed to the aim and objectives that systematically set the direction of this research. Thus,

the aim of this research is to improve software evolution.

The above aim was broken down into a set of research objectives, which determine the tasks that should be accomplished by this research. The achievements of this research in accomplishing each of the stipulated objectives will be described in the rest of this section.

Objective 1: To analyse the state-of-the-art business rule approaches in software evolution.

The preliminary investigation in the main views on software evolution suggested that a holistic solution to the current evolution problem should consider the sources of changes, i.e. business rules, in addition to software technologies. Business rules were identified as the most frequently changed user requirements and their changes bring the highest impact to both business and software systems. Therefore, analysing the state-of-the-art business rule approaches to software evolution was set as the first objective in this research.

Chapter 2 presented the review of the recent business rule approaches in various research areas, which was performed to achieve the above objective. It reviewed the roles of business rules in different research areas such as software development lifecycle, the alignment of business with its information system, software specification, object-oriented software development, software architecture, and database. From the review, there are two categories of business rule approaches that are closely related to this research: business rule conceptual modelling and evolvable software systems. This research investigated the modelling concepts and techniques of the selected approaches from these two different categories: the BRG, BROCOM, BRS, AOM, Coordination Contract, and BRBeans. The MediNET examples were used to assist the understanding of the selected approaches. The comparative evaluation framework, which is adapted from the common software engineering method evaluation framework, was developed with a great focus on the business rule modelling concepts and software evolution support. This framework was used together with feature analysis technique to evaluate the selected approaches.

The review of the state-of-the-art business rules and software evolution approaches and the evaluation of the selected approaches have provided the constructive observations that lead to the identification of the improvement opportunities to the state-of-the-art approaches to software evolution. The observations help in determining the issues that should be addressed by BROOD. They include both the best practice of the current approaches and the potential solutions to the identified problems. In short, the BROOD approach should:

- provide explicit representation of business rules,
- utilize or extend the existing well-proven widely accepted software technology,
- address the evolution problems at the metamodel level,
- consider pre-implementation evolution,
- include both product and process components, and
- close the gap between the conceptual model of business rule and software components.

Objective 2: To develop a metamodel that externalizes the representation of business rules and provides traceability to their implementation in software design.

As described in Chapter 3, the BROOD metamodel consists of three main parts: business rule, software design, and the linking elements. The business rule part defines the business rule typology as well as the structure of a business rule statement. It also includes other elements that help the business rules organization and management. Two requirements were considered during the development of business rule structure: the business rule statements should be understandable by business users and structured enough to be linked (or implemented) to the related software components. The metamodel was supplemented with EBNF definition that defines the structure and templates of each business rule type.

In terms of software design, BROOD utilizes the benefits of object-oriented paradigm by adopting the widely accepted Unified Modelling Language (UML) to represent software design. The remaining problem is to link the business rule specification to

software design. In addition, the information or changes of each business rule statement should be propagated to its related software design components.

The above problems were tackled by the introduction of the rule phrases as the linking elements. The rule phrases provide the traceability between business rule specification and design components. The examples in section 5.3.4 illustrated the use of rule phrases to link business rules to software design in MediNET. Section 6.3.1 explained the different ways of automatically propagate the business rule information or changes to the related software design components.

Objective 3: To specify a software process that guides the development and evolution of a software system using the proposed metamodel.

Chapter 4 explained the BROOD process i.e. a software process that supports the BROOD approach. SPEM was used to develop a visual model and textual specification of the BROOD process. The BROOD process focuses on three phases i.e. analysis, design, and evolution phases. The analysis and design phases consist of a flow of activities to develop business rule specification and software design models. They also include the way to link business rule specification to software design. The evolution phase contains a flow of activities that may be followed in dealing with simple and complex business rule changes. Apart from the flow of activities, the BROOD process also clearly defines the process roles that performed the specified activities and the work products that should be produced by certain activities. The clear description of these process components makes the BROOD process as a useful guideline for the BROOD users such as software engineers and business users.

Objective 4: To demonstrate the practicability of the proposed approach in an industrial strength software application.

Chapter 5 presented the examples of the application of the BROOD approach in MediNET. MediNET is an industrial strength software application, which is used by different healthcare industries with various frequently changing business rules. With regard to software development, it illustrated the process of systematically transforming the initial set of MediNET business rule statements into the structured business rule

statements (a collection of structured business rule statements is called a business rule specification). It also demonstrated the process of linking each business rule statement to the related software design components. The business rule typology and templates were found useful in assisting the development of the MediNET business rule specification.

Regarding the MediNET evolution phase, the use of BROOD in dealing with simple and complex business rule changes were described using a number of change scenarios. Business rule changes are classified as simple changes if they are concerned with the addition, modification, or deletion of business rules that do not involves the introduction of a complex rule phrase or design component. Otherwise, the changes are classified as complex changes. Simple changes, which are more frequently occurred during software operation, may be performed by business users whilst the implementation of complex changes must be performed by software engineers.

Based on the experience in applying BROOD in MediNET, it was found that the BROOD approach improves the traditional software development approaches in simplifying software evolution. It allows the simple change to be performed by business users, which in turn shorten the change implementation. The software changes that are driven by business rule changes may ensure the alignment of software with user requirements since business rules are also part of user requirements. The application of BROOD against MediNET was also found useful in improving the detailed description of the BROOD metamodel and process.

However, there are two identified BROOD weaknesses: (i) the additional steps of dealing with business rules that increase the complexity and duration of software development process, and (ii) the possible mistakes during the implementation of complex change due to human intervention. These difficulties were already anticipated at the outset of this research, which lead to the inclusion of the process and tool in the BROOD approach.

Objective 5: To develop a software tool prototype that facilitates the key activities of the proposed software process.

At the outset of this research, it was anticipated that the proposed BROOD process is more complex than the software process in the traditional object-oriented software development approaches since it includes additional activities for dealing with business rules. These activities include the construction of business rule statements, the linking of business rules to software design, and the propagation of business rule changes to their related design components. These activities was found tedious, error-prone, and time consuming to be performed manually. Therefore, there is a need for an automated software tool.

Chapter 6 explained the design and implementation of the prototype version of such tool, which is developed using the configurable GME modelling environment. The tool prototype may be used to easily compose a business rule statement using the rule phrases as its building blocks. It also simplifies the linking of rule phrases and business rules to the related design components. As the business rule is changed, the linking information is used by the tool prototype to automatically propagate the changes to the respective software design components.

7.2 Discussion of the Research Results

The BROOD approach is a well-engineered research result that is produced by a systematic software engineering research method. In general the BROOD approach consists of two main components: product and process. The former is a metamodel that defines the typology and structure (or templates) of business rules, which are useful in the capturing and specifying of business rules as well as the linking of business rules to software design. The latter is the descriptions of phases, flow of activities, process roles and work products that guide the way to use the BROOD approach in the development and evolution of a business software system. It also includes the supported tool that facilitates business users and software engineers to perform the difficult activities in the BROOD process. In this section, the BROOD approach will be systematically evaluated using the established comparative evaluation framework, which was used to evaluate the related work in Chapter 2. The observations or lessons learnt from the application on

the case study and the implementation of the case tool provide useful inputs to the evaluation. The availability of the BROOD metamodel and detailed process description supports a more objective evaluation of the BROOD approach. The following discussions are arranged according to the framework components: concepts, modelling language, process, and pragmatics.

- **Modelling Concepts**

  The modelling concepts component of the evaluation framework focuses on three criteria namely business rule definition, typology, and management elements. As described in section 2.3, BROOD adopts the business rule definition proposed by BRG since it covers both business and software perspectives. Moreover, the BRG's definition covers a wide range of business rule types which are often implemented in software system.

  With regard to business rule typology, BROOD introduces three main business rule types: constraints, action assertion, and derivations. These types are further divided into an adequate number of sub-types and templates as described in Chapter 3. In contrast to BRG, BROCOM, and BRS approaches, BROOD attempts to removes the redundancy by reducing the unnecessary business rule types. At the same time, it improves the incompleteness of business rule types in AOM, Coordination Contract, and BRBeans approaches.

  In terms of business rule management elements, BROOD provides ruleset to organize the groups and hierarchy of the closely related business rules. Apart from that, it also has business process and rule owner elements that assist the traceability of business rules to business domain.

- **Modelling Language**

  Unlike other engineering products, software system is an intangible product. The understanding of a software system largely relies on its models, either graphical or textual. Therefore, modelling language is an important component in software development since it determines the quality of the produced software models.

Among the quality criteria of a modelling language are formality, understandability, expressiveness, unambiguity, and evolvability. As mentioned earlier, the availability of the BROOD metamodel and its EBNF definition provide more objective evaluation of the BROOD modelling language. They are respectively presented in Chapter 3 and Appendix B.

The EBNF definition and rule templates correspondingly improve the formality and understandability of the BROOD modelling language. The former provides a formal description of the valid business rule statements, software design models, and linking elements. It is important in providing formality for the development of an automated tool as well as the implementation of business rules in a software system. The latter offers a greater understandability for business users since they are written using rule phrases that use a natural language.

Compared to other related approaches, BROOD has a relatively high level of expressiveness with the availability of an appropriate number of constructs to represent business rules. The keywords in the language definition and enough number of sentence templates may provide a complete representation of the BROOD modelling concepts. However, the total expressiveness of a modelling language to represent business rules is relatively hard to achieve due to the tremendous ways of expressing business rules in a natural language. The extended period of applying BROOD in different areas of businesses may provide feedbacks to improve the expressiveness of the BROOD modelling language.

BROOD is found to have a high level of unambiguity by the introduction of the appropriate typology and templates. BROOD provides a mutually exclusive set of business rule types and removes the superfluous templates in order to avoid the conflict and redundancy in representing the meaning of business rules, which consequently improve the unambiguity of the BROOD modelling language.

Regarding evolvability, BROOD provides rule phrases as the linking elements that establish business rule traceability in software design, which eventually facilitates the evolution of a software system according to business rule changes. The structure of a business rule statement also simplifies business rule changes

since users only need to redefine or change the rule phrases in order to change business rules. In addition, BROOD facilitates the automated propagation of certain business rule changes to software design, which may reduce the evolution efforts.

- **Process**

Apart from modelling language, process is also an important component in ensuring the practicability of the proposed approach. It includes various components such as activities, process roles, and work products that describe the way of applying the modelling concepts and language. Among the evaluation criteria with regard to the process component are lifecycle coverage, process description, coherence, and support for evolution.

With regard to lifecycle coverage, BROOD focuses on three important phases in a software lifecycle i.e. analysis, design, and evolution. These phases consist of the necessary activities in the development and evolution for achieving a more resilient software system. Although the implementation phase is important for a complete lifecycle, it is beyond the scope of this research. BROOD advocates model-driven or architecture-based evolution and assumes the availability of technologies that transform a software design to its implementation components.

Concerning process description, BROOD provides a detailed process description using SPEM as presented in Chapter 4. The description includes common process components such as phases, a flow of activities, process roles, and work products. It is supported by a structured process specification that is included in Appendix C. SPEM metamodel was used in the modelling of the BROOD process to provide an understandable and unambiguous process models.

Relating to coherence criterion, the BROOD process is found to have a high level of coherence since it establishes a clear flow from one activity to another activity in each phase. Moreover, it provides a logical connection between phases using the deliverables (or work products) that are produced by one phase and consumed by another phase.

Regarding support for evolution, BROOD is found superior than other related approaches since it explicitly considers evolution support in its software process. For example, a number of development activities were introduced to establish business rules traceability in software design. During evolution, the BROOD process describes a detailed flow of activities that is purposely introduced to handle simple and complex changes.

- **Pragmatics Aspect**

The pragmatics aspect is concerned with the criteria that influence the acceptance of the proposed approach such as communicability, usability, resources ability, and openness. BROOD possess high communicability by introducing sentence templates to represent business rules and adopting UML to represent software design. The business rule templates simplify the communication by using a natural language, which is understandable by business users. Moreover, the templates are structured enough to be implemented in a software system by software engineers. The use of UML improves the communication among developers since it is well-known and widely-accepted modelling language.

The use of business rule templates and UML also improves the usability of the BROOD approach. The templates allow users to create a business rule statement by simply composing the existing rule phrases whilst UML provides real world abstractions for users to naturally design a software system. Moreover, the detailed process description is provided to guide users especially in performing complex tasks such linking business rules to software design and handling different types of changes.

Regarding resources availability, BROOD provides a medium level of resources since it is a new research result. Some research results that were already converted into commercial products, such as BRS and BRBeans, provide a high number of resources that support their approaches. However, the availability of the case study examples, the tool prototype, and the detailed description of its software process locate BROOD at the same level with other related approaches.

In terms of openness, BROOD follows the benefits and limitations of UML since it uses UML in representing the software design. The implementation of the BROOD software design is not restricted to any programming language.

7.3 Summary of the Main Contributions

BROOD is proposed by this research as a novel approach to software evolution. It improves the traditional software evolution approaches by considering business rules as the sources of changes in addition to addressing the software technology. It closes the gap between the business rule conceptual modelling and the architecture of a software system by providing links between business rule specification and the related design components. The application of BROOD in MediNET strengthens the business domain aspect whilst the implementation of the BROOD tool prototype improves the technical aspect of the BROOD approach. The individual contributions of this research to various areas in the development and evolution of a software system for a rapidly changing business environment will be discussed in the remaining of this section, highlighting the benefits of the conducted research in the context of the related work.

