Formal Support for an Informal Business Modelling Method

Yun-Heh Chen-Burger, David Robertson
Department of Artificial Intelligence, The University of Edinburgh
80 South Bridge, Edinburgh, UK
emails: jessicac@dai.ed.ac.uk, dr@dai.ed.ac.uk

Abstract Business modelling methods are popular but, since they operate primarily in the early stages of software lifecycles, most are informal. This paper describes how we have used a conventional formal notation (first order predicate logic) in combination with automated support tools to replicate the key components of an established, informal, business modelling method: IBM's Business System Development Method (BSDM). We describe the knowledge which we represent formally at each stage in the method and explain how the move from informal to formal representation allows us to provide guidance and consistency checking during the lifecycle of the model. It also allows us to extend the original method to a model execution phase which is not described in the original informal method. Although our formal modelling system has not been extensively tested, it relies for much of its interaction on diagrammatic notations which already appear in BSDM and are therefore already tested in practice. The role of the formal notation in this case is not to provide a formal semantics for BSDM but to provide a framework for sharing the information supplied at different modelling stages and which we can supplement with simple forms of automated analysis.

Key-words Business Modelling, BSDM, Formal Method, Process Modelling.

1 Introduction

Traditional software engineering techniques focus from the beginning on the system that needs to be built. There is little or no effort to analyse and understand how the business, in which the new system is to be deployed, is organised and how it operates. As a result, in the past a number of software systems were built that were eventually found largely unsuitable for the organisation. In response to this problem, business models are sometimes developed prior to the usual software engineering process.

IBM's *Business System Development Method (BSDM)* is an informal method for developing business models. These business models are used to capture and represent the given business environment (for which software has to be written) in graphical and textual format. They serve as a communication tool to converge the view of senior managers and as a basis to build software systems. A good business model also helps software engineers to come to a better understanding of what software is needed for the company.

In BSDM terms:

> *"A business model is a model of the business that defines the things that the business needs to manage, and the business rules that govern the behaviour of those things."* [3]

Initially a BSDM business model consists of an *entity model*, which is later extended to a *process model*, both of which are specified in a semi-formal way using diagrams and English text. An entity model describes the key components of a business' operation, e.g. the actors who are involved in the business operations within and outside of the business, abstract and physical information about the operation, key activities carried out to achieve certain business purposes, and the inter-relationship between these things. The constituent elements of a business are captured and denoted as *entities* with dependencies placed between each entity and those others on which it relies for its existence. A process model is a collection of business processes crucial to the business' operation. The context of a business process, the circumstances which trigger a process and the consequences of its actions are described.

On top of the entity and the process model familiar to conventional BSDM practitioners, we introduce another layer, the *procedural model* which extends the process model. Figure 1 illustrates our layered modelling approach and also shows which domain knowledge is being formalised for each layer and what kind of information can be inferred through the formalisation process. The domain knowledge for each layer can be divided into two types: model components and rules, which are shown in Figure 1 as ellipses and boxes, respectively. From the entity and process models, any violations of modelling rules or guidelines and possible corrections can be inferred. From the procedural model, the state transitions of the model caused by the execution of business processes can be inferred.

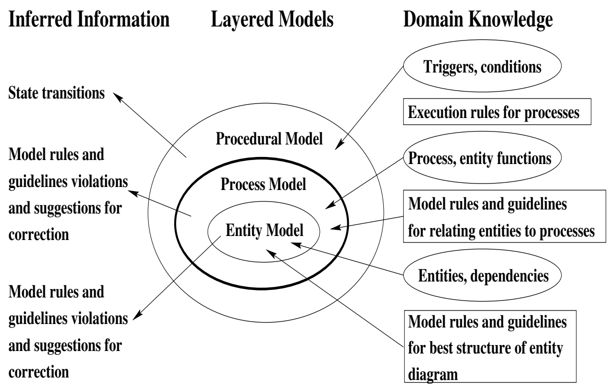

Figure 1. An Overview of The Executable Business Model

The rest of the paper is organised as follows. Section 2 summaries the motivation for this work and the overall objectives. An overview of how the formalisation of business models can be used in the context of this work is given in section 3. The basic representational approach is introduced in Section 4. The layers mentioned above are discussed in sections 5, 6 and 7, respectively. Each of these sections includes a discussion of the layer-specific domain knowledge representation and inferencing of new information. The paper concludes with section 8.

