ABSTRACT

A critical success factor for information systems is their ability to evolve as their environment changes. There is compelling evidence that the management of change in business policy can have a profound effect on an information system’s ability to evolve effectively and efficiently. For this to be successful, there is a need to represent business rules from the early requirements stage, expressed in user-understandable terms, to downstream system design components and maintain these throughout the lifecycle of the system. Any user-oriented changes could then be traced and if necessary propagated from requirements to design specifications and evaluated by both end-users and developers about their impact on the system. The BROOD approach, discussed in this article, aims to provide seamless traceability between requirements and system designs through the modelling of business rules and the successive transformations, using UML as the modelling framework.

Keywords: business rules; object-oriented design; software architecture; software evolution

INTRODUCTION

The ubiquitous nature of information systems and the increasing dependency of organizations, government and society on such systems highlight the importance of ensuring robustness in their operation. At the same time rapid changes in the environment of information systems places an increasing emphasis on the ability of these systems to evolve according to emerging requirements. A large proportion of a total systems’ lifecycle cost is devoted to introducing new requirements, and removing or changing existing system functionality (Grubb & Takang, 2003). Software evolution therefore is considered as a key challenge in the development and maintenance of information systems (Erlikh, 2000).

In recent years there has been an increasing interest of the IS community in business rules, which has resulted in dedicated rule-centric modeling frameworks and methodologies (Ross & Lam, 1999; Zaniolo et al., 1997), international initiatives for the investigation of business rules’ role in the context of knowledge management.
(Hay & Healy, 1997), conferences, workshops and tutorials (Mens, Wuyts, Bontridder, & Grijseels, 1998), and rule-centric rule management tools and application development support environments (e.g., Blaze Advisor Builder, BRS RuleTrack, Business Rule Studio, Haley Technologies, ILOG Rules, Platinum Aion, Usoft Developer and Visual Rule Studio). Whilst these efforts make significant contributions in their own right, a key challenge remains unanswered namely the linking of business rules specifications to software designs.

The aim of the BROOD (business rules-driven object oriented design) approach is to address the issue of software evolution from both requirements and design perspectives. This confluence should provide a seamless and traceable facility that arguably should bring about a more effective way of dealing with software evolution, by aligning changes of the information system to changes in its environment. BROOD adopts as its methodological paradigm that of object orientation with UML as its underlying graphical language. It augments UML by explicitly considering business rules as an integral part of an object-oriented development effort. To this end BROOD aims:

i. To explicitly model business rules in a manner understandable to end-user stakeholders.
ii. To map these to formal descriptions amenable to automation and analysis.
iii. To provide guidelines on the deployment of business rules in the development process.
iv. To provide guidelines on the evolution of requirements and related design specifications.

The article is organized as follows. Section 2 discusses the background to business rules modeling. Section 3 introduces the motivation for BROOD. Section 4 introduces the BROOD metamodel as the foundation for modeling business rules. Section 5 discusses the manner in which business rules are linked to design components via the concept of ‘rule phrase.’ The BROOD process is detailed in section 6. The BROOD approach is supported by an automated tool and this is briefly discussed in Section 7. The article concludes with an overview of BROOD, observations on its use on a large application and comparisons with traditional approaches.

The language details for business rules definition are given in appendix A. The BROOD approach is demonstrated through an industrial application which is described in appendix B. This application had originally been developed using a traditional approach. Therefore, it proved useful not only as a means of providing a practical grounding on BROOD but also on comparing and contrasting the use of BROOD with a traditional development effort.

BUSINESS RULES MODELLING

The motivation of BROOD is to provide a development environment whereby the business analysis and system design domains are supported by business rules modeling with the specific aim to facilitating more effective software evolution.

The term “business rule” has been used by different authors in different ways. For example, in (Rosca, Greenspan, Feblowitz, & Wild, 1997), business rules are:

> statements of goals, policies, or constraints on an enterprise’s way of doing business.

In (Herbst, 1996a), they are defined as:

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

Krammer considers them as “programmatic implementations of the policies and practices of a business organization” (Krammer, 1997) whilst Halle states that:

> depending on whom you ask, business rules may encompass some or all relationship verbs, mathematical calculations, inference rules,
step-by-step instructions, database constraints, business goals and policies, and business definitions. (Halle, 1994).

In general, business rules in the information systems field may be viewed in terms of two perspectives: (a) business rules as applied to conceptual modeling and (b) business rules as applied to evolvable software systems development.