- **Business rules specification**

  One of the main BROOD objectives is to explicitly consider business rules in software evolution. It leads to the development of a metamodel that defines the new structures of business rule specification, which in turn improve the way of representing business rules with regard to software evolution.

  In the area of business rule representation, there are two common issues addressed by the state-of-the-art approaches i.e. the business rule typology and the structure of a business rule statement. In terms of the business rule typology, most prominent approaches [Hay and Healy, 2000; Ross, 2003] attempt to provide a set of complete and mutual exclusive rule types. However, their typology was found redundant and conflicting due to unnecessary and overlapping business rule types. Moreover, they mainly focus on business rule conceptual modelling, which is slightly different from the focus of this research that attempt to link business rules to software design.

With regard to business rule structure, the existing approaches strive to provide an understandable and precise representation of business rules as discussed in section 2.3.1. They represent business rule structure in different ways such as sentence templates [von Halle, 2002; Ross, 2003; Skersys and Gudas, 2004] , mathematical logic [McBrien et al., 1991; Grosof et al., 1999; Antoniou and Arief, 2002], and graphical representation [Halpin, 1996; Lang and Obermair, 1997; Berndtsson and Calestam, 2003].

BROOD extends the state-of-the-art approaches to business rule representation by reducing redundancy and avoiding conflict among business rule types in its typology. The typology was also improved by considering the rule types in the software perspectives, such as their mapping to the structure of a software design, in addition to the existing consideration in the business perspectives.

In terms of the detailed structure of each business rule types, BROOD advocates the use of templates. Similar to the existing approaches, BROOD provides templates that are understandable by business users. However, it enhances the structure of the templates to make them suitable for linking existing approaches to software design in support of future software evolution.

Other than the above contributions, the BROOD metamodel also improves the poor business rule specification in the evolvable software system approaches discussed in section 2.5.2.2. The typology and templates of the BROOD business rule specification guide the capture and representation of business rules. BROOD also contributes to a convenient way to manage user requirements since business rules are also part of requirements. The business rule part of the EBNF definition (see Appendix B) offers a specification for the implementation of business rules either as an independent repository or a linked specification to software design.

- **Object-oriented design specification**

BROOD provides a new way to align object-oriented software design with user requirements via linking business rules to software design and transforming

business rule information into the detailed specification of the related software design components.

As discussed in Chapter 2, object-oriented paradigm is popular in business rule approaches to software evolution due to many benefits such as system understandability, system maintainability, component reusability, and faster development. Some research efforts introduce new object-oriented architectures that consider business rule component in improving software evolvability [Andrade et al., 2002; Yoder and Johnson, 2002]. Other efforts advocate the linking of business rule specification to other modelling components. For example, OCL is used to document a business rule specification, which is linked to the related component of the UML models [Eriksson and Penker, 2000]. Another example related to UML is the extension of use case modelling with the event script that document the business rule specification [Poo, 1999].

The BROOD contribution to object-oriented design lays at the alignment of object-oriented design specification with the frequently changing user requirements i.e. business rules. In this effort, it does not introduce a totally new object-oriented architecture but provide business rules traceability in the existing architecture. With regard to UML, it does not make a major extension to the standard UML methodology. Instead, it strives to use the standard UML to ensure its acceptance since UML is a proven and widely-accepted software development methodology. In addition to the business rule traceability, BROOD also provide a way to transform the information from business rule specification into the detailed specification in design components. Since business rules are part of user requirements, BROOD allows the alignment of user requirements with software system via software design.

- **Software evolution process**

With regard to software evolution, BROOD improves the current software evolution approaches by providing a business rule-driven and user-oriented software development process.

The traditional software evolution approaches focus on the software technology aspects in performing the evolution activities. For example, some leading software evolution approaches such as DRASTIC [Evans and Dickman, 1999], EVOLVE [Liu, 1998], and MORALE [Abowd et al., 1997] emphasize the manipulation of suitable software components in their proposed evolution process. Other approaches [Mens, 2000; Antoniol et al., 2001; Ohlsson et al., 2001] suggest that the software evolution process should concentrate on the management of software artefact. In a purely software technology perspective, they are excellent in simplifying the evolution process. However, they neglect the fact that the business environment, as proposed by the holistic views on software evolution [Perry, 1994; Bennett and Rajlich, 2000], is an important component that greatly influence the software evolution process. In particular, a holistic approach should seriously consider the frequently changing aspect of business environment in developing the technological solution to software evolution. As identified by Chapin et al., business rules are the most frequently changing components in business environment and their changes bring the highest impact on both software and business components [Chapin et al., 2001].

BROOD expands state-of-the-art in the area of software evolution by considering business rules in the evolution process. The linking of explicit business rule specification to the related design components enables rapid evolution of a software system, which is driven by business rule changes. Section 6.4.4 provides an example of the possible automation in propagating the simple business rule changes to software design. BROOD promotes the role of business users in the software evolution process since the common changes may be performed by business users through simply changing the rule phrases. Additionally, BROOD allows the pre-implementation evolution by addressing the evolution problem at software design level.

- **Software development and evolution tool**

BROOD extends the state-of-the-art software developments tools with the capability to maintain business rules, provide business rules traceability and

automate business rule changes to software design. The BROOD metamodel can also be used by business rule management tools in linking their business rule specification to software design.

Most of the existing software evolution tools such as Aspect Browser [Griswold et al., 2001] and Coordination-Contract [Gouveia et al., 2001] were developed to support their proposed approach. Similar to the drawback of their supporting approaches, they only focus on software components. There are recent trends in the commercial software development tools such as Rational Rose and Power Designer [Sybase, 2003] to include business rules in their software design model. However, these tools only allow users to define business rule specification and attach to design component as comments, tags, or constraints. The introduction of business rules brings no effect to software design except the addition of the separate specifications. Another group of related tools are business rule management tools such as ILOG and Blaze Adviser. These tools are very excellent in business rule management but they lack in detailed links to software design.

BROOD provides a metamodel that contributes a theoretical and practical foundation to improve the above three software tool categories. It improves the existing software evolution tools by automatically performing software changes based on business rule changes. It also enhances the commercial software development tools with the 'design for change' approach by providing the way to link business rules to software design. The business rule specification is not only attached to software design component, but its information is also used to add the detailed specification to the related design components. In connection with business rule management tools, BROOD utilizes their excellent business rule specification and improves them by linking the specification to software design for the purpose of future software evolution.

7.4 Issues for Further Research Work

This research work represents an important step towards the development of an approach for supporting the practice of a business rule-driven software evolution.

However, a number of further research works are required to support the full potential of this approach as well as to overcome its current limitations.

- **The elaboration dimension of software evolution**

  Another important issue in software evolution approach is the ability to understand and explain the product or process related to software evolution. The discussion in section 2.2.4 suggested that an ideal approach to software evolution should consider product, process, and elaboration dimensions. This research successfully deals with the product and process dimensions. However, it purposely ignores the elaboration dimension due to time constraint. Therefore, the elaboration techniques for BROOD are the potential areas of further research. As mentioned in section 2.2.3, the common examples of elaboration efforts include metrics [Yang et al., 1997; Li et al., 2000a] and change impact analysis [Deruelle et al., 1999; Zhao et al., 2002; Tahvildari and Kontogiannis, 2004].

  The proposed elaboration techniques should be able to derive metrics such as performance, complexity, coupling, and cohesion of both business rule and software design components based on the input rule change parameters. The techniques should also enable software engineers to anticipate the effect of business rule changes to software design. By extending to the scope of business rule changes, this research effort may improve the current approaches that perform change impact analysis in the scopes of program structure and data changes. Moreover, the information in BROOD metamodel and the availability of the BROOD tool prototype may be considered as the ground work for this further research.

- **The business rule templates**

  The BROOD metamodel defines the rule phrase types that form the business rule statements whilst business rule templates define the organization of these elements that form the valid business rule statements. The number of templates should be adequate in representing all possible forms of business rule statements. At the same time, it should not be redundant in order to avoid conflicts between

two business rule statements. The appropriate number of the business rule templates may only be achieved via the application of BROOD to a larger number of case studies, which may be considered as a necessary further work to strengthen the BROOD approach.

- **Round-trip engineering**

Round-trip engineering is the ability to change the software implementation based on its model and vice versa. Although BROOD extends round-trip engineering by providing the traceability between part of user requirements and software design (or model), it may become more useful if it has the complete round-trip engineering features. Therefore, the further work should provide traceability between software design and source codes. The design specification, which is propagated from business rule changes, should be able to be transformed into software codes and vice versa. In addition, the effect of source code and software design changes on business rule specification should also be studied.

- **The BROOD tool**

The existing BROOD tool is only a prototype version that does not implement the full functionalities of the BROOD approach. It was used to demonstrate the applicability of automating the important aspects of BROOD. Therefore, there are many opportunities to improve the current BROOD tool such as to provide the traceability among different business rules and to handle the effects of deleting certain business rule statements or rule phrases.

Another important aspect of BROOD tool that requires more research efforts is to provide a highly flexible and configurable tool in dealing with a growing number of templates. Users may be allowed to introduce a new templates based on the existing rule phrases defined by the BROOD metamodel. The possible solution is to provide the meta-templates that allow users to define a new template whilst maintaining the business rule traceability for future evolution.

7.5 Concluding Remarks

This research was motivated by the observation that software evolution is inevitable and a key research challenge in software engineering. The important sources of changes, i.e. business rules, were explicitly considered in the proposed BROOD approach to advance the traditional software evolution approaches that focus primarily on improving software technology. The introduced business rule specification, which is defined by the BROOD metamodel, is understandable by business users and well structured enough to be linked to software design. The metamodel also defines the rule phrases that act as the linking elements between business rules and software design. The BROOD process describes the detailed activities of the development and evolution of a business rule-driven software development and evolution.

BROOD improves the state-of-the-art software engineering approaches in both product and process perspectives. Regarding product perspectives, it improves the evolvability of a software system by providing business rules traceability in software design. The introduced business rule specification can be used to explicitly specify the volatile part of user requirements in the user's language. It consequently allows the alignment of the development and evolution of a software design specification with user requirements. Concerning process perspectives, it increases business users' involvement in the development and evolution of a software system, which in turn ensure the fitness for purpose of the developed and evolved software system. The automation of the important tasks in the BROOD process significantly reduces the evolution efforts.

With regard to knowledge transfer activities, the results of this research were published in two international conference proceedings [Wan Kadir and Loucopoulos, 2003; Wan Kadir and Loucopoulos, 2004a] and a well-known journal on software architecture [Wan Kadir and Loucopoulos, 2004b].

References

Abowd, G., Goel, A., Jerding, D. F., *et al.* (1997) MORALE--Mission Oriented Architectural Legacy Evolution, in *Proceedings of the International Conference on Software Maintenance'97*, Bari, Italy, September 29 - October 3.

Alves-Foss, J., Conte de Leon, D. and Oman, P. (2002) Experiments in the Use of XML to Enhance Traceability Between Object-Oriented Design Specifications and Source Code, in *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, IEEE Computer Society, pp. 3959 - 3966.

Anderson, E., Bradley, M. and Brinko, R. (1997) Use Case and Business Rules: Styles of Documenting Business Rules in Use Cases, in *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications*, Atlanta, Georgia, United States, October 6, pp. 85 - 87.

Andrade, L. and Fiadeiro, J. (2000) Evolution by Contract, in *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications 2000, Workshop on Best-practice in Business Rules Design and Implementation*, Minneapolis, Minnesota USA, October 15-19.

Andrade, L. and Fiadeiro, J. (2001) Coordination Technologies for Managing Information System Evolution, in *Proceedings of the 13th Conference on Advanced Information Systems Engineering*, Interlaken, Switzerland, 4-8 June, LNCS 2068, Springer-Verlag, pp. 374-387.

Andrade, L., Fiadeiro, J., Gouveia, J., *et al.* (2002) Separating Computation, Coordination and Configuration, *Journal of Software Maintenance and Evolution: Research and Practice* 14(5), pp. 353-359.

Antoniol, G., Canfora, G., Casazza, G., *et al.* (2001) Maintaining Traceability Links During Object-oriented Software Evolution, *Software : Practice and Experience* 31(4), pp. 331-335.

Antoniou, G. and Arief, M. (2002) Executable Declarative Business Rules and Their Use in Electronic Commerce, in *Proceedings of the ACM Symposium on Applied Computing*, Madrid, Spain, pp. 6-10.

Appleton, D. S. (1984) Business Rules: The Missing Link, *Datamation* 30(16), pp. 145-150.

Arsanjani, A. (2000) Rule Object: A Pattern Language for Adaptable and Scalable Business Rule Construction, in *Proceedings of the 7th. Pattern Languages of Programs Conference*, Monticello, Illinois, USA, August 13-16.

Astley, M. and Agha, G. A. (1998) Modular Construction and Composition of Distributed Software Architectures, in *Proceedings of the Int. Symposium on Software Engineering, for Parallel and Distributed Systems*, Kyoto, Japan, 20-21 April, IEEE Computer Society.

Bajec, M. and Krisper, M. (2001) Managing Business Rules in Enterprises, *Elektrotehniški vestnik* 68(4), p. 236–241.

Basili, V. R. (1992) The Experimental Paradigm in Software Engineering, in *Proceedings of the International Workshop on Experimental Software Engineering Issues: Critical Assessment and Future Directions*, Dagstuhl Castle, Germany, 14 - 18 September, Springer-Verlag, pp. 3 - 12.

Bennett, K. (1996) Software Evolution: Past, Present and Future, *Information and Software Technology* 38(11), November, pp. 673-680.