2 Motivation and Objectives

As is the case for any modelling activity, creating a model is only the first step in a larger cycle. Once a model has been designed, it needs to be verified and validated. *Verification* is the process of checking that no modelling rules have been violated. *Validation* is the process of confirming that the model is a true representation of the real world. The lack of a formal representation in methods like BSDM makes verification of a business model a tedious and error prone task. To validate a model, the modeller must as a minimum be able to work through the execution of typical scenarios for business processes and then compare these with the real world. For all but very simple business models, this is not achievable through a simple paper and pencil exercise and, hence, detailed validation has not been possible, in the past.

The objective of our work is to provide the necessary support to automate, as far as possible, the verification and validation of business models. By doing so, we provide the means to complete the modelling cycle, i.e. the modeller can go through several iterations of design, verification and validation until a satisfactory business model has been produced.

In formalising BSDM business modelling, the objective was not only to develop an appropriate formal representation of a model, but also to take advantage of the domain knowledge of BSDM about how to build such models and the existing set of rules about how to evaluate the quality of them. Furthermore, the formal representation of a BSDM business model must be able to capture not only the static aspects of the model, but also the dynamic ones.

Although the various states of the model can be captured, there is no explicit way in conventional BSDM of describing how a process is carried out and how entities are manipulated by a process. To be able to execute a business model, i.e. to simulate the execution of business processes, this knowledge of how to carry out a process is essential. An additional objective of this work was, therefore, the introduction of a procedural model on top of the entity and process model. Also, to be able to deal with the dynamic aspects of the models, the explicit representation of time needed to be introduced into the formalism.

Before describing in more detail how models and modelling rules are formally represented and then reasoned about, in order to enable the reader to put these subsequent sections in proper context, we first give a modelling overview below.

3 Modelling Overview

In this section, we briefly describe the overall context in which we use the formalisation of a business model. Figure 2 describes the modelling overview.

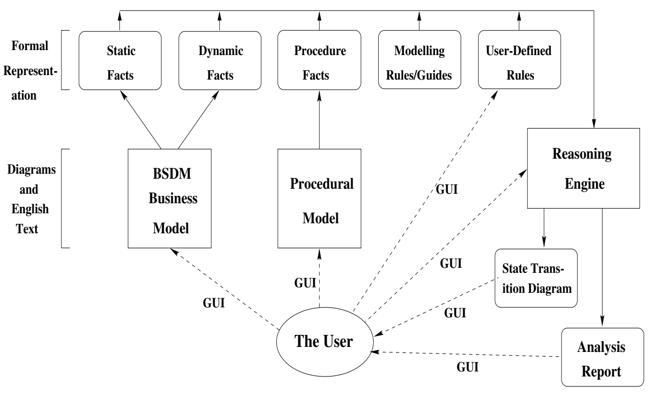

Figure 2. A Model Overview

At the beginning the user creates a business model which is described using BSDM notation. From the model, a formal representation can then be derived, one for the static and one for the dynamic aspects of the model, respectively. In addition, general as well as user-defined modelling rules are formalised. The general rules are considered to be fixed and, hence, require no user input. The user-defined rules are created with the help of input from the user. (At this stage, the procedural model is not yet needed.)

The formal representation of the model as well as the rules provide the input to a reasoning engine which analyses the business model. An analysis report is produced which describes any violation of modelling rules and makes recommendations for how the model can be improved. The user can then modify the business model accordingly and start the next iteration of this process. The cycle is repeated until no further errors and recommendations are produced, or when the user decides not to incorporate any more of the given recommendations. Once no more modelling rule violations are reported by the reasoning engine, verification of the model is complete.

Before the user can simulate and therefore validate the model, he/she must create the procedural model, which can then be formalised and fed into the reasoning engine. During simulation of the model the user interacts directly with the reasoning engine — telling it which process to execute from which starting state. The reasoning engine maintains a state transition diagram and information about the various states, which the user can compare with the effects of business processes in the real world. If the user decides that some changes in the business model are necessary, these changes must be formalised and then again input to the reasoning engine.