**Business Rules in Conceptual Modeling**

1. Business rules as part of requirements gathering and systems analysis have not been ignored by structured analysis, information engineering or object-oriented analysis approaches (Moriarty, 1993) which, to varying degrees, subsume or represent business rules as part of notation schemes used to specify application requirements (Gottesdiener, 1997) Ross (1997) comments that traditional IS methodologies have addressed rules poorly, and only relatively late in the system development lifecycle. (Hay & Healy, 1997) mention that rules dealing with information structure may be represented by any of several flavors of entity—relationship or object class diagrams, and responses to events may be shown via essential data flow diagrams (McMenamin & Palmer, 1984) or as entity life history diagrams (Robinson & Berrisford, 1994).

   From a conceptual perspective there are approaches that consider business rules as an integral part of the modeling and analysis of systems’ requirements. An early effort in this direction was the RUBRIC project (Loucopoulos & Layzell, 1986; van Assche, Layzell, Loucopoulos, & Speltinex, 1988) parts of which were integrated into the information engineering (Martin, 1989) method.

   In BROCOM (Herbst, 1996b, 1997), the rule language is a type of structured English, and therefore it is highly expressive. Moreover, rules are organized according to a rich meta-model, and can be retrieved based on a number of different criteria. As far as methodological guidance is concerned, Herbst proposes the development of various models which are helpful during the analysis phase, but the process of creating and using them is not clearly defined. The transition from analysis to design and implementation has not been addressed by this approach.

   The DSS approach (Rosca, Greenspan, & Wild, 2002; Rosca et al., 1995) focuses on the analysis phase of IS development by supporting the rationale behind the establishment of rules. DSS adopts the ECA (event-condition-action) paradigm for structuring rule expressions and also links these expressions to the entities of an underlying enterprise model. The absence of a formal rule language confines the use of DSS on modeling tasks.

   The Business Rules Group (BRG), formerly known as the GUIDE Business Rule Project (Hay & Healy, 1997), investigated an appropriate formalization for the analysis and expression of business rules (Hay & Healy, 2000). This approach identifies terms and facts in natural language rule statements, and consequently, it offers a high level of expressiveness. The meta-model it provides for describing the relations between these terms and facts is very detailed. Therefore, rule models are (a) highly manageable and (b) formal and fully consistent with the information models of a specific organization.

   The IDEA method (Zaniolo et al., 1997) focuses on the maintenance of formality and consistency with underlying business models. The method offers guidance for every activity being involved in the development of a rule-centric information system. The IDEA method is directed towards the use of specific active and deductive databases, and of the corresponding rule languages. As a result of this, (a) IDEA rules are rather difficult to be expressed or even understood by business people; and (b) the choice of technologies to be employed for the development of an information system is rather limited.

   The BRS approach (Ross, 1997) is formal, in accordance with the underlying data models of an organization, offers sufficient methodologi-
cal guidance, and allows management of rule expressions based on a very detailed meta-model. It is also one of the few methods that adopts a graphical notation for expressing rules. 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 modeling (Ross, & Lam, 2003). BRS also provides the BRS RuleTrack™, an automated tool for recording and organizing business rules.

The object constraint language (OCL) of UML (Eriksson & Penker, 2000) is tightly bound with the widely accepted UML but lacks methodological guidance for the collection of rules. Rule structures are implied by the allocation of rules to classes, attributes, associations and operations.

A comparative evaluation of the treatment of business rules for conceptual modeling by three widely used approaches is shown in Table 1.

Table 1. Comparative evaluation of business rule in conceptual modeling

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

Lifecycle coverage: A-Analysis, D-Design, I-Implementation, M-Maintenance
Business Rules in Evolvable Software Evolution

The majority of approaches in this category aim to improve the understanding and evolution of a software system by logically and physically separating business rule components from other software components.

The adaptive object model (AOM), which is also known as the dynamic object model (Riehle, Tilman, & Johnson, 2000), is “a system that represents classes, attributes, and relationships as metadata” (Yoder, Balaguer, & Johnson, 2001). 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, a user only needs to change the metadata instead of changing the code to reflect domain changes.

The coordination contract method aims to separate coordination from computation aspects (or core components) of a software system (Andrade, Fiadeiro, Gouveia, & Koutsoukos, 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 & Fiadeiro, 2000). Volatile business products are implemented as contracts. A 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, Medvidovic, & Taylor, 1998), glue (Schneider, 1999), actor (Astley & Agha, 1998) or change absorbers (Evans & Dickman, 1999).

Business Rule Beans (BRBeans), formerly known as accessible business rules (Rouvellou, Degenaro, Rasmus et al., 1999; Rouvellou, Degenaro, Rasmus 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 minimize 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. A 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, Diaz, Fernandes et al., 2003).