Bennett, K. H. and Rajlich, V. T. (2000) Software Maintenance and Evolution : A Roadmap, in *Proceedings of the Conference on the Future of Software Engineering*, Limerick, Ireland, 4-11 June, ACM Press, pp. 73-87.

Berndtsson, M. and Calestam, B. (2003) Graphical Notations for Active Rules in UML and UML-A, *ACM SIGSOFT Software Engineering Notes* 28(2), March.

Brinkkemper, S. (1996) Method Engineering: Engineering of Information Systems Development Methods and Tools, *Information and Software Technology* 38(4), April, pp. 275-280.

Bubenko, J. A. and Wangler, B. (1993) Objectives Driven Capture of Business Rules and Information Systems Requirements, in *Proceedings of the IEEE Conference on Systems, Man and Cybernetics*, Le Touquet France, 17-20 October, pp. 670 - 677.

Burd, E. and Munro, M. (1999) An Initial Approach Towards Measuring and Characterising Software Evolution, in *Proceedings of the 6th Working Conference on Reverse Engineering*, Los Alamitos, CA, USA, IEEE Computer Society, pp. 168-74.

Chapin, N., Hale, J. E., Khan, K. M., *et al.* (2001) Types of software evolution and software maintenance, *Journal of Software Maintenance and Evolution: Research and Practice* 13(1), pp. 3-30.

Cibrán, M. A., D'Hondt, M., Suvée, D., *et al.* (2003) JAsCo for Linking Business Rules to Object-Oriented Software, in *Proceedings of the International Conference on Computer Science, Software Engineering, Information Technology, e-Business and Applications*, Rio De Janeiro, Brazil, 5-7 June, pp. 1-7.

Dam, K. H. and Winikoff, M. (2003) Comparing Agent-Oriented Methodologies, in *Proceedings of the Fifth International Bi-Conference Workshop on Agent-Oriented Information Systems (at AAMAS '03)*, Melbourne, 14 July.

Demuth, B. and Hussmann, H. (1999) Using UML/OCL Constraints for Relational Database Design, in *Proceedings of the 2nd International Conference on The Unified Modeling Language*, pp. 598-613.

Deruelle, L., Bouneffa, M., Goncalves, G., *et al.* (1999) Local and Federated Database Schemas Evolution - an Impact Propagation Model, in *Proceedings of the 10th International Conference in Database and Expert Systems Applications DEXA'99 (LNCS Vol.1677) . 1999, pp.902-11.*, Berlin, Germany, Springer-Verlag, pp. 902-911.

Diaz, O., Iturrioz, J. and Piattini, M. G. (1998) Promoting Business Policies in Object-Oriented Methods, *Journal of Systems and Software* 41(2), May 1998, pp. 105-115.

Eriksson, H.-E. and Penker, M. (2000) *Business Modeling with UML: Business Patterns at Work*. New York, USA, John Wiley & Sons, Inc.

Erlikh, L. (2000) Leveraging Legacy System Dollars for e-Business, *IEEE IT Professional* 2(3), May-June, pp. 17 - 23.

Evans, H. and Dickman, P. (1997) DRASTIC: A Run-Time Architecture for Evolving, Distributed, Persistent Systems, in *Proceedings of the 11th European Conference on Object-Oriented Programming*, Lisbon, Portugal, June 14-18, Springer-Verlag, pp. 244-75.

Evans, H. and Dickman, P. (1999) Zones, Contracts and Absorbing Change: An Approach to Software Evolution, in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '99)*, Denver, Colorado, USA, October, 34, ACM, pp. 415-434.

Fanta, R. and Rajlich, V. (1999) Restructuring Legacy C code into C++, in *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society, pp. 77-89.

Finkelstein, A. and Kramer, J. (2000) Software Engineering : A Roadmap, in *Proceedings of the Conference on the Future of Software Engineering*, Limerick, Ireland, 4-11 June, ACM Press, pp. 3-22.

Floyd, C. (1986) A Comparative Evaluation of System Development Methods, in *Proceedings of the IFIP WG 8.1 Working Conference on Information Systems Design Methodologies: improving the practice*, Noordwijkerhout, Netherlands, North-Holland Publishing Co., pp. 19 - 54.

Fu, G., Shao, J., Embury, S. M., *et al.* (2001) A Framework for Business Rule Presentation, in *Proceedings of the 12th International Workshop on Database and Expert Systems Applications*, Munich, Germany, pp. 922-926.

Gamma, E., Helm, R., Johnson, R., *et al.* (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Garlan, D. (2000) Software Architecture : A Roadmap, in *Proceedings of the Conference on the Future of Software Engineering*, Limerick, Ireland, 4-11 June, ACM Press, pp. 91-101.

Gottersdiener, E. (1997) Business Rules Show Power, Promise, *Application Development Trends*, March, 1997.

Gouveia, J., Koutsoukos, G., Andrade, L., *et al.* (2001) Tool Support for Coordination-Based Software Evolution, in *Proceedings of the TOOLS Europe*, IEEE Computer Society Press, pp. 184-196.

Griswold, W. G., Yuan, J. J. and Kato, Y. (2001) Exploiting the Map Metaphor in a Tool for Software Evolution, in *Proceedings of the 23rd International Conference on Software Engineering*, Los Alamitos, USA, May, IEEE Computer Society, pp. 265-274.

Grosof, B. N., Labrou, Y. and Chan, H. Y. (1999) A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML, in *Proceedings of the 1st ACM Conference on Electronic Commerce*, Denver, Colorado, United States, pp. 68 - 77.

Groznik, A. and Kovacic, A. (2002) Business Renovation: From Business Process Modelling to Information System Modelling, in *Proceedings of the 24th International Conference on Information Technology Interfaces*, 1, pp. 405-409.

Grubb, P. and Takang, A. A. (2003) *Software Maintenance: Concepts and Practice.* Singapore, World Scientific Publishing.

Haggerty, N., Wall, J. and Etten, v. (2001) Defining the Requirements for a Business Rule Repository, *The Data Administration Newsletter*, April.

Halpin, T. (1996) Business Rules and Object Role Modeling, *Database Programming & Design* 9(10), October, pp. 66-72.

Halpin, T. (2001) Augmenting UML with Fact-orientation, in *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, Maui, Hawaii, 03 - 06 January, 3.

Hars, A. and Marchewka, J. T. (1996) Eliciting and Mapping Business Rules to IS Design: Introducing a Natural Language CASE Tool, in *Proceedings of the Decision Sciences Institute*, Orlando, Florida, 24-26 November, 2, pp. 533-535.

Hay, D. and Healy, K. A. (2000) Defining Business Rules ~ What Are They Really?, Technical Report Rev 1.3, the Business Rules Group.

Herbst, H. (1996) Business Rules in Systems Analysis: A Meta-model and Repository System, *Information Systems* 21(2), April 1996, pp. 147-166.

Herbst, H. (1997) *Business Rule-Oriented Conceptual Modeling*. Germany, Physica-Verlag.

Herbst, H., Knolmayer, G., Myrach, T., *et al.* (1994) The Specification of Business Rules: A Comparison of Selected Methodologies, in *Proceedings of the IFIP Working Group 8.1 Conference CRIS 94*, University of Limburg, Maastricht, Elsevier, pp. 29-46.

Ho, I., Komiya, Z., Pham, B., *et al.* (2003) An Efficient Framework for Business Software Development, in *Proceedings of the International Conference on Cyberworlds*, 3-5 Dec., pp. 336 - 343.

Hong, S., van den Goor, G. and Brinkkemper, S. (1993) A Formal Approach to the Comparison of Object-Oriented Analysis and Design Methodologies, in *Proceedings of the 26th Hawaii International Conference on System Sciences*, 5-8 Jan., 4, IEEE Publication, pp. 689 - 698.

Hruby, P. (1998) Mapping Business Processes to Software Design Artifacts, in *Proceedings of the European Conference on Object-Oriented Technology (ECOOP'98)*, Berlin, Germany, Springer-Verlag, pp. 234-6.

Hurlbut, R. R. (1998) *Managing Domain Architecture Evolution Through Adaptive Use Case and Business Rule Models*, PhD Thesis, Illinois Institute of Technology.

Hürsch, W. L. and Seiter, L. M. (1996) Automating the Evolution of Object-Oriented Systems, in *Proceedings of the Second JSSST International Sysmposium on Object Technologies for Advanced Software*, Berlin, Germany, Springer-Verlag, pp. 2-21.

I.B.M. (2003) *IBM WebSphere Application Server Enterprise,* ver 5.0.2, International Business Machine Corporation.

IBM (2003) *IBM WebSphere Application Server Enterprise,* ver 5.0.2, International Business Machine Corporation.

Itou, K. and Katayama, T. (2000) Evolutionary Development of Object Behaviors, in *Proceedings of the International Symposium on Principles of Software Evolution*, Kanazawa, Japan, 1-2 November, IEEE Comput. Soc., pp. 68-77.

Jacobson, I., Booch, G. and Rumbaugh, J. (1999) *The Unified Software Development Process*. Massachusetts, Addison Wesley Longman.

Jarzabek, S. and Hitz, M. (1998) Business-oriented and Component-based Software Development and Evolution, in *Proceedings of the International Workshop on Large-Scale Software Composition*, Vienna, Austria, August 28.

Kang, S., Jang, M. and Sohn, J. (2004) Design of Rule Object Model for Business Rule Systems, in *Proceedings of the 6th International Conference on Advanced Communication Technology*, 2, pp. 818-822.

Kardasis, P. (2001) *Business Rule Modelling*, PhD Thesis, Dept. of Computation, UMIST.

Kardasis, P. and Loucopoulos, P. (2003) Managing Business Rules During the Requirements Engineering Process in Rule-Intensive IT Projects, in *Proceedings of the 6th International Conference on Business Information Systems (BIS 2003)*, Colorado Springs, Colorado, USA, 4-6 June, pp. 239-247.

Kardasis, P. and Loucopoulos, P. (2004) Expressing and Organising Business Rules, *Information and Software Technology* 46(11), pp. 701-779.

Khan, K. M., Kapurubandara, M. and Chadha, U. (2004) Incorporating Business Requirements and Constraints in Database Conceptual Models, in *Proceedings of the First Asian-Pacific Conference on Conceptual Modelling*, Dunedin, New Zealand, 31, Australian Computer Society, Inc., pp. 59 - 64.

Kon, F., Gill, B., Anand, M., *et al.* (2000) Secure Dynamic Reconfiguration of Scalable CORBA Systems with Mobile Agents, in *Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000*, Proceedings (LNCS Vol.1882). Springer-Verlag, pp. 86-98.

Kovacic, A. (2004) Business Renovation: Business Rules (Still) the Missing Link, *Business Process Management Journal* 10(2), pp. 158-170.

Kovari, P., Diaz, D. C., Fernandes, F. C. H., *et al.* (2003) *WebSphere Application Server Enterprise V5 and Programming Model Extensions: WebSphere Handbook Series*, International Business Machines Corporation.

Lam, W., Loomes, M. and Shankararaman, V. (1999) Managing Requirements Change Using Metrics and Action Planning, in *Proceedings of the Proceedings of the Third European Conference on Software Maintenance and Reengineering*, Los Alamitos, CA, USA, IEEE Comput. Soc., pp. 122-128.

Lang, P. and Obermair, W. (1997) Modeling Business Rules with Situation/Activation Diagrams, in *Proceedings of the 13th International Conference on Data Engineering*, Birmingham, UK, 7-11 April, IEEE, pp. 455-464.

Layzell, P. J. and Loucopoulos, P. (1988) A Rule-Based Approach to the Construction and Evolution of Business Information Systems, in *Proceedings of the 4th IEEE International Conference on Software Maintenance*, Phoenix, Arizona, USA, October, pp. 258-264.

Ledeczi, A., Maroti, M., Bakay, A., *et al.* (2001a) The Generic Modeling Environment, in *Proceedings of the Workshop on Intelligent Signal Processing*, Budapest, Hungary, 17 May.

Ledeczi, A., Volgyesi, P. and Karsai, G. (2001b) Metamodel Composition in the Generic Modeling Environment, in *Proceedings of the 15th European Conference on Object-Oriented Programming*, University Eötvös Loránd, Budapest, Hungary, June 18-22.

Lehman, M. M. (1997) Laws of Software Evolution Revisited, in *Proceedings of the European Workshop on Software Process Technology '96*, October, LNCS 1149, Springer Verlag.

Lehman, M. M. and Belady, L. A. (1985) *Program Evolution: Processes of Software Change.* London, Academic Press Inc.

Leite, J. C. S. and Leonardi, M. C. (1998) Business Rules as Organizational Policies, in *Proceedings of the 9th International Workshop on Software Specification and Design*, Ise-Shima, Japan, 16-18 April, pp. 68-76.

Leonardi, M. C. and Leite, J. C. S. (2002) Using Business Rules in Extreme Requirements, in *Proceedings of the 14th International Conference on Advanced Information Systems Engineering*, Toronto, Canada, 27-31 May, 2348, Springer-Verlag.

Li, W. (1999) On Managing Classes for Evolving Software, in *Proceedings of the Proceedings of the Seventh International Workshop on Program Comprehension*, Los Alamitos, CA, USA, IEEE Comput. Soc., pp. 144-150.

Li, W., Etzkorn, L., Davis, C., *et al.* (2000a) An Empirical Study of Object-Oriented System Evolution, *Information and Software Technology* 42(6), 15 April 2000, pp. 373-438.