4 Basic Representational Approach

The formal representation used in this paper is based on an extended version of first order predicate logic. We use either a Prolog-like syntax or first order predicate logic expressions. The formal representation of modelling concepts and rules is organised in an inheritance class hierarchy which was inspired by the *PIF* core class hierarchy [5]. At the root of this hierarchy is a superclass called *rootclass*. The predicate

class(Super, Sub)

is used to denote the relationship between classes in this inheritance hierarchy: the class *Super* is a superclass of class *Sub*. Similar to the PIF class hierarchy, properties associated with a superclass are passed onto its subclasses through the inheritance hierarchy. Unlike the PIF hierarchy, however, this hierarchy not only captures the facts about a business model, it also includes classes that represent modelling rules.

A rule may be applied to a class anywhere in the hierarchy. It may be associated with one or more other classes. Subclasses inherit the association with rules from their parents. For example, if a particular rule has been associated with a particular class called *originate* process, then that rule applies to all subclasses of *originate*. Modelling these rules as their own set of classes allows for a easier identification and analysis of the modelling rules that apply to a particular domain.

5 A BSDM Entity Model

Figure 3 shows part of an example BSDM business model as it appears in our system. The notation used is that described in the conventional BSDM manuals so existing BSDM practitioners are conversant with the notation. This simplified model describes the selection and evaluation of modules at a university.

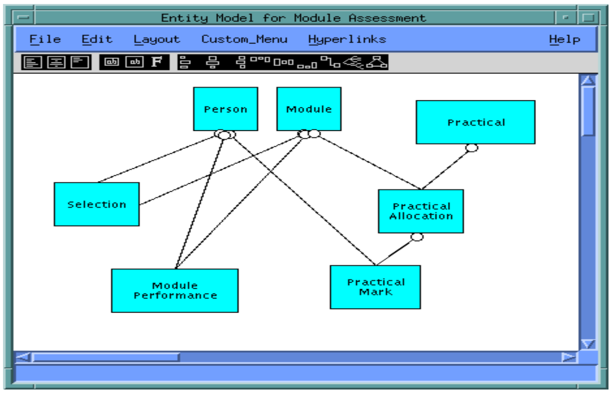

Figure 3. A BSDM's Entity Model

Boxes show the entities involved: 'Person', 'Module', 'Practical', 'Practical Allocation', 'Practical Mark', 'Selection' and 'Module Performance'. Lines with a circle at one end denote a *dependency* relationship between two entities: the entity with the circle ending is the "parent" entity, whereas the line ending entity is the "child" entity. For example, in this figure, 'Selection' has two parent entities: 'Person' and 'Module'. This denotes that it is impossible to have a 'Selection' (of a module) without knowing the particular person and the particular module. Hence, a 'dependency' places constraints on a child entity.

5.1 Representation of Entity Models

An entity represents a class of things in the world. Each 'Person', 'Module', 'Practical', 'Practical Allocation', 'Practical Mark', 'Selection' and 'Module Performance' is a specific kind of entity, and they are therefore represented as subclasses of *entity* in the inheritance class hierarchy. Using this class hierarchy, we can represent all entities shown in this business model. For example, entity 'Person' is represented as:

class(entity, 'Person').

To represent the dependency between entities, a predicate

parent_type(Entity, Set_of_Parents)

is used, where *Entity* denotes the child entity and *Set_of_Parents* is the corresponding set of parent entities. For example, 'Person' does not have a parent entity, therefore, its Set_of_Parents is empty, as shown below. 'Practical Allocation' has two parent entities, therefore these two entities are enclosed in the set, also shown below.

parent_type('Person', [ ])
parent_type('Practical Allocation', ['Module', 'Practical'])

5.2 Representation of Domain Knowledge

As mentioned earlier, the conventional BSDM manuals provide advice on how to build good business models. According to the degree of enforcement on a model, this can be classified into two categories: model rules and guidelines. Model rules must be followed if the model is to be sound. Model guidelines are recommendations for a model of good style. To distinguish these two different strengths of enforcement, two implication operators are deployed in the formalism: '$\Rightarrow$' is used to represent a model rule and is read as 'must be'; '$\triangleright$' is used for model guidelines and is read as 'should be'. A formula 'P $\Rightarrow$ ($\triangleright$) Q' reads 'if P is true then Q must (should) be true'.

