Knowledge management using business process modeling and workflow techniques

Citation for published version:
Kuo, H-L, Chen-Burger, Y-H & Robertson, D 2003, Knowledge management using business process modeling and workflow techniques. in 18th International Joint conference on Artificial Intelligence (IJCAI), Workshop on Knowledge Management and Organizational Memories.

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
18th International Joint conference on Artificial Intelligence (IJCAI), Workshop on Knowledge Management and Organizational Memories

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Knowledge Management using Business Process Modeling and Workflow Techniques

Hsiang-Ling Kuo\textsuperscript{1}, Yun-Heh Chen-Burger\textsuperscript{2}, Dave Robertson\textsuperscript{3}
\textsuperscript{1} School of Informatics, The University of Edinburgh, UK
email: s0125762@ed.ac.uk
\textsuperscript{2} AIAI, The University of Edinburgh, UK email: jessicac@inf.ed.ac.uk
\textsuperscript{3} CISA, The University of Edinburgh, UK email: dr@inf.ed.ac.uk

Abstract

Enterprise Modeling (EM) methods are recognised for their value in providing a more organised way to describe a complex, informal domain. A problem with EM is that it does not always provide direct input for software system development. There is a “gap” between EM and software systems. One way of bridging this gap is to provide a formalisation, here called that subsumes a wide variety of core modeling notations in a single using a business process language. It is possible to have different views of what is core to such a language but our attempt at such a view is articulated in the Fundamental Business Process Modeling Language (FBPML), which is a merger of IDEF3 and PSL. A workflow language, the FBPML Workflow Language (FWFL), is constructed and used to provide a declarative description of a workflow system. FWFL is tested in the “PC-configuration” domain. We also suggest using a validation and verification support framework to analyse and verify the business process model (BPM). Finally, some complexity results are presented for this type of modeling.

1 Introduction

Business are becoming larger and more diverse, thus operations are more complex than ever. Although information technology is widely applied in business operations, it still lacks a precise means of communication between business model and software system development. EM methods are well recognised for their value in providing an organised way of describing a complex, informal domain, and are often used as a tool for Knowledge Management (KM). There are many types of EM methods such as business modeling methods, process modeling methods, organisational modeling methods and related ontology design methods. However, EM does not always provide direct input for software system development which leaves a gap between enterprise modeling and software systems.

This paper tries to bridge the gap between Enterprise Modeling (EM) and Software Systems. To approach this, we have created a formalisation called “FWFL” for all models using FBPML which provides a declarative description of a workflow system. A workflow system may then be designed and implemented based on “FBPML” and “FWFL”. This would play an important communication role in the operation of an organisation.

In order to verify and analyse the BPM, a three-level framework is also introduced as a means of analysing BPMs and workflow systems. Finally, the complexity of BPMs and some comparisons with other related work are discussed.

2 Literature Review

Many business process modeling languages have been invented for different business process modeling needs. In this section, we will review two types of process modeling language and introduce a third one – Fundamental Business Process Modeling Language (FBPML).

2.1 IDEF3 Process Description Capture Method

IDEF3, a process description capture method, has two aspects: capturing process flow and object state. The primary goal of IDEF3 is to provide a well-structured method by which a domain expert can express knowledge about the operation of a particular system or organisation [Mayer et al., 1995]. It also captures the behavior of an existing or proposed system by structured description. Because of its well-structured approach, we can use IDEF3 as a knowledge acquisition device for describing what a system does or how an organisation works.

IDEF3 includes two forms of description: a process flow description and an object state transition network. A process flow description describes “how things work” in an organisation [Mayer et al., 1995]. It focuses on the processes and their temporal, causal, and logical relations. An object state transition network focuses on objects and their state change behaviors. This paper focuses on the process flow description because it is related to what we use in this paper.

An IDEF3 process flow description captures a description of processes and the relationships between them. It provides a graphical and structural representation that domain experts and analysts from different disciplines can use to communicate with each other. This includes knowledge about events and objects involved in the process, and the constraining relations which determine the behavior of each occurrence (process and object). It uses UOB (units of behavior), links, junc-
Joint Joint Joint Joint Split Split is used to represent a process. In this document, we will use this notation to elaborate our argument. In diagram, there are three types of nodes: Main Node, Junction and Annotation. Main Nodes include Activity, 

**Activity and Primitive Activity:** An Activity describes a type of process that may be decomposed into sub-processes. This is called “decomposition”. When all sub-processes are finished, the high level process is also finished. **Primitive Activity** is a leaf node activity that may not be further decomposed. However, there may be alternative ways of executing a process. When one alternative process is not executed and finished properly, another alternative process may collaborate with the current one to accomplish the task. We call these sub-alternative processes “specialisation”. In FBPML, three main components of an activity are trigger(s), precondition(s), and action(s). An activity also has an unique hierarchical position (HP) and a name to identify it.

**Time and Role:** The definition of Role in FBPML is useful, an enabler may play a Role that includes a set of activities and may have responsibilities for these activities. **Time Point** indicates a particular point in time during the process model. **Junctions** are widely used in many process modeling languages. In FBPML, there are four different types of junction: **Start, Finish, And** and **Or**. The **Start and Finish** junctions represent the beginning and end of a BPM. **Start** is the entry of a BPM. **Finish** is the point at which the model stops.

**And** or **Or** junctions indicate a one-to-many relationship, and a temporal constraint between the activities connecting to them [Chen-Burger et al., 2002]. Both of these junctions have two kinds of interpretation: **joint and split**. They represent the different topologies of a BPM. Figure 1 shows these four different types of topology.