Li, Y., Yang, H. and Chu, W. (2000b) Generating Linkage between Source Code and Evolvable Domain Knowledge for the Ease of Software Evolution, in *Proceedings of the Proceedings of the International Symposium on Principles of Software Evolution*, Los Alamitos, CA, USA, IEEE Comput. Soc., pp. 196-205.

Lied, R. (1997) Domain Engineering Experiences in the 5ESS Switch, in *Proceedings of the Proceedings of the ISS'97 : World Telecommunications Congress. `Global Network Evolution: Convergence or Collision?'* Toronto, Ont., Canada, 2, Pinnacle Group, pp. 533-538.

Lientz, B. P. (1983) Issues in Software Maintenance, *ACM Computing Surveys* 15(3), September, pp. 271-278.

Liu, K. and Ong, T. (1999) A Modelling Approach for Handling Business Rules and Exceptions, *Computer Journal* 42(3), pp. 221-231.

Liu, L. (1998) EVOLVE: Adaptive Specification Techniques for Object-oriented Software Evolution, in *Proceedings of the 31st Hawaii International Conference on System Sciences*, 5, IEEE Computer Society, pp. 396-405.

Loucopoulos, P. and Layzell, P. J. (1989) Improving Information System Development and Evolution Using a Rule-Based Paradigm, *Software Engineering Journal* 4(5), pp. 259-267.

Loucopoulos, P., Theodoulidis, B. and Pantazis, D. (1991) Business Rules Modelling : Conceptual modelling and Object-Oriented Specifications, in *Proceedings of the IFIP WG8.1 Working Conference on the Object-Oriented Approach in Information Systems*, Quebec City, Canada.

Magee, J., Dulay, N. and Regis, J. (1994) Regis: A Constructive Development Environment for Distributed Programs, *Distributed Software Engineering Journal* 1(5).

Masuhara, H., Sugita, Y. and Yonezawa, A. (2000) Dynamic Compilation of a Reflective Language using Run-Time Specialization, in *Proceedings of the Proceedings of the International Symposium on Principles of Software Evolution*, Kanazawa, Japan, 1-2 November, IEEE Computer Society, pp. 128-137.

McBrien, P., Niezette, M., Pantazis, D., *et al.* (1991) A Rule Language to Capture and Model Business Policy Specification, in *Proceedings of the CAiSE 91*, Trondheim, Norway.

McCullough, D., Korelsky, T. and White, M. (1998) Information Management for Release-based Software Evolution using EMMA, in *Proceedings of the Proceedings of the 10th International Conference on Software Engineering and Knowledge Engineering (SEKE '98)*, Knowledge System Institute, IL, USA.

Mehta, A. and Heineman, G. T. (2001) Evolving Legacy System Features using Regression Test Cases and Components, Technical Report WPI-CS-TR-01-14, Computer Science Department, Worcester Polytechnic Institute.

Mens, K., Wuyts, R., Bontridder, D., *et al.* (1998) Workshop Report - ECOOP'98 Workshop 7: Tools and environments for business rules, in *Proceedings of the ECOOP'98 Workshops, Demos, and Posters*, Brussels, Belgium, 20-24 July, Springer, pp. 189-196.

Mens, T. (2000) Conditional Graph Rewriting as a Domain-Independent Formalism for Software Evolution, in *Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance*, LNCS Vol. 1779, Springer, pp. 127-43.

Mens, T. and D'Hondt, T. (2000) Automating Support for Software Evolution in UML, *Automated Software Engineering* 7(1), pp. 39-59.

Misra, A., Karsai, G. and Sztipanovits, J. (1997a) Model-integrated Development of Complex Applications, in *Proceedings of the 5th International Symposium on*

Assessment of Software Tools and Technologies, IEEE Computer Society Press, pp. 14-23.

Misra, A., Long, E. and Sztipanovits, J. (1997b) Evolutionary Design for Manufacturing Execution Systems, in *Proceedings of the World Manufacturing Congress*, Auckland, New Zealand, November.

Mohan, P., Yussuff, S., Crichlow, J., *et al.* (2000) A Rule Language for Specifying and Reusing Object-Oriented Business Rules, in *Proceedings of the IASTED International Conference on Software Engineering and Applications*, Las Vegas, NV, USA, 6-9 Nov., IASTED/ACTA Press, pp. 285-290.

Morgan, T. (2002) *Business Rules and Information Systems : Aligning IT with Business Goals.* Boston, MA, Addison-Wesley.

Moriarty, T. (1993) The Next Paradigm, *Database Programming and Design*, February.

Ohlsson, M. C., Andrews, A. A. and Wohlin, C. (2001) Modelling Fault-proneness Statistically over a Sequence of Releases: A Case Study, *Journal of Software Maintenance and Evolution: Research and Practice* 13(3).

OMG (2001) *OMG Unified Modeling Language Specification,* ver 1.4, Object Management Group.

OMG (2002) *Software Process Engineering Metamodel Specification,* ver 1.0, Object Management Group.

Oreizy, P., Medvidovic, N. and Taylor, R. N. (1998) Architecture-Based Runtime Software Evolution, in *Proceedings of the International Conference on Software Engineering 1998 (ICSE'98)*, Kyoto, Japan, April 19-25, pp. 177-186.

Perkins, A. (2002) Business Rules = Meta Data, *Business Rules Journal* 3(No. 1), Jan 2002.

Perrochon, L. and Mann, W. (1999) Inferred designs, *IEEE Software*,

Perry, D. E. (1994) Dimensions of Software Evolution, in *Proceedings of the International Conference on Software Maintenance*, Los Alamitos, CA, USA, IEEE Comput. Soc. Press, pp. 296-303.

Perry, D. E. (1999) Software Evolution and `Light' Semantics, in *Proceedings of the International Conference on Software Engineering*, Los Angeles, CA, ACM.

Perry, D. E. and Wolf, A. L. (1992) Foundations for the Study of Software Architecture, *Software Engineering Notes* 17(4), October 1992, pp. 40-52.

Petrounias, I. and Loucopoulos, P. (1994) A Rule-based Approach for the Design and Implementation of Information Systems, in *Proceedings of the 4th International Conference on Extending Database Technology*, Cambridge, UK, 28-31 March, pp. 159-172.

Poo, D. C. C. (1999) Events in Use Cases as a Basis for Identifying and Specifying Classes and Business Rules, in *Proceedings of the 29th Conference on Technology of Object-*

Oriented Languages and Systems, Nancy, France, 7-10 June, IEEE Comp Soc, pp. 204-213.

Ramsey, F. V. and Alpigini, J. J. (2002) A Simple Mathematically Based Framework for Rule Extraction using Wide Spectrum Language, in *Proceedings of the Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on*, pp. 44-52.

Riehle, D., Tilman, M. and Johnson, R. (2000) Dynamic Object Model, Technical Report WUCS-00-29, Dept. of Computer Science, Washington University.

Rosca, D. and D'Attilio, J. (2001) Business Rules Specification, Enforcement and Distribution for Heterogeneous Environments, in *Proceedings of the 25th Annual International Conference on Computer Software and Applications*, pp. 3-9.

Rosca, D., Greenspan, S. and Wild, C. (2002) Enterprise Modeling and Decision-Support for Automating the Business Rules Lifecycle, *Automated Software Engineering* 9(4), pp. 361 - 404.

Rosca, D. and Wild, C. (2002) Towards a Flexible Deployment of Business Rules, *Expert Systems with Applications* 23(4), 2002/11/, pp. 385-94.

Ross, R. G. (2003) *Principles of the Business Rule Approach.* Boston, USA, Addison-Wesley.

Ross, R. G. and Lam, G. S. W. (2001) RuleSpeak™ Sentence Templates: Developing Rule Statements Using Sentence Patterns, Technical Report, Business Rule Solutions, LLC.

Ross, R. G. and Lam, G. S. W. (2003) *The BRS Proteus™ Methodology*, Business Rule Solutions.

Rouvellou, I., Degenaro, I., Rasmus, K., *et al.* (1999) Externalizing Business Rules from Enterprise Applications: An Experience Report, in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Denver, Colorado, November 1-5.

Rouvellou, I., Degenaro, L., Diament, J., *et al.* (2004) Business Users and Program Variability: Bridging the Gap, in *Proceedings of the Eighth International Conference on Software Reuse*, Madrid, Spain, 5-9 July.

Rouvellou, I., Degenaro, L., Rasmus, K., *et al.* (2000) Extending Business Objects with Business Rules, in *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe 2000)*, Mont Saint-Michel/ St-Malo, France, June, IEEE Computer Society Press, pp. 238-249.

Saranauwarat, S. and Taniguchi, H. (2000) Operating Systems Support for the Evolution of Software: An Evaluation Using WWW Server Software, in *Proceedings of the Proceedings International Symposium on Principles of Software Evolution*, Kanazawa, Japan, 1-2 November, IEEE Comput. Soc., pp. 292-301.

Schneider, J. (1999) *Components, Scripts, and Glue : A Conceptual Framework for Software Composition*, PhD Thesis, Ins. of Computer Science and Applied Mathematics, University of Bern.

Senivongse, T. (1999) Enabling Flexible Cross-Version Interoperability for Distributed Services, in *Proceedings of the Intl. Symp. on Distributed Objects and Applications (DOA'99)*, Edinburgh, UK.

Shao, J. and Pound, C. J. (1999) Extracting business rules from information systems, *BT Technology Journal* 17(4), Oct 1999, pp. 179-186.

Shaw, M. and Garlan, D. (1996) *Software Architecture : Perspective of an emergance discipline*. New Jersey, Prentice Hall Inc.

Shibayama, E., Toyoda, M., Shizuki, B., *et al.* (2000) Design Issues of Visual Languages for Supporting Software Evolution, in *Proceedings of the Proceedings of the International Symposium on Principles of Software Evolution*, Kanazawa, Japan, 1-2 November, IEEE Comput. Soc., pp. 241-248.

Siau, K. and Rossi, M. (1998) Evaluation of Information Modeling Methods: A Review, in *Proceedings of the Proceedings of the 31st Hawaii International Conference on System Sciences*, Kohala Coast, HI USA, 6-9 Jan., 5, pp. 314 - 322.

Skersys, T. and Gudas, S. (2004) Business Rules Integration in Information Systems Engineering, in *Proceedings of the 13th International Conference on Information Systems Development: Advances in Theory, Practice and Education*, Vilnius, Lithuania, 9-11 September, Kluwer Publishing, pp. 252-263.

Snoeck, M. (2002) Sequence Constraints in Business Modelling and Business Process Modelling, in *Proceedings of the 4th International Conference on Enterprise Information Systems*, Ciudad Real, Spain, 2-6 April, pp. 683-690.

Stiemerling, O., Kahler, H. and Wulf, V. (1997) How to Make Software Softer - Designing Tailorable Applications, in *Proceedings of the DIS (Designing Interactive Systems) '97*, Amsterdam, August 17-20, ACM Press, pp. 365-376.

Sturm, A. and Shehory, O. (2003) A Framework for Evaluating Agent-Oriented Methodologies, in *Proceedings of the Fifth International Bi-Conference Workshop on Agent-Oriented Information Systems (at AAMAS '03)*, Melbourne, 14 July.

Svahnberg, M. and Bosch, J. (2000) Issues Concerning Variability in Software Product Lines, in *Proceedings of the Third International Workshop on Software Architectures for Product Families*, Springer Verlag.

Sybase (2003) *PowerDesigner OOM User's Guide,* ver 9.5.2, Sybase, Inc.

Tahvildari, L., Gregory, R. and Kontogiannis, K. (1999) An Approach for Measuring Software Evolution Using Source Code Features, in *Proceedings of the IEEE Asia-Pacific Software Engineering (APSEC'99)*, Takamatsu, Japan, December, pp. 10-17.

Tahvildari, L. and Kontogiannis, K. (2004) Requirements Driven Software Evolution, in *Proceedings of the Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pp. 258-259.

Truex, D. P., Baskerville, R. and Klein, H. (1999) Growing Systems In Emergent Organisations, *Comm. Of ACM* 42(8), August 1999.

Vives, F. and Dombiak, G. (2000) Implementing Business Rules in a Flexible Architecture, in *Proceedings of the OOPSLA Workshop Best Practices in Business Rule Design and Implementation*, Minnesota, USA, October 15-19.

von Halle, B. (2001a) Building A Business Rule System, *Data Management Review*, January.

von Halle, B. (2001b) Powered by Rules, *Business Rules Journal* 2(10), October.

von Halle, B. (2002) *Business Rule Applied: Building Better Systems Using Business Rules Approach.* New York, John Wiley & Sons, Inc.

VU (2003) *GME 3 User's Manual,* ver 3.0, Institute for Software Integrated System, Vanderbilt University.

Wan Kadir, W. M. N., Kumoi, R., Katmon, N., *et al.* (2000) A Web-based Healthcare Information Systems, in *Proceedings of the Seminar on Accounting and Information Technology*, UUM, Sintok, Malaysia, 13-14 November.

Wan Kadir, W. M. N. and Loucopoulos, P. (2003) Relating Evolving Business Rules to Software Design, in *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, Las Vegas, Nevada, USA, 23-26 June, pp. 129-134.

Wan Kadir, W. M. N. and Loucopoulos, P. (2004a) Linking and Propagating Business Rule Changes to IS Design, in *Proceedings of the International Conference on Information System Development*, Vilnius, Lithuania, 9-11 September.