For instance, BSDM recommends that the depth of an entity model should be no more than 4 layers, i.e. 4 steps through parent links. This is to prevent a model from being over-constrained by several layers of dependencies through levels of entities. If we define *property(Entity, level, N)* to mean that *Entity* is located at level *N* in the business model, then the "4-layer"- guideline can be described as:

$$property(Entity, level, N) \triangleright N < 5.$$

In addition to advice given explicitly in BSDM manuals, there are rules which are not mentioned in the method but are natural deductions from the method itself and, therefore, must be followed to create a sound model. These rules are also identified and captured as part of the formalisation. An example of such a rule is that any circular dependency relationship (a parent being dependent on its child) must not be allowed in a business model. To describe this rule, we define *ancestor(P, Q)* to mean that *P* is either a parent entity of *Q*, formally defined by the *parent_type* predicate, or that it is an ancestor entity of *Q* through the transitivity property of the *parent_type* predicate. The "non circular dependency"-rule is then represented by the expression below.

$$class(entity, X) \Rightarrow \neg ancester(X, X)$$

5.3 Inference

Each of the above logic expressions are *negated* and translated into semantically equivalent Prolog goals which use the above introduced representation of primitives (such as *class*, *parent_type* and *ancestor*) in the model. For example, logical expressions of the form below:

$property1(X) \Rightarrow property2(X)$ (must always be true), and
$property1(X) \triangleright property2(X)$ (should always be true)

are negated and implemented as Prolog goals for which we try to find an instance to satisfy (and can backtrack to generate all instances):

$$ruleGoal\_n :- property1(X), \\ not(property2(X)), advise(ruleGoal\_n, X).$$
$$guidelineGoal\_n :- property1(X), \\ not(property2(X)), advise(guidelineGoal\_n, X).$$

For example, the negated "circular dependency"-rule above will be "there exists at least one entity which has circular dependency relationship". Our implemented Prolog goal will be proven to be true, if it can find at least one entity which has circular dependency relationship.

In response to the user's request to verify a business model, the inference engine tries to prove each *ruleGoal_n* and *guidelineGoal_n* to be true using backward chaining. If any *ruleGoal* or *guidelineGoal* can be proven to be true, the corresponding modelling rule or guideline has been broken in the model, and advice is given to the user, via the Prolog goal advise(Violation_id, Source_data).

Having the knowledge about these rules and guidelines embedded in the system, the inference engine can provide information about the entities and processes which it has found are the cause of the violation. It is also able to give advice on how to correct the error. For instance, if the system has found a cyclic dependency in the model, a warning is given to the user containing information about this violation. It provides a list of entities which are involved in the cycle and suggests possible dependency links to be erased.

Although a detailed discussion of it is outwith the scope of this paper, it should be pointed out that the inference engine makes use of *case-based reasoning* techniques [4]. BSDM provides a catalogue of generally recommended topologies (standard models), which have been formalised and stored in a case library which the inference engine uses for analysis/correction of new models. The system is capable of detecting when a user's model is similar to an existing standard model in the library, report the differences and suggest actions which would bring the new model in line with a recommended standard model[1].

6 Process Model and Process-Entity Matrix

Conventional BSDM provides an informal method for extending an entity model to a process model. Figure 4 shows a screenshot of our system when describing an example process model which was built on top of the entity model shown in Figure 3. Two processes are shown in the figure: 'PRACTICAL MARK assignment' and 'MODULE PERFORMANCE assessment'. Each includes four entities in its scope.

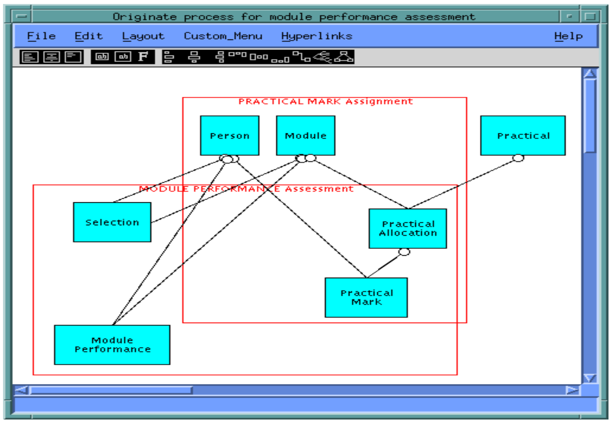

Figure 4. A BSDM's Process Model

A BSDM process includes at least one entity in its scope. The relationship between a process and its included entities are the specific roles that each entity plays in the process. This relationship is called an *entity function* in BSDM, and is summarised in tables like the one in Figure 5.