A comparative evaluation of the treatment of business rules evolvable software systems development by the three approaches is shown in Table 2.

MOTIVATION FOR THE BROOD APPROACH

According to Lehman’s laws (Lehman & Belady, 1985), a software system that is used in a real-world environment inevitably must change or become progressively less useful in that environment. 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 due to its evolution.

The position put forward in this article is that developers need to identify the sources of changes for software evolution in the system’s environment and that some of the most volatile of these components tend to be business rules. In section 0 many contemporary approaches were reviewed all of which aim to externalize
business rules from software components.

At the conceptual modeling level, there are approaches that separate syntax and semantics for modeling business rules. This effort localizes the changes to business rule components, and also increases the understanding and maintainability of business rules specification. This category of approaches provides a great deal of help in dealing with the concepts related to business rules, but they provide relatively little description on the design and implementation aspect of business rules.

At the implementation level, approaches 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 the overall software structure. This group of approaches provides very good facilities for developing evolvable software components but is less helpful in representing business rules at the conceptual business level.

The BROOD approach addresses both business modeling and the linking of business model components to software architecture components. By focusing on the conceptual level, BROOD attempts to externalizing changes from software components. This user-oriented view enhances understandability and maintainability since it encourages the direct involvement of business stakeholders in the maintenance of their business rules.

---

**Table 2. Comparative evaluation of business rules in evolvable software systems**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Concepts</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Business Rule Definition</td>
<td>Implicit</td>
<td>Implicit</td>
<td>Explicit</td>
<td></td>
</tr>
<tr>
<td>Business Rule Taxonomy</td>
<td>primitive, composite, workflow</td>
<td>ECA</td>
<td>derivation, constraint, invariant, script, classifier</td>
<td></td>
</tr>
<tr>
<td>Business Rule Management Elements</td>
<td>Nil</td>
<td>Nil</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td><strong>Modelling Language</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Understandability</td>
<td>High</td>
<td>Medium</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>Expressiveness (business rules)</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>Formality</td>
<td>Low</td>
<td>High</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>Evolvability</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td><strong>Process</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lifecycle coverage</td>
<td>(Evolutionary)</td>
<td>D + I + T + M</td>
<td>A + D + I + T + M</td>
<td></td>
</tr>
<tr>
<td>Process description</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td>Coherence</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>Support for evolution</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td><strong>Pragmatics</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communicability</td>
<td>High</td>
<td>Medium</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>Usability</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>Resources availability</td>
<td>Medium</td>
<td>Medium</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td>Openness</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</td>
<td></td>
</tr>
</tbody>
</table>

Copyright © 2008, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
By introducing a linking component between the conceptual model of business rules and software design, BROOD attempts to increase business rule traceability. Traceability is highly desirable since one can keep ‘forward’ and ‘backward’ tracks of changes between business and software.

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

THE BROOD METAMODEL

The initial concept of the metamodel was introduced in (Wan Kadir & Loucopoulos, 2003; Wan Kadir & Loucopoulos, 2004). The metamodel is complemented by a language definition based on the context-free grammar EBNF, which is included in appendix A. 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.

At the outset, three main desirable characteristics were set for developing an appropriate business rule metamodel, which would be consistent with the aims of BROOD:

- It should have an exhaustive and mutually exclusive typology to capture different types of business rules.
- It should have the structured forms of expressions for linking the business rules to software design.
- It should include rule management elements to improve business rule traceability in a business domain.

These three characteristics form the basis for the development of the business rule metamodel, which is shown in Figure 1. This figure shows the business rules metamodel together with parts of the UML metamodel that deal with static (classes) and dynamic (actions and events) aspects. The key requirement of BROOD for tracing changes from business to software through the use of business rules is achieved by integrating these three metamodels.

Business Rules Typology

The metamodel classifies business rules into three main types, which are constraint, action assertion, and derivation.

Constraints

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 latter asserts the relationship types as well as the cardinality and roles of each entity participating in a particular relationship.

Examples of attribute constraints from the MediNet application expressed according to the BROOD syntax (see attribute constraint definition in appendix A) are:

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

Examples of relationship constraints for MediNET (see relationship constraint definition in appendix A) are:

- 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.
**Actions**

Action assertion concerns a behavioral 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*. An event can be either a simple or a complex event where the latter is constructed by one or more simple events using the logical connectives AND/OR. A condition may be a simple or complex condition. A 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.

An action is performed by a system in response to the occurrence of an event and the satisfaction of the relevant condition. The execution of action may change the state of the system. An action may be a simple action or a sequence of simple actions. Simple actions can be further categorized into three different types, trigger actions, object manipulation actions, and user actions. Trigger action invokes an operation, a process, a procedure, or another rule under certain circumstances. Object manipulation action 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.