Wan Kadir, W. M. N. and Loucopoulos, P. (2004b) Relating Evolving Business Rules to Software Design, *Journal of Systems Architecture* 50(7), pp. 367-382.

Wang, X., Sun, J., Yang, X., *et al.* (2004) Business Rules Extraction from Large Legacy Systems, in *Proceedings of the 8th European Conference on Software Maintenance and Reengineering*, 24-26 March, pp. 249-258.

Yadav, S. B., Bravoco, R. R., Chatfield, A. T., *et al.* (1998) Comparison of Analysis Techniques for Information Requirement Determination, *Communications of the ACM* 31(9), August.

Yang, H., Luker, P. and Chu, W. C. (1997) Measuring Abstractness for Reverse Engineering in a Re-Engineering Tool, in *Proceedings of the Proceedings International Conference on Software Maintenance*, Los Alamitos, CA, USA, IEEE Comput. Soc., pp. 48-56.

Yoder, J., Balaguer, F. and Johnson, R. (2001a) The Architectural Style of Adaptive Object-Models, in *Proceedings of the 15th European Conference on Object-Oriented Programming*, University Eötvös Loránd, Budapest, Hungary, June 18-22.

Yoder, J. W., Balaguer, F. and Johnson, R. (2001b) Adaptive Object Models for Implementing Business Rules, in *Proceedings of the Third Workshop on Best-Practices for Business Rules Design and Implementation, Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, Tampa Bay, Florida, USA, October 14-18.

Yoder, J. W. and Johnson, R. (2002) The Adaptive Object Model Architectural Style, in *Proceedings of the Proceeding of The Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 '02)*, Montreal, Canada, August 25-31.

Zhao, J., Yang, H., Xiang, L., *et al.* (2002) Change Impact Analysis to Support Architectural Evolution, *Journal of Software Maintenance and Evolution: Research and Practice* 14(5).

Appendix A

MediNET - The Case Study

This appendix provides a description of MediNET application. MediNET is used as a case study to illustrate and experiment the concepts proposed by this thesis. It was chosen as a case study because it is needed to be flexible for use by different businesses with various frequently changing business rules. This appendix starts with the overview of MediNET. In the following sections, the descriptions on MediNET business processes, entities, and rules are presented. In simple terms, business process is defined as a set of activities to accomplish certain business goals. Business entities represent the concrete persons or things, and the abstract concepts relevant to MediNET. The entities contain information that is consumed, stored or produced by MediNET. Business rules are statements that guide, determine, and constraint the way of performing business tasks.

A.1 MediNET Overview

MediNET is provided by an Application Service Provider (ASP) to serve the healthcare community by offering a suite of Internet applications. The ASP is a solution provider who rents information technology applications to users and charges them based on the number of performed transactions. The benefits are many, including: access from anywhere there is an Internet connection, less maintenance to worry about, easier and more frequent software enhancements, and of course, lower costs. Users only need to pay as and when they use the application.

MediNET addresses the administrative and back-end processing requirements of the healthcare business community i.e. both the healthcare providers and their paymasters. It is an Internet-based, role-specific suite of applications which is intended to act as a secondary layer to the existing administrative and information systems. It allows various components of the healthcare industry to exchange business data instantaneously and automate their routine administrative tasks. Therefore, facilitated businesses are able to reduce their administrative burdens, become more efficient and make better informed business decisions.

In general, MediNET users can be divided into three categories: paymasters, healthcare providers (HCPs), and supplier. Paymasters are those who pay for medical or healthcare services, for examples employers, insurers and managed care organizations. Healthcare providers are the professionals who dispense medical treatment, for examples general practitioners (GPs), hospitals and dentists. However, the current implementation of MediNET, and the scope of this case study, is only limited to employers as the paymasters and GPs as the

HCPs. The supplier is obviously the company who owns, provides and maintains the MediNET applications.

Paymasters use MediNET to maintain the basic parts of the patient records. For example, a paymaster (employer) may register his payee (employee) as a panel patient of his panel clinics. The employer may decide to what extent the employee is entitled to treatment. Additionally, the employer is also allowed to view medical certificates issued to his employees as well as view the invoices issued to him.

HCPs use MediNET to manage patient records, patient billing and paymaster invoicing. Each HCP has a number of clinics located at different sites and offer various medical services. MediNET allows the clinic staff to validate a patient's eligibility for medical treatment and, if applicable, determine the limit of bill amount set by his employer. The clinic may also register a new panel patient with certain conditions, for example with the presentation of a confirmation letter from the patient's employer. The bill is issued as the consultation is completed. The bills are automatically categorised and collated into appropriate invoices to be delivered to the relevant paymasters.

The supplier uses MediNET to perform system administrative tasks such as information maintenance and application usage invoicing. It maintains user accounts and manages most information related to HCPs and paymasters. Using MediNET, the supplier may also calculate the charges and produce the invoices of the respective customers.

It is important to note that MediNET is neither an Electronic Medical Record (EMR) application nor a clinical diagnostic tool. Rather, MediNET is merely an administrative back-end tool that addresses the business processes of its users. Although MediNET is able to record some aspects of the patient's interaction with the clinic, it is not designed to discharge the functionalities of a complete EMR system. With regard to GPs, MediNET is not specifically designed to help them in diagnosis. It helps GPs to manage the administration of their transactions.

A.2 The Business Processes

There are three main business processes in MediNET: registration, billing, and invoicing. In brief, there are two type of registration i.e. patient registration and consultation registration. Patient registration can be done in one of two locations, either at a healthcare provider (HCP) or paymaster location. Paymaster HR Officer may register paymaster payee as a panel patient where the paymaster pays the patient's bill. HCP Clinic Assistant may register both cash and panel patients. HR Officer and Clinic Assistant are the examples of MediNET users which are described in section A.3.4.

Each time a patient visits HCP clinic for medical services, the patient should register for consultation and his name is then put in a queue. When the consultation is completed, HCP

Clinic Assistant will issue a bill based on the doctor's prescription. Cash patients must pay their bills, whilst the bills for panel patients are paid by their paymaster – the panel patient bills are later included in a paymaster's invoice. However, each bill is verified by the HCP Account Clerk before it is automatically sorted and inserted into invoice as an invoice item. Finally, the invoice is sent to the paymaster. These processes are illustrated in Figure A-1.

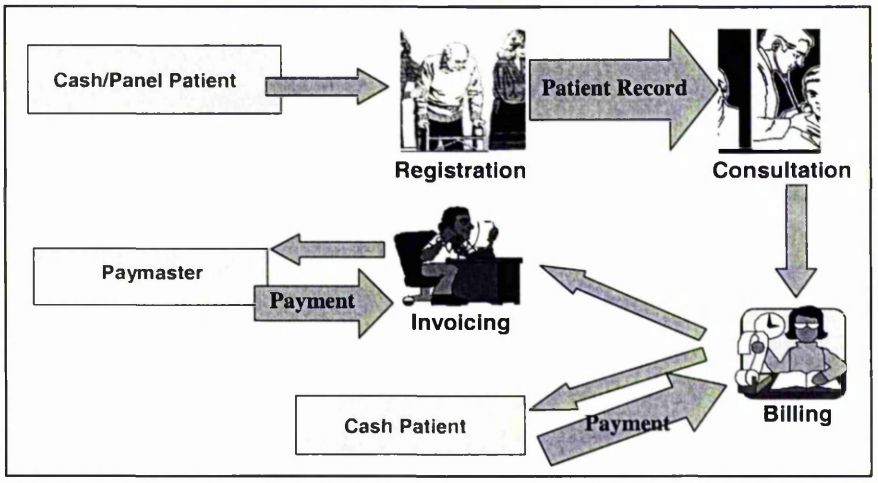

Figure A-1 The MediNET Main Business Processes

Apart from the above processes, there are other business processes which are related to information management, payment handling, and various report generation. They are discussed at the end of this section.

A.2.1 Registration

Patient Registration: Each person must be registered as a patient before he can receive any service from HCP. The registration can be done either at HCP or paymaster location. At the HCP location, HCP Clinic Assistant may register a person either as a cash or panel patient. If the person wants to register as a panel patient he must show an official letter from his paymaster, and the paymaster must be a registered paymaster. At the paymaster location, HR Officer may register his staff as panel patients. HR Officer may terminate any patient from his list of payees, for example, when the patient is no longer employed by the paymaster. In this case, the status of the patient will be changed from panel to cash patient. Although he can terminate the patient from being a panel patient, the HR Officer cannot delete the patient record, since all the patient records also belong to the HCP. Only the MediNET System Administrator has the authority to delete the records. During registration, each patient is assigned a unique patient registration number. If required, the patient will be directed to the consultation registration.

Consultation Registration: All registered patients must register for consultation each time they visit the HCP for medical services. The consultation registration is used to ensure the patient

eligibility via online patient validation as well as to prepare necessary patient information prior to the consultation. HCP Clinic Assistant input the supplied patient registration number or any other acceptable search text. The input is automatically verified using the information available in the MediNET system. Next, the verified patient is put in a consultation queue. The patient details including queue number, patient name, paymaster name, and maximum bill amount is printed and given to the patient. The Clinic Assistant uses the shelf location reference number which is obtained from the patient record to get the physical patient's medical record. The medical record will be given to the doctor. For a panel patient, the status (active, blocked, or archived) of his paymaster is also verified during consultation registration. The patient is allowed to choose the paymaster if he has more than one paymaster. If the paymaster status indicates that it is currently banned from HCP services, the patient will be informed that he must pay the bill himself or change to another paymaster. As the consultation registration is completed, the patient will be directed to the waiting area.

A.2.2 Billing

Bill Preparation: After a consultation is completed, the doctor will issue prescriptions for the patient. The prescriptions contain the drug list that should be dispensed to the patient and the medical services which were given to him. Based on the prescriptions and patient information, HCP Clinic Assistant creates a bill that includes patient information, consultation descriptions, drug descriptions and the amount details. In addition, it also includes paymaster code (for panel patient), issue date, and issue staff. The bill will be given to the patient together with his drugs and a copy will be kept for future processing. The panel patient's bill will be later used to create a paymaster invoice.

Bill Modification: There is a possibility of mistakes made by HCP Clinic Assistant during bill creation. In this case, the HCP Shift Leader may directly perform correction to the incorrect bill. The bill can only be modified by the HCP Shift Leader working in the same shift with the HCP Clinic Assistant who creates the bill. However, to prevent fraud, the HCP Shift Leader cannot modify the printed or paid bills.

Bill Payment: Upon receiving the payment from a patient, HCP Clinic Assistant records the method and amount of payment. The payment method can be cash, cheque, or credit card. The bill state is changed from 'unpaid' to 'fully paid' or 'partly paid' depending on the amount paid by patient. This function is only applied to cash or partially sponsored panel patient since paymaster will pay for the fully sponsored panel patient. For the panel patient, the bill state will be changed to 'invoiced' when the bill is inserted into invoice.

A.2.3 Invoicing

Bill Verification: HCP Account Clerk assorts and verifies every copy of the actual printed bill given to panel patients against the respective bill recorded in MediNET before it is inserted into an invoice. Traditionally, the verification task is to ensure that bill amount is not exceeding the limit set by the panel company. It is also used to ensure the actual amount charged to the patient

corresponds to the amount recorded in the system. After the bills are verified, they will be sorted and stored in the paymaster billing list for preparing an invoice.

HCP Service Invoice Preparation: HCP normally sends its service charge invoices to the paymasters at the end of each month. However, the invoices can be created at any time and all verified bills can be daily inserted as invoice items into the invoices by HCP Account Clerk in order to avoid back-log at the end of month. Initially, the invoice state is set to 'active'. The invoice items cannot be inserted after the invoice state is set to 'published'. The published invoice may be viewed by the paymaster if the paymaster also subscribed to MediNET. However, HCP may need to print the invoice and sent it manually if the paymaster is not subscribed to MediNET. HCP Account Clerk may also send the invoice weekly or fortnightly according to earlier arrangement between HCP and the paymaster. This arrangement also determines the end date of each invoice, which is set during the invoice creation process.

Invoice Viewing and Printing: A paymaster Account Officer is allowed to view and print a completed invoice that belongs to his company provided that he has an access to the system. The details of each invoice item vary from one paymaster to another according to the paymaster requirements. In other words, the viewing and printing function should be flexible and customizable in terms of information to be included for each invoice item. For examples, paymaster A may be want to display only patient name and bill amount for each invoice item whilst paymaster B needs to include date and time information. After viewing or printing the invoice, paymaster may decide either want to pay or reject the invoice.

Modification of Rejected Invoice: The invoice may be rejected due to a number of reasons such as the listed patient is not a staff, or the bill or invoice amount is exceeding the earlier stated limit. HCP Account Clerk is responsible to modify the rejected invoices according to the reasons of rejection supplied by paymaster. The invoice will be issued back to the panel company.

Invoice Payment: Upon receiving of a payment, HCP Account Clerk will record the payment information. Using balance forward method, the payment is then automatically allocated to the oldest unpaid invoice. After the first invoice is settled, it continues on to the second oldest invoice if the cash remains unapplied using a FIFO type algorithm. Payment allocation is made flexible since one or more invoices can be paid using one or more payments. Thus, there can be a single payment for one invoice, single payment for more than one invoices, or multiple payments for one invoice. The payment records are used to keep track the credit history of the paymaster which in turn provides necessary information for producing the outstanding balance report.