Process-Entity Matrix		
	PRACTICAL MARK Assign.	*MODULE PERFORM. Asse.*
Person	*refer normal*	
Module	*refer normal*	
Practical All.	*refer normal*	*refer normal*
Practical Mark	*originate focal*	*in-flight*
Selection		*refer normal*
Module Perfor.		*originate focal*

Figure 5. Process-Entity Matrix

The three types of entity functions used in our example are: *originate focal function*, *refer normal function* and *in-flight function*. The first one is used when a process creates occurrences (instances) of the corresponding entity. The second indicates that the process only refers to information in the entity. The third is used for entity/process pairs where the process normally only refers to the information in the entity, but can also create the entity if it doesn't already exist when the process starts execution, i.e. the third case is really a combination of the previous two. The intersections of the matrix are the relationships (entity functions) between the corresponding processes and entities.

The BSDM modeller specifies the role of each entity in each process. Based on the various roles the entities in a process play, the process is then classified as a particular type of process in the class hierarchy in the formalism, e.g. process *PRACTICAL MARK Assignment* is an *originate focal* process, and it is therefore represented as:

class(originate_focal, 'PRACTICAL MARK Assignment').

This process' primary purpose is to assign a mark to a person's practical, i.e. to create an occurrence of the *Practical Mark* entity (the *originate focal* entity). To create the occurrence, information about the particular *person* (entity), *module* (entity) and *practical allocation* (entity) are used. Hence, these entities are *refer normal* entities. In process *MODULE PERFORMANCE Assessment*, *Practical Mark* is an *in-flight* entity. This means that all practical marks must be known before the *module performance* can be calculated; otherwise, the corresponding practical marks must be calculated, as part of the process, before the *module performance* can be calculated.

Since entity functions are relationships between processes and entities, they are modelled as subclasses of *relationship* and *entity_function* in the class Hierarchy. To represent entity functions, a predicate *Entity_function(Process_name, Entity_name)* is used. Since three different kinds of entity functions are used in the above example: *refer normal*, *originate focal* and *in-flight*. The example formal representation of each of these entity functions are given below, respectively.

$$refer\_normal\_fun('PRACTICAL\ MARK\ Assignment', 'Person')$$
$$originate\_focal\_fun('PRACTICAL\ MARK\ Assignment', 'Practical\ Mark')$$
$$originate\_if\_fun('MODULE\ PERFORMANCE\ Assessment', 'Practical\ Mark')$$

6.1 Representing Domain Knowledge

BSDM provides check lists of rules and guidelines which are used to help the modeller to review if a developed model is appropriate. For instance, the following question should be checked: 'Are all entities originated (created) by at least one process?'. Since the modeller may decide that the creation of an entity lies outwith the scope of his/her model, the question forms a guideline, rather than a strict rule. The guideline states that each entity must play the role of either *originate focal*, *originate normal* or *in-flight* entity-function in at least one process:

$$class(entity, Entity\_name) \triangleright$$
$$\exists Process\_name. \left(\begin{array}{l} originate\_focal\_fun(Process\_name, Entity\_name) \vee \\ origiante\_normal\_fun(Process\_name, Entity\_name) \vee \\ originate\_if\_fun(Process\_name, Entity\_name). \end{array} \right)$$

It is, however, sensible to state that all entities in the model must be included in at least one process for some roles. For this, we use the following rule:

$$class(entity, Entity\_name) \Rightarrow$$
$$\exists Process\_name. \left(\begin{array}{l} originate\_focal\_fun(Process\_name, Entity\_name) \vee \\ originate\_normal\_fun(Process\_name, Entity\_name) \vee \\ originate\_if\_fun(Process\_name, Entity\_name) \vee \\ change\_focal\_fun(Process\_name, Entity\_name) \vee \\ change\_normal\_fun(Process\_name, Entity\_name) \vee \\ refer\_normal\_fun(Process\_name, Entity\_name) \vee \\ refer\_master\_fun(Process\_name, Entity\_name). \end{array} \right)$$

There are more model rules and guidelines which we will not mention here. They all follow the same basic notation as the two examples above.