Examples of action assertion for MediNET (see action assertion definition in appendix A) are:

When new invoice created then calculate invoice end date.
When patient consultation completed then removed 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.

Derivation
A derivation rule derives a new fact based on existing facts. It can be of one of two types, computation, which uses a mathematical calculation or algorithm to derive a new arithmetic value, or inference, which uses logical deduction or induction to derive a new fact. Typically, an inference rule may be used to represent permission such as user policy for data security. An example of a computation derivation rule such as “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” would be expressed as:

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

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.

The Rule Template
Rule templates are the formal sentence patterns by which business rules can 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 0.

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 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 rules definition in Appendix A. Business rules templates are shown in Table 3.

The Rule Management Elements
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.

Rule set is used to 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 paid’ can be grouped in a rule set which is related to ‘bill preparation’ process. By properly organizing rules, the complexity of managing a large set of rules can be reduced.
Table 3. Business rule templates

<table>
<thead>
<tr>
<th>Types</th>
<th>Templates</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Attribute Constraint</strong></td>
<td>&lt;entity&gt; must have</td>
</tr>
<tr>
<td></td>
<td>&lt;attributeTerm1&gt; must be</td>
</tr>
<tr>
<td></td>
<td>&lt;attributeTerm&gt; must be in &lt;list&gt;.</td>
</tr>
<tr>
<td><strong>Relationship Constraint</strong></td>
<td>[%&lt;cardinality&gt;]&lt;entity1&gt; is a/an &lt;role&gt; of [%&lt;cardinality&gt;]&lt;entity2&gt;.</td>
</tr>
<tr>
<td></td>
<td>[%&lt;cardinality&gt;]&lt;entity1&gt; is associated with [%&lt;cardinality&gt;]&lt;entity2&gt;.</td>
</tr>
<tr>
<td></td>
<td>&lt;entity1&gt; must have</td>
</tr>
<tr>
<td></td>
<td>&lt;entity1&gt; is a/an &lt;entity2&gt;.</td>
</tr>
<tr>
<td><strong>Action Assertion</strong></td>
<td>When &lt;event&gt; [if &lt;condition&gt;] then &lt;action&gt;.</td>
</tr>
<tr>
<td></td>
<td>The templates of &lt;event&gt; :</td>
</tr>
<tr>
<td></td>
<td>&lt;attributeTerm&gt; is updated</td>
</tr>
<tr>
<td></td>
<td>&lt;entity&gt; is deleted</td>
</tr>
<tr>
<td></td>
<td>&lt;operation&gt;</td>
</tr>
<tr>
<td></td>
<td>the current date/time is &lt;dateTime&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;number&gt; &lt;timeUnit&gt; time interval from &lt;dateTime&gt; is reached</td>
</tr>
<tr>
<td></td>
<td>&lt;number&gt; &lt;timeUnit&gt; after &lt;dateTime&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;userEvent&gt;</td>
</tr>
<tr>
<td></td>
<td>The templates of &lt;condition&gt; :</td>
</tr>
<tr>
<td></td>
<td>&lt;attributeTerm1&gt; &lt;relationalOperator&gt; &lt;value</td>
</tr>
<tr>
<td></td>
<td>&lt;attributeTerm&gt; [not] in &lt;list&gt;</td>
</tr>
<tr>
<td></td>
<td>The templates of &lt;action&gt; :</td>
</tr>
<tr>
<td></td>
<td>trigger &lt;process&gt;</td>
</tr>
<tr>
<td></td>
<td>set &lt;attributeTerm&gt; to &lt;value&gt;</td>
</tr>
<tr>
<td></td>
<td>create</td>
</tr>
<tr>
<td></td>
<td>&lt;userAction&gt;</td>
</tr>
<tr>
<td><strong>Computation</strong></td>
<td>&lt;attributeTerm&gt; is computed as &lt;algorithm&gt;</td>
</tr>
<tr>
<td><strong>Derivation</strong></td>
<td>if &lt;condition&gt; then &lt;fact&gt;.</td>
</tr>
<tr>
<td></td>
<td>The templates of &lt;fact&gt; :</td>
</tr>
<tr>
<td></td>
<td>&lt;entity&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;entity&gt; may [not] &lt;action&gt;</td>
</tr>
</tbody>
</table>

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 own owner. It is important to define the owner information in a 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. An owner may be an organizational unit, 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 RULE PHRASE
A rule phrase in BROOD links a user-oriented business rule definition to a software design component. There are alternative ways in which this may be achieved. For example, using a rule object or rule engine, or making use of OCL. The use of rule object or rule engine increases the semantic distance between analysis and design and imposes implementation considerations. 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 claims that no mathematical background is required in using OCL.