MediNET Usage Invoicing: The supplier (ASP) produces invoices on MediNET usage to both HCPs and paymasters. HCPs are charged based on the following two-tier fee structure:

Monthly Subscription Fee	**RM 50.00** per month
Transaction Fee	**RM 0.15** per transaction

For example, let assume that a General Practice that uses the MediNET application to treat a total of 400 transactions for the month of February 2004. Thus, the fee would be calculated as follows:

No	Description	Calculation	Fee Payable (RM)
1	Monthly Subscription Fee	RM 50.00 x 1 month	50.00
2	Transaction Fee	RM 0.15 x 400 transactions	60.00
		Total fee for the month	**110.00**

For the paymasters, they will be charged based on the number of their payees. The following table shows the current fee structure for the paymasters:

Employee + Dependant Amount Range (person) *		Monthly Subscription Fee (RM)
Min	Max	
1	25	35
26	50	65
51	75	95
76	100	125
101	150	165
151	200	200
201	300	250
301	400	300
401	500	375
501	600	450
601	700	525
701	800	600
801	900	675
901	1000	725
1001	~	negotiable

A.2.4 Other Business Processes

Apart from the main processes illustrated in Figure A-1, there are a number of other important business processes in MediNET system. They are related to the service provider functions such as MediNET usage invoicing and information management. The description of these additional business processes are organised into MediNET system administration, HCP-related, and paymaster-related processes.

MediNET System Administration Processes

System Login: In order to ensure the system security, each user is given a unique user ID and a password. The user must enter his user ID and password during system login. The login information is verified to check for staff position and user access rights, and relevant subsystems will be started upon the successful system login. User is only allowed to access the granted subsystems according to their rights. The password can be changed whenever it is necessary.

MediNET User Administration: MediNET users can be categorized into supplier, HCP, and paymaster. In short, MediNET System Administrator (Sys Admin) is the super user who creates

the administrator user accounts for HCP and paymaster. The HCP and Paymaster Sys Admin maintain user information such as user ID, password, name, description, user type of his respective users. User type is used to determine the access rights granted to each user. For example, paymaster HR Officer is only allowed to register and view patients from his/her own company. User administration function is only accessible by Sys Admin. Please refer to section A.3.4 for more information on MediNET users.

HCP and Paymaster Records Administration: Another important responsibility of MediNET Sys Admin is the management of HCP and paymaster information. This task includes the addition, deletion, and modification of HCP and paymaster records. All HCPs and paymasters must register with MediNET before they can use the applications. Upon registration, they will be given an administrator account to manage their own users.

HCP-related Processes

Medical Item Record Maintenance: Medical item refers to any types of consultation, medical services, and drugs, for examples X-Ray, CT scan, paracetamol, and specialist consultation. The information about medical item includes item code, description, type code, and measure unit. By maintaining this information, HCP Clinic Assistant may check the list of medical items available in HCP as well as set default amount, measurement unit, and unit price.

Panel Record Maintenance: Upon receiving a manual or online application from a paymaster, HCP may decide either to approve or reject the application. If the application is accepted, HCP Sys Admin will create a new panel paymaster for his HCP. During system operation, the status of a paymaster is set as follows:

- '*active*': It is set as an initial value during the creation of paymaster record. This status indicates that the paymaster does not have any problem with outstanding balances. It is also set when the paymaster cleared any pending payments after being blocked by HCP.
- '*blocked*': It will be set when a paymaster has a bad payment record according to the current rules determined by HCP. Once the paymaster is blocked, their payees is not permitted to obtain medical services from the HCP using the paymaster bill. Instead, they have to foot their bill on their own.
- '*archived*': When a paymaster decide to terminate HCP from being its panel clinic, or HCP itself wish to terminate the paymaster, the paymaster status is set to 'archived'. The change of the status should be preceded by an official letter. The payment records of the paymaster will be stored as an archive until the grace period elapsed and, most importantly, until the paymaster settle all of its debts.

Accounting Reports Preparation: The system should be able to produce three main accounting reports namely daily expenses, monthly and outstanding balance reports. Daily expenses report is originally generated and completed for each clinic by an HCP Clinic Assistant. The examples

of common expenses are allowances for locum tenens such as service, food, and stationeries allowances. This report must also include the amount of petty cash received from the main office. The daily transaction report is verified by HCP Shift Leader at the end of each work shift. The information in the daily transaction report will be used to generate monthly report, which is generated by an Account Clerk. Outstanding balance report is used to keep track the invoice outstanding balance of a paymaster. The Account Clerk may keep track the credit history of a paymaster by generating the invoice outstanding balance report. The information is used to issue reminders.

Documenting and Reporting Dispensed Drugs: Each time HCP Clinic Assistant creates a bill, based on doctor's prescriptions, he often insert a number of drugs which are dispensed to the patient. The dispensed drug list should be automatically recorded for future drug listing. According to government regulations for private medical practitioners, the dispensed drug listing is required to be produced at any time as requested by State Health Department.

Paymaster-related Processes

HCP Appointment/Termination: Paymaster may apply to appoint a new HCP or terminate existing HCP using online application form.

Displaying Medical Certificates (MCs): A paymaster HR Officer is allowed to view or check the MCs issued to his employees by HCPs. The MCs can be retrieved using their employee's ID or name. The date range can also be specified for displaying the MCs.

Healthcare Expenses Report: Paymaster generates healthcare expenses report to understand the trends in medical benefit spending.

A.2.5 The Modular Design of MediNET Software System

With regard to the current implementation, MediNET is divided into three main subsystems i.e. myPeople, myHCP, and myMediNET. myPeople and myHCP were implemented as web-based application to fulfil the distributed nature of their users. The former is used by paymasters to maintain their payee information, view MC and invoices, select or remove their HCPs, and generate healthcare-related reports. The latter is used by HCP to perform patient registration, billing, invoicing, information management, and report generation. myMediNET is used by the supplier to prepare application usage invoices, maintain HCP and paymaster records, and various report generation. These subsystems and their modules are shown in Figure A-2.

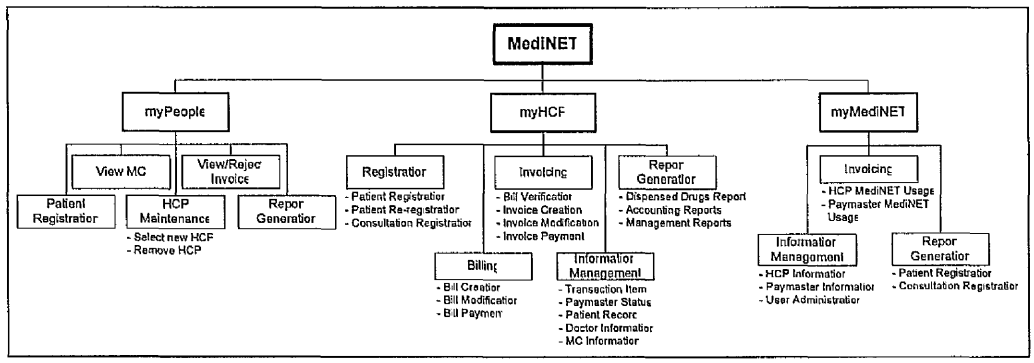

Figure A-2 The Structure of MediNET Implementation

A.3 The Business Entities

A number of entities which is relevant to MediNET were identified in the description of the above business processes.

A.3.1 The Main Business Entities

There are three main business entities in MediNET i.e. HCP, paymaster, and patient. These entities iniatiate and perform the business processes.

HCP provides medical services to the registered patients. It has at least one clinic (branch). Each clinic may register zero or more patients. HCP can be appointed as a panel HCP by several paymasters, and the paymasters are called panel paymasters for that HCP. Each HCP and clinic is given a unique ID. Among other details that should be kept to describe an HCP are name, address, phone, fax, contact person, and e-mail. HCP may have one or more HCP users. Each user has a unique ID, password, and user type. A more detailed discussion on MediNET users are made in section A.3.4.

Paymaster may appoint zero or more HCPs to be its panel HCPs. It also has zero or more employees registered as the patients of the clinics under its panel HCPs. Each paymaster is given a unique ID, and other details such as name, address, phone number, e-mail, fax, and contact person are stored in the paymaster record. The paymaster status is initially set to active. The paymaster may also choose to specify the limit of patient's bill and invoice amount.

Patient is registered to one or more clinics as a cash or panel patient. If the patient is registered as a panel patient, he should have at least one paymaster as a payer. Each patient must have a unique patient registration number (PRN). Other details such as full name, address, date of birth, sex, blood type, allergy, phone number, registration date, and nationality are stored in the patient record. A patient may have a number of dependants which in turn can be registered as patients. Most paymasters pay patient's dependant bills. However, guardian information must be included in dependant claims. Thus, each dependant should include his guardian PRN. The shelf location of the patient's medical record is also created during patient registration. For a

cash patient who wants to re-register as a panel patient, a new registration is not needed. It is sufficient for HCP Clinic Assistant to only change the patient status. The patient status can also be changed from 'panel' to 'cash' when the patient is no longer paid by his paymaster.

A.3.2 Billing-related Entities

During billing, there are a number of entities which are consumed or produced such bill, bill item, bill payment, transaction item, and medical certificate.

A bill is issued to a patient after the consultation is completed. Each bill is given a unique bill number where there will be no same bill number within any clinic. Apart from patient information, clinic ID, staff ID, and issue date are also included in a bill.

The prescriptions supplied by a doctor are used to insert the bill items. There is at least one bill item contained in each bill. The quantity of bill item is supplied to calculate its cost; the amount of the bill is calculated as the sum of all bill items. The bill item has an item code which refers to a specific transaction item. Transaction items are grouped under several transaction types. Among the examples of transaction types are consultation, medication, ward charges and X-Ray. Each transaction item has an item code to retrieve details on a particular transaction item. The details include measure unit, service tax, and default amount.

A cash patient bill must be paid by the patient. The amount, date and type of payment are recorded for future reference. For a panel patient, a bill is paid by his paymaster. The bill is later inserted into invoice as an invoice item. In this case, the payment is received as an invoice payment. If applicable, the medical certificate is issued to the patient.

A.3.3 Invoicing-related Entities

In general, there are two types of invoices in MediNET i.e. HCP service and MediNET usage invoices. HCP service invoice is issued by HCP to claim a payment from paymaster for medical services given to its payees. MediNET usage invoices can be further categorised into two types: HCP MediNET usage and paymaster MediNET usage invoices. They are used by the supplier to claim the usage of the MediNET from HCPs and paymasters respectively. HCP Service invoice contains all bills issued to the paymaster's payees for the selected date range. Different paymaster chooses different level of details. For example, some paymasters would like to have patient's name, address, billing date, and bill amount displayed in their invoices whilst others might be to see only the patient registration number, patient's name and amount. MediNET usage invoice is calculated based on the table given in section A.2.3.

Regardless of its type, an invoice can be associated with zero or more payment. A payment can be allocated to pay one or more invoices. The amount of payment is recorded and a payment number is assigned to each payment. A payment can be made using different methods: cheque, cash, credit card or account transfer. Other information that depends on payment type should be recorded for example, reference number, bank name, and payment date. Reference number is

either transaction or cheque number. Since payment can be used to pay one or more invoices, it is necessary to always keep the balance updated.

A.3.4 MediNET Users

MediNET users are categorised into three user groups i.e. supplier, paymaster, and HCP. Each user group consist of different types of users according to the users' roles or based on the nature of their responsibility in the respective organization. User type is used to determine the access rights granted to the user. For example, paymaster Human Resource (HR) Officer is only allowed to register and view patients from his/her own company. The categorization of MediNET users is shown in Figure A-3.

There are two types of supplier users: MediNET System Administrator (Sys Admin) and Account Staff. MediNET Sys Admin is responsible to create and manage paymaster and HCP Sys Admins. These newly created system adminsitrator accounts will in turn create and manage other paymaster and HCP users respectively. The MediNET Sys Admin is also responsible to perform system administrative task such as performance tuning and data backup. The Account Staff mainly involves with the monitoring and preparation of MediNET usage invoices for paymasters and HCPs.

Paymaster users are given the authority to access the system remotely from their office via the internet. Among the user type from this category are Paymaster HR Officer and Paymaster Account Officer. It is important to note that the role names used to represent the user types might be different from the post names used by paymasters. The Paymaster HR Officer is responsible to add, modify or terminate their employees as the panel patients in MediNET system. This type of user also has the privilege to view the medical certificate (MC) list that has been certified to their employees based on date range or staff ID. The Paymaster Account Officer's role is to deal with the invoices issued by HCPs. The invoices contain the cost of medical services given to their staffs. This type of users can view and print the invoices.

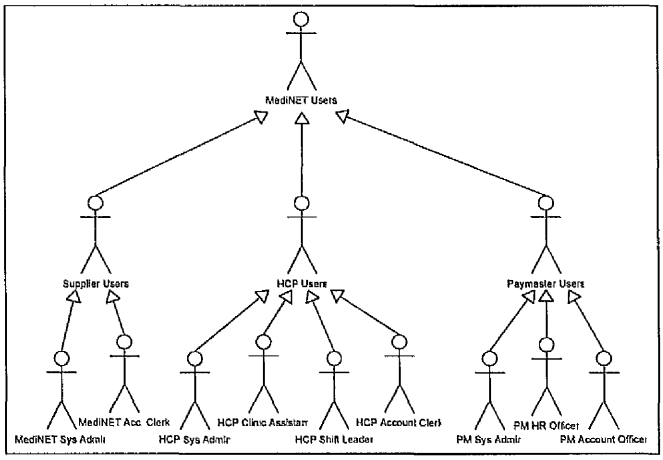

Figure A-3 MediNET Users

With regard to HCP user group, there are four user types in this group namely HCP Sys Admin, HCP Clinic Assistant, HCP Shift Leader and HCP Account Clerk. If the HCP is medical group practices, HCP Clinic Assistant and HCP Shift Leader are located at each general practice (GP) whilst the HCP System Administrator and HCP Account are located at the headquarters.