6.2 Inference

Inference using the process model follows the same approach as described for the entity model. Rules/guidelines are *negated* and translated into Prolog goals which make use of the given class hierarchy and the basic formalisation of model primitives. Once one of the Prolog goals has been proven to be true, a violation of the corresponding process model rules or guidelines is found. The relevant information about the cause of the violation together with possible ways of correcting the error are provided to the user.

A feature of this design which was not mentioned in Section 5.3 (though it is also applied there) is the use of inheritance when applying certain rules/guidelines. Each rule/guideline only needs to be stated once, at the highest appropriate level of the class hierarchy. It is automatically applied to all of the corresponding subclasses by the inference engine.

So far only the static aspects of a business model have been discussed. The following section describes how we execute processes.

7 An Extension: Procedural Model

Conventional BSDM business models describe semi-formally what processes are, what they consist of and what can be done by them. The BSDM method, however, is not specific about how a process may be executed. In order to allow the execution of BSDM business processes, we introduce a *procedural model* which specifies the logical sequence of a process' execution. This is not the only possible form of execution consistent with the earlier stages of BSDM, but is characteristic of the sort of execution which fits well with the style of modelling. The notation used to describe this stage of modelling is not present in conventional BSDM manuals, so practitioners require training to use it.

The procedural model was inspired by process modelling methods such as [6], [7], [8] and workflow modelling method [2], but unlike these methods whose major emphasis is on specifying the execution sequence between different processes or tasks, the procedural model has been designed to specify the execution sequence within a process. Although a process can carry out several actions, most of the time it does not define an execution order on these actions. This is consistent with the declarative style of conventional BSDM which concerns itself with things which should be done, and does not give a deterministic order on execution. However, some commitments to the order of execution are required at this stage of design and these are reflected in the successive layers of Figure 6.

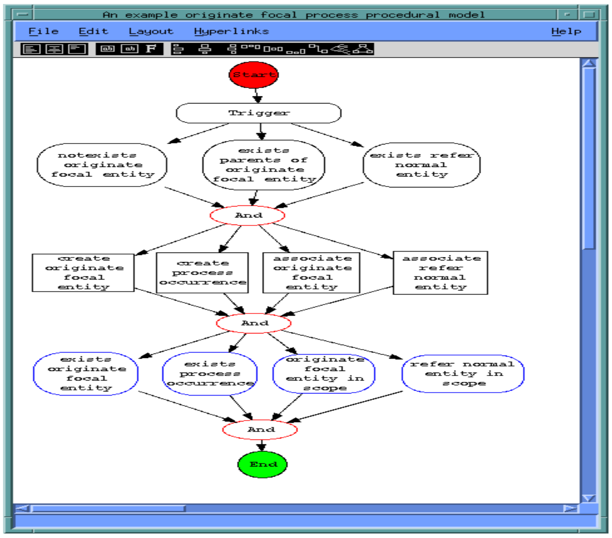

Figure 6. An Example Procedural Model for Originate Focal Process

The figure shows an example Procedural Model for an *originate focal* process. This model is derived from a generic structure of successive layers of tasks which can be used as a template for most originate focal processes. A process always begins with a 'start' circle after which comes a 'trigger' box which is followed by preconditions and an 'and' node which leads to actions. After actions are the postconditions and finally the 'end' circle.

A trigger represents a request to invoke a particular process. The presence of a trigger can be caused externally (i.e. by the user) or internally (i.e. by another process). If a trigger is present, all preconditions specified in the procedure must be satisfied before any actions can be carried out. All postconditions will be confirmed after all actions are finished. In the example, 'not exists originate focal entity' is a precondition. An example action box is 'create originate focal entity', and an example postcondition box is 'exists originate focal entity'.

Triggers deal with the dynamic state of a model and can be dynamically generated. A trigger needs to specify which type of process it invokes, when a process should start to act, and what actions are to be carried out. In addition, it also needs to distinguish itself from other triggers. The formal representation of a trigger and an example trigger ID 1, which invokes a process called 'Module Performance Assessment' at time 3, is shown below (the action to be carried out is to create an occurrence of the originate focal entity of the process):

trigger(Begin_time, Process_name, Trigger_id, Action_list).
trigger(3, 'MODULE PERFORMANCE Assessment', 1,
[originate_focal_entity]).