Rule phrases are considered as the building blocks for 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.

The mappings between rule phrase types and UML model elements are summarized in Table 4. 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 is linked to operation specification.

Rule phrases for event, condition, and action, which are the building blocks for action assertion rules, are naturally linked to statechart diagram. Event, condition, and action 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. 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.

THE BROOD PROCESS

The BROOD process is described using the process model based on the syntax and semantics of the OMG software process engineering metamodel (SPEM). SPEM was developed by the Object Management Group to provide a metamodel and notations for specifying software processes and their components (OMG, 2002). SPEM extends the unified modeling language (UML) (OMG, 2001) metamodel with

| Table 4. Association 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) |
process specific stereotypes. A part of SPEM that shows most of the important components of a process structure is shown in Figure 2.

In SPEM, a work product is an artifact 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 that 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 performs or assists with specific activities.

The core activities of the BROOD process are situated in the 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 the design phase to produce a more traceable and consequently evolvable software system.

The flow of activities in each BROOD phase is shown in Figure 3.

The Analysis Phase
As shown in Figure 4, the analysis phase starts with an architectural analysis activity that considers the work products from requirements phase such as use-case model, business model, initial architecture descriptions, and supplementary requirements. A software architect performs architectural analysis by identifying the analysis packages based on the functional requirements and knowledge of the application domain. Each package realizes a set of closely related use cases and business processes to minimize the coupling between packages, which in turn localizes business changes. This activity identifies analysis classes and outlines their name, responsibilities, attributes, and relationships. In order to extract more information about the behavior of the classes, collaboration or interaction diagrams 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.

Considering the MediNet application, architectural analysis resulted in three packages business processes i.e. registration, billing, and invoicing. The registration package groups all

---

Figure 2. An excerpt from OMG software process engineering metamodel (OMG, 2002)
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.

The outline of analysis class diagrams and packages are further refined by class analysis and package analysis activities, respectively. A 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 can be identified to understand the behavior of objects from certain classes. These steps are repeated until a complete analysis class diagram, statechart diagram and package are achieved.

The activity of business rule modeling considers the informal statements captured during initial requirements and identifies the types for each business rule statement based on the BROOD typology. Business rule statements are transformed into more structured business
rule specifications according to the templates’ definition.

Table 5 shows a set of structured rules for the MediNet application. This template provides the means of managing rules as they get discovered and analyzed and acts as a ‘repository’ of rules for their entire lifecycle.

The Design Phase

The design phase involves 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 an application domain. The application-general subsystems are related to implementation technology decisions such as the introduction of user interface and database connectivity layers. The MediNet subsystems definition is shown in Figure 5.

The class design activity elaborates further the static and dynamic information of classes that were defined during the analysis phase. Additional information on the operations, attributes, and relationships can be added to each class. The specification of operations and attributes is made using the syntax of the chosen programming language. If necessary, the methods that specify the algorithm for the implementation of operations are specified.

The class design activity for the MediNet application resulted in detailed specification of for the three packages of registration, billing and invoicing. The class association diagram of Figure 6 shows the class details for invoicing. In order to reduce diagrammatic complexity all parameters and return values are hidden in the class operations.

The calculation of invoice amount is different for different types of invoice. 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. 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. In terms of payment, MediNET allows balance forward invoicing method in addition to open item method.

Within the design process classes are further elaborated in terms of the events and conditions that trigger their transition from one state to another. These are shown as statechart diagrams. For example, a statechart diagram for the HCServiceInvoice object is shown in Figure 7.

Within the BROOD design phase, rule phrase specifications are developed. 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. A 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.

For the MediNet application, the rules shown in Table 5 are specified according to rule phrases syntax as shown in Table 6.

The first rule in Table 6 shows the rule phrase derived from the attribute constraint rule, informally defined in the analysis phase as “A patient must have a unique registration
Table 5. Business rule statements for the MediNET application

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

number.” The rule phrases ‘a patient’ and ‘registration number’ are 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 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.

The second rule in Table 6 shows a relationship constraint. 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 as in the attribute constraint rule, that is their purpose is to propagate business changes to design elements. The former specifies the multiplicity of an association end whilst the latter specifies the role of an association end.