HCP Sys Admin is responsible to control most of the stored information such as HCP user accounts, general practice records, transaction item records, and patient records. HCP Account Clerk is dealing with the preparation and issuing of the paymaster invoices, and updating of the record of payment and payment allocation. This type of users also has the responsibility to deal with the Paymaster Account Officers regarding any rejected invoices, and make the modification to the invoice when necessary.

HCP Clinic Assistant works in different shifts. She registers walk-in patients that haven't registered in the MediNET database as well as performing a consultation registration. She also issues the bills to the patients, generates daily transaction report, and maintains the record of dispensed drugs. For each shift, there will be one HCP Shift Leader who responsible for every operation performed by the clinic assistant during her shift. The Shift Leader has the authority to access all functionalities available to the clinic assistant, with additional privileges to modify any incorrect bills, and to verify and modify the daily transaction report.

A.3.5 Data Stored in MediNET

The physical database design of MediNET application is shown in Figure A-4 at the end of this appendix. It is important to mention that the diagram in Figure A-4 only includes tables relevant to the scope of the case study.

A.4 The Informal Business Rule Statements

Based on the MediNET descriptions described in the previous sections and the information from other sources, the initial set of business rules were identified. These business rules were initially written as a set of informal business rule statements. The examples of the informal business rule statements, which were organized according to their business process, are listed below:

A.4.1 Registration

Patient Registration

- Each patient must be assigned a unique registration number.
- Each patient must supply name, date of birth, address, and gender.
- If a patient want to register as a panel patient, the patient must supply the document from the paymaster that consist of important information such as paymaster name, paymaster address, employee number, employee level, and department name.
- A patient may have more than one paymaster.
- If a patient wants to register as a dependant of a panel patient, the patient must show the document from his guardian's paymaster and supply the PRN of his guardian.

Panel Patient Eligibility

- Panel patient may only register with the clinics from the panel HCPs of his paymaster.
- If the healthcare benefit coverage stated that the panel patient may only register at the selected clinics, the panel patient must only register at the selected clinics. Otherwise, the patient may register at any clinics belong to his paymaster's panel HCP.
- The number of panel clinics registered by patient must not exceed the maximum number of clinics allowed by his paymaster.

Patient Consultation

- Each patient must be registered with the HCP before they can register for consultation.
- Any patient with an outstanding balance should be banned from consultation registration.
- If the patient has more than one paymaster, the patient must choose one paymaster to pay the bill.
- If the patient is a panel patient, then the requested statement must be in the list of his eligible entitlements.
- If the patient is a panel patient, then the allowed treatments must be in the list of eligible procedures or entitlements of his healthcare benefit.
- At the end of each successful consultation registration, the patient is inserted into the consultation queue.
- The patient should be entered to the consultation queue in time sequence, or first-come first-serve basis. However, if the patient is an emergency, he may skip the queue.
- If the patient suffers life-threatening problem upon registration, the patient is considered as an emergency patient.

A.4.2 Billing

Bill Preparation

- When the consultation is completed, a bill is created for the patient.
- A patient bill has one or more bill items.
- Each bill item is computed as the unit amount multiply by the quantity. Some items have a service tax value which is needed to be added to the item amount.
- Each bill item is associated with an item from the clinic transaction items.
- The bill amount is calculated as the sum of amounts of all bill items.
- If a patient is a cash patient, the balance will be initialised to the value of the bill amount and the bill status is set to unpaid.
- If a patient is a panel patient and his paymaster pays the bill in full, the balance is set to 0 and the bill status is set to unpaid.
- If the paymaster is partially pay the bill, the balance is calculated as the bill amount minus the maximum bill amount set by the paymaster.
- HCP may ask the patient to pay or simply absorb the balance based on the earlier agreement with the paymaster. For cash and partially paid panel patients, the balance of a bill is updated when the payment is made.
- The balance of a bill is computed as the amount of the bill minus the amount of the payment.
- A bill can be modified only if the user role is Chief Clinic Assistant.

- The amount of a panel patient's bill must not exceed the maximum bill amount set by the paymaster.

Bill Payment

- If the paymaster is blocked or the patient is a cash patient, the patients must foot the bill on their own.

At the end of billing process, some rules applied to the flow of bill information.

- If the bill was issued to a panel patient, the bill copy must be verified and sorted to be processed as a panel transaction item. Panel transaction items are important to create both paymaster and application usage invoices.
- If the bill was issued to a cash patient, the patient must pay the bill. The bill mistakes can be rectified before the payment is made. However, it can only be modified by HCP Shift Leader.

A.4.3 Invoicing

Invoice Preparation

- Each bill must be verified before it is inserted into an invoice.
- When the bill is inserted into an invoice, the bill status is set to 'invoiced'.
- The invoice status must be set to close at 12:00am on the next day after the invoice end date.
- If the panel's invoice interval is monthly, the end date is set to the end of the month.

Calculation of Invoice Amount

Different rules are applied in calculating the amount of different invoices.

- For paymaster MediNET usage invoice, the amount is calculated based on the number of the paymaster's payees according to the table in section A.2.3.
- For HCP MediNET usage invoice, the amount is calculated as the amount of transaction fees, which are calculated as the transaction fee multiply by the total number of transactions, plus the monthly fee. The current rate of HCP usage fees were shown by the table in section A.2.3.
- For HCP service invoice, the amount is calculated as the total of the payees' bill amounts.

Invoice Rejection and Modification

- The invoice can be rejected if the patient is not a payee, the amount of any invoice item (bill) is beyond the bill limit, or the amount of invoice is beyond the invoice limit.
- When the account clerk receive the rejection request, if the rejection request is accepted, then the invoice state is set to 'rejected'.

Invoice Payment and Payment Allocation

- When the payment is allocated to the invoice and there is no more invoice balance, the invoice state is set to 'paid'.
- Each payment is allocated to the oldest unpaid invoice(s) using a FIFO type algorithm.
- Invoices can be paid by several payments.

Past Due Invoices and Reminder

- The first reminder will be sent if a payment is not received within 30 days from the invoice date.

- If the payment is not received after 60 days from the invoice date, the Account Clerk will issue the second reminder.
- If there is still no payment after 90 days from the invoice date, the paymaster information will be sent to the System Administrator. At this point, System Administrator blocks the panel company from receiving any service from the HCP.
- A paymaster (panel company) is under probation if the paymaster has an invoice with category 1 past due and the current balance is more than RM 5,000.00.

A.4.4 Other Business Rules

Paymaster and HCP Registration

- If a paymaster chooses to set the maximum invoice limit, the paymaster must select the pre-defined limit. However, the paymaster must pay the monthly fee which is calculated as 4000 divide by the pre-defined limit.
- Due to the important of the stored information, MediNET imposes some rules in the deletion of patient, paymaster, and HCP records. The patient, paymaster, and HCP records can only be deleted by MediNET System Administrator after their status is set to 'archived' for particular grace period. The grace period must be set to at least 5 years.

Patient Record Maintenance

- Paymaster HR Officer may terminate any patient from their list of payees.
- When a paymaster terminates a patient from being his payee and the number of paymaster is greater than zero, the number of paymaster is decreased by one.

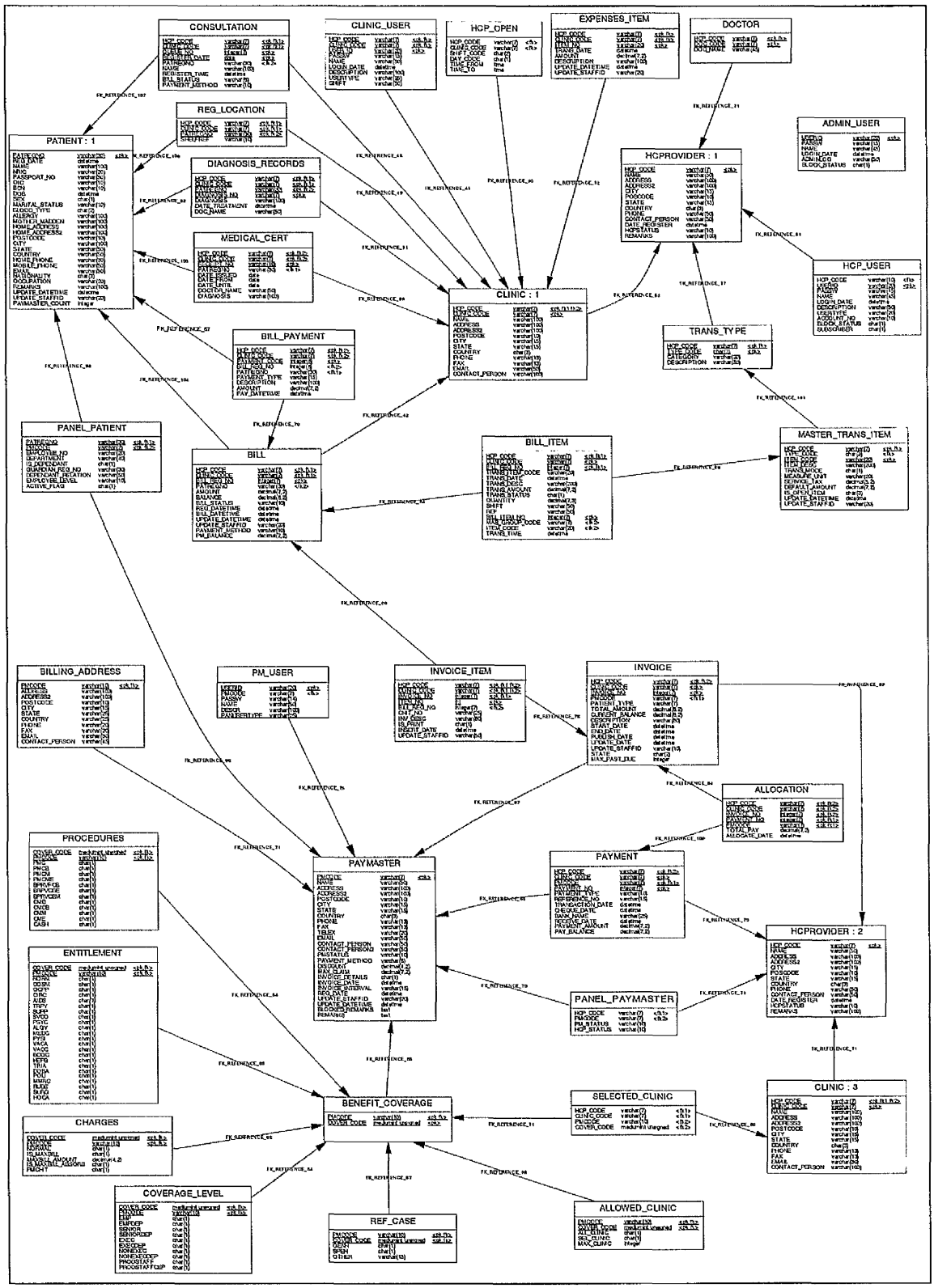

Figure A-4 MediNET Physical Database Design (the latest version)

Appendix B

The EBNF Specification for the BROOD Metamodel

B.1 The semantics of EBNF (ISO/IEC 14977) syntax

x	x (unquoted text) is a non-terminal symbol
'x'	x (quoted text) is a terminal symbol
=	definition symbol
,	concatenation symbol
|	definition separator symbol
[x]	x appears at maximum once
{x}	x appears zero or more times
n * x	x appears n times
(* *note* *)	*note* is a comment

Note:

- Symbol priority from the highest to the lowest: * , | = ;
- All symbols are case sensitive.
- Since there exist similar terms in both business rule and software design specifications, and some terms from the former are used in the later (and vice versa), the different conventions for non-terminal symbols are used. Underscore character is used to separate words in business rule syntax definitions, whilst capital letter is used as the first letter of each word in UML Class Diagram and State Machine Diagram definitions.