Preconditions for a process are requirements which make sure that process actions can be carried out successfully. Their existence is often linked to the actions to be carried out by a process. For instance, in Figure 6 the first precondition used is 'notexists originate focal entity', i.e. if a process is to generate its originate focal entity, then this entity must not already exist. This precondition is formally represented as:

precondition(originate_focal_entity,
[notexists,originate_focal_entity]).

The representation of a postcondition is similar to that of a precondition; the predicate name is replaced with 'postcondition'.

7.1 Representing Domain Knowledge

Model rules derived from the procedural model are primarily concerned with the operational aspect of a process execution. A process is normally carried out when its trigger is present and all of the precondition statements are satisfied. In some circumstances the process may only sometimes be applied, and therefore, a weak imply, $\triangleright$, is used.

The inference rule below states that *if* a trigger of a process is present, *then* there exists a time which equals to or greater than one time-unit before the calculated end time of a process's action, and all of the preconditions of this process occurrence (denoted by *process_precond* predicates) are found to be true at the time, *then* all actions of this process occurrence *should* be put into effect (denoted by negation of *no_effect* predicate) at time + 1 (which becomes the actual end time of the process). An occurrence of this process is created with its begin and end time specified as part of these actions; the end time of a process is denoted by a *occ_end_time* predicate.

$$
\begin{array}{l}
class(process, Process\_name) \wedge \\
trigger(Begin\_time, Process\_name, \\
\qquad Trigger\_Id, Process\_actions) \wedge \\
time\_cost(Process\_name, Time\_cost) \\
\Rightarrow \exists Time. \\
\left(\begin{array}{l}
Time >= (Begin\_time + Time\_cost - 1) \wedge \\
\neg \left(\begin{array}{l}
\exists Precondition.process\_precond(Process\_name, \\
\qquad Trigger\_Id, Precondition) \wedge \\
not\_valid(process\_precond(Process\_name, \\
\qquad Trigger\_Id, Precondition), Time)
\end{array} \right) \\
\triangleright \\
\neg \left(\begin{array}{l}
\exists Action \in Process\_actions \wedge \\
no\_effect(process\_action(Process\_name, \\
\qquad Trigger\_Id, Action), Time + 1)
\end{array} \right)
\end{array} \right)
\end{array}
$$

When a process occurrence finishes its actions at *End_time*, all of the corresponding postconditions of a process should be true. This is shown in the below expression.

$$
\begin{array}{l}
occ\_end\_time(Process\_name, Process\_Id, End\_time) \\
\triangleright \\
process\_postcond(Process\_name, \\
\qquad ProcessId, Postcondition) \wedge \\
valid(process\_postcond(Process\_name, \\
\qquad ProcessId, Postcondition), End\_time)
\end{array}
$$

7.2 Inference and State Transition Diagram

The inference engine generates the dynamic behaviour of the model, i.e. it simulates the execution of processes, according to the instructions given by the user. More specifically, the user selects a set of triggers for processes to be enacted. The scheduler of the inference engine then determines if any of the triggers in the previous set, or any of the new ones has a due time less than or equal to the current system time. If so, the corresponding processes are added to a *process agenda*. For those processes in the agenda which have all of their preconditions fulfilled, the process occurrences are created and actions executed. This leads to a new system state. Corresponding feedback is given to the user and the system time advanced. The next cycle begins with the new system state and the user can decide on the next set of triggers. The model simulation ends when the user decides to quit the system. The inference engine is able to backtrack to any previous system state (going backwards in time), thereby allowing the user to experiment with alternative paths of execution of the model. The execution history is maintained in a state transition diagram, the details of which are represented by the underlying formalism.

Notice that a transition from one state to another could be the result of more than one process. We, therefore, assume that processes which are carried out at the same time are not in conflict with each other. To prevent conflicting processes from executing at the same time, process constraints can be deployed, but this issue is not discussed in this paper.

Figure 7 shows an example state transition diagram which demonstrates the possible states generated by the system. It uses the two processes specified in the process-entity matrix in section 6.