In the action assertion rule “When a payment is not received within 30 days from the invoice date, then the first reminder will be sent,” the rule phrases that represent the event, condi-
tion, 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 computation and inference rules are linked to the operation specification—the computation rule is linked to the specification of calculateAmount() operation in HCPMe-niNETUsageInvoice 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 Evolution Phase
In general, business rule changes may be classified into simple and complex changes. A 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. A complex change involves the addition or deletion of rule phrases or design elements.

Ordinarily, simple business rules changes could be performed by business users. The examples of five change scenarios that require simple business changes in MediNET system are shown in Table 7.

The implementation of a complex business rule change requires more effort 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. Three examples of complex rules changes are shown in Table 8.

The first scenario initiates the modification of two existing business rule statements, the calculation of bill and the calculation of invoice amount. These business rule changes consequently lead to a minor change in software design, that is the introduction of hasMaxBill attribute in the Paymaster class.

In the second scenario, the paymaster decided to introduce different healthcare benefit coverage to different levels of their payees. For example, executive staff is entitled to any medical treatment and medical procedures...
whilst production staff is 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 third scenario requires the intervention of a software developer. This scenario 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 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 engineers to modify the action component of the transition from the initial state to ‘Active’ state in the STD for HCServiceInvoice object.

THE BROOD SUPPORT TOOL

The BROOD process introduces several additional activities to the traditional object-oriented software design process. These additional activities include the documentation of business rules and their linking to software design components. To assist a developer with these BROOD-specific activities, a tool has been developed that supports the activities of business rule specification and management, software design editing, and business rule change propagation.

The BROOD tool was developed on top of the generic modeling environment (GME) (Ledeczi et al., 2001; VU, 2003), which is a configurable modeling environment.

The metamodel and templates, which are discussed in section 0, were used to implement the BROOD tool environment.

Table 6. Rule phrases and linked software design elements for the MediNet application

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

Table 6. Rule phrases and linked software design elements for the MediNet application
Table 7. Simple change scenarios for the MediNet application

<table>
<thead>
<tr>
<th>Change Scenarios</th>
<th>Changed Business Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. HCP allows patients to make ‘more than one payment for their bills’ instead of the previously set ‘single payment for each bill’.</td>
<td>One patient bill is associated with zero or more payments.</td>
</tr>
</tbody>
</table>
| 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.          |

GME was used to visually edit the software design models, business rule specification, and rule phrase entries. Three main modules (known as interpreters in GME) 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, since a manual undertaking of such propagation would be impractical for most applications.

The BROOD tool has been designed to be used by both software developers and business users. A user-friendly interface is provided to ease the management and traceability of business rules by non-IT users. An overview of the BROOD support tool is shown in Figure 8.

The metamodel, the graphical model editor, the rule phrase management, the business rules composition and the business rules modification functions are part of the core component and user application layer in the BROOD tool architecture. The rule phrase entries, business rule specification, and software design models are stored in the storage layer.

The BROOD tool maintains the consistencies between business rule and the linked software design each time a business rule is created or modified. It provides full automated support in performing simple changes and partial support for complex changes since these require creative skills of software engineers in making a design decision.

There are four main types of model that can be managed using the BROOD tool: rule phrase entries, business rule, class diagram, and statechart diagram. Users may select the type of model to be created from a set of choices. An example of the BROOD model editor is shown in Figure 9. The model editor provides a convenient way to create a model and also to connect it or parts of it to other models.

While graphical model editing is convenient for visual models such as those of class and statechart diagrams, it is less helpful for business rules specification.
The graphical model editor can be used for some simple business rules definition such as cardinality, relational operator, list, and optionality but for more complex rules the BROOD tool offers a dedicated rule editor, the add business rule (ABR) module. This module performs two main tasks: business rule composition and software design updating. In business rule composition mode, rule phrases are used to construct a business rule statement. In software design updating mode the module updates the software design model that corresponds to the composed rule.

The BROOD tool also helps with the implementation of business rule changes. The modify business rule (MBR) module was developed to assist tool users in performing this task, an example of which is shown in Figure 9.

A full description of the tool is beyond the scope of this article. It should be stressed however, that the tool plays an important part in the effective application of the BROOD approach by simplifying a sometimes tedious, error-prone, and time-consuming task of linking and propagating business rule changes to software design components.

DISCUSSION

The main aim of BROOD has been to facilitate the process of software evolution through: (a) externalization of business rules and their explicit modeling and (b) the linking of each

<table>
<thead>
<tr>
<th>Change Scenarios</th>
<th>Changed Business Rules</th>
</tr>
</thead>
</table>
| 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%) |

Table 8. Complex change scenarios for the MediNet application
Figure 8. Overview of the BROOD tool

Figure 9. Example of the BROOD model editor
modeled business rule with a corresponding software component. This approach provides full traceability between end-user concepts and software designs. By combining BROOD to design traceability in source code (Alves-Foss, Conte de Leon, & Oman, 2002), it is possible to achieve effective traceability in a software system.