B.2 Business rule syntax using EBNF

Business rule organisation and typology

```
business_rule_model = rule_set, owner;
rule_set            = (rule_set | rule_statement), {rule_set | rule_statement},
                      [owner], [business_process];
business_rule       = (constraint | action_assertion | derivation), name,
                      [is_mandatory], [priority], [is_propagatable];
```

Contraint

```
constraint          = att_constraint | rel_constraint;
```

Attribute Constraint

```
att_constraint      = entity, ('must have' | 'may have'), ['a unique'], att_term
                      | att_term, ('must be' | 'may be'), relational_op, (value |
```

```
                    att_term)
                    | att_term, 'must be in', list;
att_term          = attribute, 'of', entity;
```

Relationship Constraint

```
rel_constraint    = ( [cardinality], entity, 'is a/an', role, 'of', [cardinality],
                    entity
                    | [cardinality], entity, 'is associated with', [cardinality],
                    entity
                    | entity, ('must have' | 'may have'), [cardinality], entity
                    | entity, 'is a/an' entity ),
                    {Association};
```

Action Assertion

```
action_assertion  = 'WHEN', event, ['IF', condition], 'THEN', action,
                    {StatechartDiagram, Transition};
```

Event

```
event             = simple_event | complex_event;
simple_event      = (change_event | time_event | user_event), {Class, Operation};
change_event      = att_term ('is updated' |) |
                    entity ('is deleted' | 'is created')
                    (operation | business_rule), 'is triggered';
time_event        = date_time |
                    n, time_unit, 'time interval from', date_time , 'is reached' |
                    number, time_unit, 'after', date_time;
user_event        = string;
complex_event     = simple_event, {('Or' | 'And'), simple_event {('Or' | 'And'),
                    simple_event}
```

Condition

```
condition         = simple_condition | complex_condition;
simple_condition  = ['Not'], attribute_term, relational_op, (value | attribute_term)
                    | attribute_term, ('in' | 'not in'), list;
complex_condition = simple_condition, ('Or' | 'And'), simple_condition, {('Or' |
                    'And'), simple_condition};
```

Action

```
action            = simple_action | action_sequence;
simple_action     = trigger_action | object_ manipulation_action | user_action,
                    {Class, Operation};
trigger_action    = 'trigger', (process | operation | business_rule);
object_manipulation_action = 'set', att_term, 'to', value |
                             ('create' | 'delete'), object;
action_sequence           = simple_action, {simple_action};
```

Derivation

```
derivation        = computation | inference;
```

Computation

```
computation          = attribute_term, 'is computed as', algorithm, {Class, Operation};
algorithm            = string;
                       (* i.e. any specification language for specifying the algorithm
                          e.g. OCL, pseudo-code, etc.   *)
```

Inference

```
inference            = 'If', condition, 'then', fact;
fact                 = (attribute_term | entity), relational_op, ['a'], value ) |
                       entity, ('may' | 'may not'), action, {Class, Operation};
```

Rule Phrases / Linking Elements / Low-Level Definitions

```
(*  Some low level non-terminal symbol such as <real>, <integer> and <string>
    are not defined.                                                    ****   *)
entity               = phrase, Class;
attribute            = phrase, Class, Attribute;
operation            = phrase, {Class, Operation};
cardinality          = phrase, maxCard, minCard;
eventPhrase          = phrase, event, {Class, Operation};
actionPhrase         = phrase, action, {Class, Operation};
role                 = string;
list                 = string,{string};
phrase               = string;
value                = string | integer | real | date | time;
number               = real | integer;
time_unit            = 'second' | 'minute' | 'hour' | 'day' | 'month' | 'year';
relational_op        = 'equal' | 'not equal' | 'less than' | 'less than or equal' |
                           'greater than' | 'greater than or equal';
name                 = string;
priority             = 'high' | 'medium' | 'low';
is_mandatory         = boolean;
is_propagatable      = boolean;
boolean              = 'true' or 'false';
```

B.3 UML Class Diagram syntax using EBNF

```
(*
    ********************************************************************************
    ****
    The following definitions were derived from OMG Unified Modeling Language
    Specification (Ver 1.5). There is no intention for producing a complete and
    precise EBNF definitions for UML Class Diagram or Statechart Diagram, and they
    are almost impossible to achieve using EBNF alone. The following definitions are
    merely used to formally experiment and demonstrate the technical aspects of
    linking business rules to software design elements.
    ********************************************************************************
    *  *)

ClassDiagram          = name, {Note | TaggedValue | Stereotype | Constraint | Class |
                        Generalization | Association);
```

Core Package

```
Class                 = {Comment | TaggedValue | Stereotype | Constraint}, name,
                        [IsActive], [IsAbstract], {Attribute | Operation};

IsActive              = boolean;

IsAbstract            = boolean;

Interface             = {Comment | TaggedValue | Stereotype | Constraint}, name,
                        {Attribute | Operation};

AssociationClass      = {Comment | TaggedValue | Stereotype | Constraint}, name,
                        AttachedAssociation, {Attribute | Operation};

AttachedAssociation = string;
```

Attribute

```
Attribute             = {Comment | TaggedValue | Stereotype | Constraint }, name, Type,
                        Visibility, InitialValue, Changeability, Multiplicity,
                        [IsActive[, [IsAbstract], {Attribute | Operation}, isMandatory,
                        isUnique;

Visibility            = 'private' | 'public' | 'protected';

Changeability         = boolean;

Initial_value         = string;

Multiplicity          = string;

isMandatory           = boolean;

isUnique              = boolean;
```

Operation

```
Operation             =  {Comment | TaggedValue | Stereotype | Constraint}, Name,
                        [Visibility], [ReturnType], [Concurrency], [IsAbstract],
                        [IsLeaf], [IsRoot], {Parameter};

Parameter             = {Comment | Note | TaggedValue | Stereotype | Constraint}, Name,
                        [Type], [DefaultValue], [Kind];

ReturnType            = string;

Concurrency           = string;

IsAbstract            = boolean;
```

```
IsLeaf          = boolean;
IsRoot          = boolean;
OwnerScope      = string;
```

Generalization

```
Generalization  = {Comment | Note | TaggedValue | Stereotype | Constraint},
                  Parent, Child;
Parent          = string;
Child           = string;
```

Association and AssociationEnd

```
Association     = {Comment | Note | TaggedValue | Stereotype | Constraint},
                  {AssociationName}, AssociationEnd, AssociationEnd;
AssociationName = Name, [Direction];
Direction       = Source, Target;
Source          = Type, [RoleName];
Target          = Type, [RoleName];
RoleName        = string;
AssociationEnd  = {Comment | Note | TaggedValue | Stereotype | Constraint}, Type,
                  [Name], [Qualifier], [AggregationKind], [Multiplicity],
                  [isNavigable], [Visibility], [Changeability], [Ordering];
```

AggregationKind

```
AggregationKind = 'AggregationWhole' | 'AggregationPart' | 'CompositionWhole' |
                  'CompositionPart';
Ordering        = string;
Qualifier       = string;
```

Extension Package

```
TaggedValue     = {Tag, Value};
Tag             = string;
Value           = string;
```

Stereotype

```
Stereotype      = {Comment | Note | TaggedValue | Constraint}, Name, [BaseClass];
BaseClass       = string;
```

Constraint

```
Constraint      = {Comment | Note}, C_Body, C_Element;
C_Body          = string;
C_Element       = string;
```

B.4 UML Statechart Diagram syntax using EBNF

```
StatechartDiagram   = Name, {CompositeState | State | Transition}, [Note];
State               = Name, {EntryAction}, {DoAction}, {ExitAction}, [Note];
Transition          = {TransitionLabel | Note}, Source, Target;
TransitionLabel     = Event, {Guard}, {Action}, {SendClause};
Event               = Name, {Parameter | Note};
(* CompositeState and PseudoState are not included in the scope of the current
   research. They are mentioned for the purpose of completeness.        *)
CompositeState      = Name, IsConcurrent, IsRegion, { State | CompositeState |
                      PseudoState | Transition | Note};
PseudoState         = {Action | Transition | Note}, Kind;
Kind                = Initial | DeepHistory | ShallowHistory | Join | Fork | Branch |
                      Final;
IsConcurrent        = boolean
IsRegion            = boolean;
Parameter           = string;
Source              = string;
Target              = string;
Action              = string;
Guard               = string;
Note                = string;
SendClause          = string;
```

Appendix C

The BROOD Process Specification

The process is adapted from Rational Unified Process software process model. Only focus on three phases: analysis, design, and evolution. The following specification is based on OMG Software Process Engineering Metamodel.

Process: **Business Rule-based Object-Oriented Design (BROOD)**

Phase: **Analysis**

Activity: **Analyze Business Rule Statements**

ProcessRole: **Functional Analyst**
ActivityParameters {kind: input}
WorkProduct: **Use-Case Model** {state: **revised**}
WorkProduct: **Business Rule Statements** {state: **revised**}
ActivityParameters {kind: output}
WorkProduct: **Business Rule Specification** {state: **initial draft**}
Steps
Step: **Identify business rule type**
Step: **Rewrite business rules according to sentence templates**
Step: **Resolve rule conflicts and redundancy**

Activity: **Architectural Analysis**

ProcessRole: **Software Architect**
ActivityParameters {kind: input}
WorkProduct: **Use-Case Model** {state: **revised**}
WorkProduct: **Business Model** {state: **completed**}
WorkProduct: **Architecture Description** {state: **initial draft**}
WorkProduct: **Supplementary Requirements** {state: **revised**}
ActivityParameters {kind: output}
WorkProduct: **Analysis Class Diagram** {state: **outline**}
WorkProduct: **Analysis Package** {state: **outline**}
WorkProduct: **Architecture Description** {state: **revised draft**}
Steps
Step: **Identify analysis packages**
Step: **Identify analysis classes**
Step: **Describe analysis object interactions**

Activity: **Analyze a Class**

ProcessRole: **Component Engineer**
ActivityParameters {kind: input}
WorkProduct: **Analysis Class Diagram** {state: **outlined**}
ActivityParameters {kind: output}

WorkProduct: **Analysis Class Diagram** {state: **completed**}

Steps

Step: **Identify class responsibilities**

Step: **Identify class attributes**

Step: **Identify class relationships**

Activity: **Analyze a Package**

ProcessRole: **Component Engineer**

ActivityParameters {kind: input}

WorkProduct: **Analysis Package** {state: **outlined**}

WorkProduct: **Architecture Description** {state: **revised draft**}

ActivityParameters {kind: output}

WorkProduct: **Analysis Package** {state: **completed**}

Steps

Step: **Analyze the cohesiveness of each package**

Step: **Analyze the dependencies between packages**

Phase: **Design**

Activity: **Architectural Design**

ProcessRole: **Software Architect**

ActivityParameters {kind: input}

WorkProduct: **Use-Case Model** {state: **revised**}

WorkProduct: **Analysis Model** {state: **completed**}

WorkProduct: **Architecture Description** {state: **revised draft**}

WorkProduct: **Supplementary Requirements** {state: **revised**}

ActivityParameters {kind: output}

WorkProduct: **Design Class Diagram** {state: **outline**}

WorkProduct: **Design Package** {state: **outline**}

WorkProduct: **Architecture Description** {state: **revised**}

Steps

Step: **Identify subsystems and their interfaces**

Step: **Identify architectural significant classes**

Step: **Identify generic design mechanisms**

Activity: **Design a Class**

ProcessRole: **Component Engineer**

ActivityParameters {kind: input}

WorkProduct: **Design Class Diagram** {state: **outlined**}

ActivityParameters {kind: output}

WorkProduct: **Design Class Diagram** {state: **completed**}

WorkProduct: **Design Statechart Diagram** {state: **completed**}

Steps

Step: **Identify operations**

Step: **Identify attributes**

Step: **Identify relationships**

Step: **Describe method**

Step: **Describe state**

Step: **Link Statechart diagram element to class diagram**

Activity: **Design a Sub-System**

ProcessRole: **Component Engineer**
ActivityParameters {kind: input}
WorkProduct: **Sub-System** {state: **outlined**}
WorkProduct: **Architecture Description** {state: **revised**}
ActivityParameters {kind: output}
WorkProduct: **Sub-System** {state: **completed**}
Steps
Step: **Design sub-system dependencies**
Step: **Design sub-system interfaces**

Activity: **Develop Business Rule Specifications**

ProcessRole: **Component Engineer**
ActivityParameters {kind: input}
WorkProduct: **Business Rule Specifications** {state: **initial draft**}
ActivityParameters {kind: output}
WorkProduct: **Business Rule Specifications** {state: **revised draft**}
Steps
Step: **Define rule phrases**
Step: **Link rule phrase to design elements**
Step: **Form structured rule statements**
Step: **Populate rule attributes**
Step: **Organize rule set**

Activity: **Validate Business Rule Specifications**

ProcessRole: **Functional Analyst / Business User**
ActivityParameters {kind: input}
WorkProduct: **Business Rule Specifications** {state: **revised draft**}
ActivityParameters {kind: output}
WorkProduct: **Business Rule Specifications** {state: **completed**}
Steps
Step: **Ensure correctness of business rule specifications**
Step: **Ensure understandability of business rule specifications**

Phase: **Evolution**

Activity: **Examine Business Rule Change Request**

ProcessRole: **Business User / Functional Analyst**
ActivityParameters {kind: input}
WorkProduct: **Business Rule Change Request** {state: **initial**}
WorkProduct: **Business Rule Specifications** {state: **completed**}
ActivityParameters {kind: output}
WorkProduct: **Business Rule Change Request** {state: **revised**}
Steps
Step: **Determine the type of business rule change**
Step: **Revise business rule change request (for complex change)**

Activity: **Perform Business Rule Modification Change**

ProcessRole: **Business User / Functional Analyst**

ActivityParameters {kind: input}
WorkProduct: **Design Model** {state: **completed**}
ActivityParameters {kind: output}
WorkProduct: **Design Model** {state: **changed**}
Steps
Step: **Locate the relevant business rule specification**
Step: **Perform change on business rule specification**
Step: **Propagate change to software design**

Activity: **Analyze Business Rule Change Request**

ProcessRole: **Software Architect**
ActivityParameters {kind: input}
WorkProduct: **Business Rule Change Request** {state: **revised**}
WorkProduct: **Design Model** {state: **completed**}
ActivityParameters {kind: output}
WorkProduct: **Business Rule Change Plan** {}
Steps
Step: **Identify the effect of changes**
Step: **Produce the detailed change plan**

Activity: **Implement Business Rule Change**

ProcessRole: **Component Engineer**
ActivityParameters {kind: input}
WorkProduct: **Business Rule Change Plan** {}
ActivityParameters {kind: output}
WorkProduct: **Design Model** {state: **changed**}
Steps
Step: **Review the change plan**
Step: **Perform the changes**