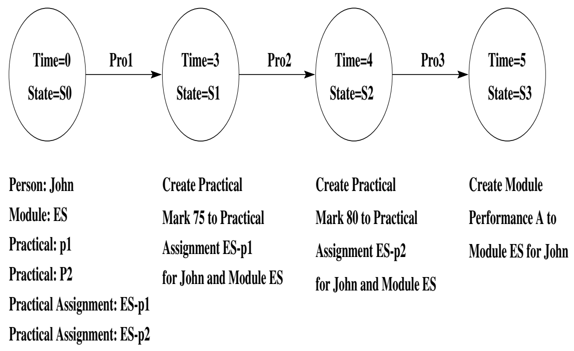

Figure 7. A State Transition Diagram for Originate Focal Process

The first circle denotes an initial state *S0* whose time is 0. Assuming that in the initial state we only have a few facts (occurrences): a person *John*, a module *ES*, a practical number one *p1*, and another practical number two *p2*, and their assignments to module *ES*: *ES-p1* and *ES-p2*. We also assume that there are three triggers given by the user initially: two invoke the 'PRACTICAL MARK Assignment' process which assigns the practical marks for practical assignment *ES-p1* and *ES-p2*, for *John* and module *ES*. The third trigger invokes the process 'MODULE PERFORMANCE Assessment' which assigns the module performance for *John* and *ES*. Notice that in this example we have assumed all triggers are given up front at the initial state for simplicity, but those triggers can be specified dynamically by the user, since the reasoning engine is interactive.

The directed link *pro1* denotes that process *pro1* was executed. It transferred the initial state *S0* to state *S1*. Assuming that process *pro1* ends at time 3, the newly created state *S1* is generated at time 3.

Process *pro1* has assigned a practical mark 75 to practical assignment *ES-p1* for *John* and module *ES*. Process *pro2* furthermore assigned practical mark 80 to practical assignment *ES-p2* for *John* and module *ES*. This process also transfers state *S1* to state *S2* denoted by a link labelled *pro2*. Since process *pro2* has ended at time 4, this has become the system time of the state. By the same reasoning, state *S3* is determined by a process *pro3* which assigns a module performance A to *ES* for *John*, and the system time is 5.

8 Conclusion

In recent years the software development cycle has been extended with the creation of business models prior to the requirements analysis phase in order to avoid the production of software systems which do not fit the needs of an organisation. IBM's BSDM is one of the development methods which has embraced this idea of business models.

In this paper we have shown how key components of an informally specified business model can be formalised using first order predicate logic. Having achieved the move from an informal to a formal representation of models and modelling rules we were able to provide guidance and consistency checking during the life cycle of a model — making use of techniques such as case-based reasoning. It also allowed us to complement the original method with a model execution phase. Although we found that not all aspects of business models can be formalised, e.g. English descriptions of models are often open to interpretation and context dependent, the support for automated analysis is most useful for verification and validation of business models.

References

[1] Y Chen-Burger, D Robertson, J Fraser, and C Lissoni. *A support Tool For Business Modeling in BSDM*. In British Computer Society, editor, *Proceeding of Applications and Innovations in Expert Systems*, December 1995.

[2] D Hollingsworth. *The Workflow Reference Model*. Workflow Management Coalition, 1994.

[3] IBM UK Ltd. *Business System Development Method, Introducing BSDM*, 2nd edition, May 1992.

[4] Janet Kolodner. *Case-Based Reasoning*. Morgan Kaufmann Publishers, Inc., 2929 Campus Drive, suite260, SanMateo, CA, USA, 1993.

[5] J Lee, M Gruninger, Y Jin, T Malone, A Tate, and G Yost. *The PIF Interchange Format and Framework*. The PIF Working Group, J Lee, Dept. of Decision Sciences, Univ. of Hawaii, 1996.

[6] T W Malone, K Crowston, J Lee, B Pentland, C Dellarocas, G Wyner, J Quimby, C Osborne, and A Bernstein. *Tools for inventing Organizations: Toward a handbook of organizational processes*. Centre for Coordination Science, MIT.

[7] Richard J Mayer, Michael K Painter, and Paula S deWitte. *IDEF Family of Methods for Concurrent Engineering and Business Re-engineering Applications*. Technical report, Knowledge Based Systems Inc., 1992.

[8] Craig Schlenoff, Amy Knutilla, and Steven Ray. *Proceedings of the Process Specification Language (PSL) Roundtable*. *NISTIR 6081, National Institute of Standards and Technology, Gaithersburg, MD*, 1997.