The BROOD metamodel offers a complete foundation and infrastructure for the development of a software system that is resilient to business rule changes. 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. In contrast to BRG, BROCOM, and BRS approaches, BROOD attempts to remove 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 the concept of ruleset to organize the groups and hierarchy of the closely related business rules.

In terms of its modeling language, BROOD offers a high level of expressiveness. The keywords in the language definition and a sufficient number of sentence templates should provide adequate representation constructs. In general, achieving total expressiveness of the modeling language business rules is relatively hard to achieve due to the large number of ways of expressing business rules in a natural language. The usability of BROOD in this context will be proved in due course once the approach has been applied on different domains and applications. BROOD was 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 conflict and redundancy in representing the meaning of business rules.

In practical terms, BROOD can be applied using the 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 use of business rule templates and UML 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 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 as linking business rules to software design and handling different types of changes.

The utility of BROOD was demonstrated in this paper through the use of the MediNet industrial application. This application had originally been developed using a standard object-oriented approach. It was therefore possible (and indeed desirable) to use the case study not only as a way of demonstrating BROOD but also for comparing and contrasting BROOD to a traditional development approach.

By considering UML for software design, BROOD maintains the well-known object-oriented design quality attributes such as modularity, high cohesion, low coupling, efficiency, and portability. BROOD however provides additional quality attributes such as requirements traceability, software evolvability, and approach usability.

The traditional approach deployed for MediNet did not provide explicit traceability of business policy defined during the requirements specification phase. Instead, it provides a so-called ‘seamless transition’ from the use case models that document the user requirements to the analysis and design models. This resulted in business rules being embedded in both requirements specification and software design models. In contrast, with BROOD there was a natural transformation of the MediNET requirements into the structured business rules specification and in turn this specification was directly related to software design components.

Concerning software evolution, the implementation of changes using the traditional approach required the use of expertise with
specific knowledge of the MediNET software design. Since software engineers do not normally initiate business changes, they had to repeat all phases in MediNET development lifecycle especially requirements and analysis phases. Locating the related software design components was hard since there was no explicit link between the MediNET design models and its user requirements.

In relation to approach usability, the traditional approach was easier to apply during development since it did 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 these 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 could perform the simple business rule changes as demonstrated in the MediNET application. Rapid change implementation is important especially in business critical applications with intolerable downtime. The detailed process description facilitated the implementation of complex changes in MediNET.

In summary, BROOD contributes to three critical areas namely business rules specifica-

---

**Figure 10. Example of the BROOD business rules modifier**

![Image of BROOD business rules modifier](image-url)
tion, object-oriented design, and software evolution process. The proposed business rule specification 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 structures of rule templates have been defined so as to make them suitable for linking to software designs in support of future software evolution. A specification is aligned to changing user requirements via the linking of business rules to software designs through a detailed transformation of business rule into the specification of related software design components. Thus, the externalization of frequently changing aspects of a system into detailed business rules and the maintenance of associations between these and corresponding software components should provide a strong framework for effective software evolution.

ACKNOWLEDGMENT
The authors would like to thank the human resource department of Universiti Teknologi Malaysia (UTM) for partially sponsoring this research, and Penawar Medical Group, Malaysia for the permission to use its MediNET healthcare information system requirements specification as the case study. The authors wish to also express their gratitude to the three anonymous reviewers and to the editor of the special issue, Professor Dinesh Batra, whose insightful and detailed comments have contributed to the production of a much improved version of this article.

REFERENCES


IBM (Cartographer). (2003). *Ibm websphere application server enterprise*


OMG (Cartographer). (2001). *Omg unified modeling language specification*

OMG (Cartographer). (2002). *Software process engineering metamodel specification*


APPENDIX A. SPECIFICATION OF BROOD

Part I: The BROOD Metamodel

Business Rule Organisation and Typology

```plaintext
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];

Constraint

coststraint = 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

```plaintext
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 ('must be updated' | )
| entity ('is deleted' | 'is created')
```

Copyright © 2008, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
(operation | business_rule), 'is triggered';

\[
\text{time_event} = \text{date_time} \mid \text{n, time_unit, 'time interval from', date_time, 'is reached'} \mid \text{number, time_unit, 'after', date_time};
\]

\[
\text{user_event} = \text{string};
\]

\[
\text{complex_event} = \text{simple_event}, \{('Or' | 'And'), \text{simple_event} \{('Or' | 'And'), \text{simple_event}\}\}
\]

\[
\text{Condition} = \text{simple_condition} | \text{complex_condition};
\]

\[
\text{simple_condition} = \{'\text{Not'}\}, \text{attribute_term}, \text{relational_op}, \{\text{value} | \text{attribute_term}\} | \text{attribute_term}, ('\text{in'} | '\text{not in'}), \text{list};
\]

\[
\text{complex_condition} = \text{simple_condition}, ('\text{Or'} | '\text{And'}), \text{simple_condition}, \{('\text{Or'} | '\text{And'}), \text{simple_condition}\};
\]

\[
\text{Action} = \text{simple_action} | \text{action_sequence};
\]

\[
\text{simple_action} = \text{trigger_action} | \text{object_manipulation_action} | \text{user_action}, \{\text{Class, Operation}\};
\]

\[
\text{trigger_action} = '\text{trigger'}', \{\text{process} | \text{operation} | \text{business_rule}\};
\]

\[
\text{object_manipulation_action} = '\text{set}', \text{att_term}, 'to', \text{value} | ('\text{create'} | '\text{delete'}), \text{object};
\]

\[
\text{action_sequence} = \text{simple_action}, \{\text{simple_action}\};
\]

\[
\text{Derivation} = \text{computation} | \text{inference};
\]

\[
\text{Computation} = \text{attribute_term}, 'is computed as', \text{algorithm}, \{\text{Class, Operation}\};
\]

\[
\text{algorithm} = \text{string};
\]

\[
(*) \text{i.e. any specification language for specifying the algorithm e.g., OCL, pseudo-code, etc. *)
\]

\[
\text{Inference} = '\text{If}', \text{condition}, 'then', \text{fact};
\]

\[
\text{fact} = (\text{attribute_term} | \text{entity}), \text{relational_op}, ['a'], \text{value} | \text{entity}, ('\text{may'} | '\text{may not'}), \text{action}, \{\text{Class, Operation}\};
\]

\[
\text{Rule Phrases / Linking Elements / Low-Level Definitions}
\]

\[
(*) \text{Some low level non-terminal symbol such as <real>, <integer> and <string> are not defined.} \quad \quad \text{**** *)}
\]

\[
\text{entity} = \text{phrase, Class};
\]

\[
\text{attribute} = \text{phrase, Class, Attribute};
\]

\[
\text{operation} = \text{phrase, \{Class, Operation\}};
\]

\[
\text{cardinality} = \text{phrase, maxCard, minCard};
\]

\[
\text{eventPhrase} = \text{phrase, event, \{Class, Operation\}};
\]

\[
\text{actionPhrase} = \text{phrase, action, \{Class, Operation\}};
\]

\[
\text{role} = \text{string};
\]

\[
\text{list} = \text{string,\{string\}};
\]

\[
\text{phrase} = \text{string};
\]

\[
\text{value} = \text{string} | \text{integer} | \text{real} | \text{date} | \text{time};
\]

\[
\text{number} = \text{real} | \text{integer};
\]

\[
\text{time_unit} = \text{'second' | 'minute' | 'hour' | 'day' | 'month' | 'year'};
\]

\[
\text{relational_op} = \text{'equal' | 'not equal' | 'less than' | 'less than or equal' | 'greater than' | 'greater than or equal'};
\]

\[
\text{name} = \text{string};
\]

\[
\text{priority} = \{'\text{high'} | '\text{medium'} | '\text{low'}\};
\]

\[
\text{is_mandatory} = \text{boolean};
\]

\[
\text{is_propagatable} = \text{boolean};
\]

\[
\text{boolean} = \{'\text{true'} | '\text{false'}\};
\]

Copyright © 2008, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
Part II: The BROOD Process

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

**APENDIX B. THE MEDINET APPLICATION**

MediNET is a suite of internet applications that addresses the administrative and back-end processing requirements of the healthcare business community. It acts as a secondary layer to
the existing administrative and information systems. MediNET 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.

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 example 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 application. It rents MediNET to HCPs and paymasters as and when the applications are needed and charges them based on the number of performed transactions.

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 patient billing and paymasters invoicing. 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 rule to block non-paying paymasters, and the conditions to issue reminder for past due invoices. These frequent changes indicate the need for an approach to simplify the implementation of changes in MediNET software system.
Universiti Teknologi Malaysia, and currently, he is the Head of the Department. He is also a member of pro-tem committee of Malaysian Software Engineering Interest Group (MySEIG). His research interests include architecture-based software evolution, requirements traceability, component reusability, business rules, object-oriented design and CASE tools.