**And Joint** or **Or Joint** indicates that there is more than one process preceding the “And” or “Or” junction but there is only one process following the junction. 1(a) and (b) show these kinds of junction. Both have three in-coming processes (A,B,C) and one out-going process (D). **And Joint** indicates the process execution sequence and the temporal constraint on the process. It means that all the processes (A,B,C) on the left-hand side must be finished before process D can be started. If one of the left-hand side processes cannot be finished, the entire flow cannot continue to the next stage. **Or Joint** means that when at least one of the left-hand side processes is finished then process D can be started; it does not need to wait for the other processes to be finished.

**And Split** or **Or Split** means that only one process will proceed to the “And” or “OR” junction, but more than one process will follow the junction. 1(c) and (d) illustrate this.
And_Split indicates that as long as the preceding process is finished then all the following processes become temporally qualified and may also be started. It also indicates that all the following processes must be triggered, but are allowed to be finished at some later time. Or_Split means at least one of following processes will be triggered and executed properly, after the preceding process is finished. It does not have any constraint about how many processes will be triggered. It may be one, two or more, depending on the trigger conditions of those processes. The execution sequence of triggered processes may be parallel or sequential.

Combination of “And” and “Or” Junctions: The “OR” and “And” junctions may also be combined to represent a more complicated BPM. There are four different kinds of combination. Suppose process A is connected with processes B, C, and D. Process E follows processes B, C and D in these different combination. The first type of combination is “And_Split” (figure 1 (c)) and “And_Joint” (figure 1 (a)). It means that when process A is finished then process B, C and D will be and must be started. After all the processes B, C and D are finished, then process E can start. It has the strictest restriction in a BPM. The combination of “Or_Split” (figure 1 (d)) and “Or_Joint” (figure 1 (b)) means that after process A is finished, at least one of the processes B, C and D will be started and executed. Process E will not be started unless one of the triggered processes is finished. It is a looser constraint than an “And_Split” and “And_Joint” junction.

The combination of “And_Split” (figure 1 (c)) and “Or_Joint” (figure 1 (b)) means that when process A is finished, all the processes B, C and D must be started and executed. If process B, C or D is finished, then process E can be started. It is different from an “And-And” junction in that the “Or_Split” (figure 1 (d)) and “And_Joint” (figure 1 (a)) junction indicates that at least one of the processes B, C and D will be started and executed after process A is finished. Process E will not be started unless “all of the triggered processes” are finished. The triggered processes may be a combination of some of them. Because of the “Or_Split” junction, it does not need to trigger all the preceding processes. It thus has more flexibility than the “And_Split” and “And_Joint” junction.

Annotations include Idea Note and Navigation Note. Idea Note records information which is related to the processes but not part of the process model. Navigation Note records the relationships between diagrams in a model [Chen-Burger et al., 2002]. Neither of them contribute to the formal semantics of the process model. Instead, they are used to help users to understand the processes more clearly from an intuitive point of view.

Nodes (Main Nodes and Junctions) are connected by links. Two types of links are provided: the precedence-link and the synchronisation-bar. A precedence-link indicates a temporal constraint between two processes. It means that activity B cannot start until activity A has finished. A synchronisation-bar also places a temporal constraint between two time points. This notation enables any time points to be made equivalent and therefore enables process operations to be synchronised.

### 2.4 Comparison between IDEF3, PSL and FBPLM

After briefly reviewing the IDEF3, PSL and FBPLM, we will focus on their similarities and differences. Table 1 summaries this brief comparison.

**Similarities:** IDEF3, PSL and FBPLM all focus on process expression techniques to provide a standard methodology when describing a process. These three business process representations allow domain experts to express knowledge about how a system works, and can provide a common language between different applications.

**Differences:** Although IDEF3, PSL and FBPLM are conceptually similar, in table 1 we lists the major differences between them. IDEF3 provides a graphical way to represent

<table>
<thead>
<tr>
<th>ITEM</th>
<th>IDEF3</th>
<th>PSL</th>
<th>FBPLM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>Process modeling language</td>
<td>Interchange language between different manufacturing applications</td>
<td>Business modeling language especially supports software and workflow system development</td>
</tr>
<tr>
<td>Notation</td>
<td>Rich Graphic Notation</td>
<td>Ontology and Formal Semantics</td>
<td>Simpler version Notation but semantics are presented</td>
</tr>
<tr>
<td>Basic Process Description</td>
<td>UOB (Unit of Behavior) Activity, Primitive-Activity</td>
<td>Activity, Primitive-Activity</td>
<td></td>
</tr>
<tr>
<td>Distinguish terms between process, activity and task</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Link between processes</td>
<td>Precedence Links with Different types of Constraints</td>
<td>Ordering Relation over Activities(ext)</td>
<td>Precedence-Link, Synchronisation-Bar</td>
</tr>
<tr>
<td>Junction</td>
<td>AND, OR, XOR</td>
<td>AND, OR, XOR Junction(core) Junction(ext)</td>
<td>✓ (not the same as PSL and IDEF3 but is an extension and refinement of both)</td>
</tr>
<tr>
<td>Time</td>
<td>Not in the notation but may be expressed informally</td>
<td>Duration(ext), Temporal Ordering Relation(ext)</td>
<td>Time point, duration, length</td>
</tr>
<tr>
<td>Role</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Annotation</td>
<td>Referent and Note</td>
<td>Idea Note and Navigation Note</td>
<td></td>
</tr>
<tr>
<td>Decomposition</td>
<td>A process may be decomposed into sub-processes</td>
<td>SubActivity(ext)</td>
<td>A process may be decomposed into sub-processes</td>
</tr>
<tr>
<td>Specialisation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Execution</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1: Comparison Between IDEF3, PSL and FBPLM

✓ – Yes
✗ – Not applied
core – PSL core concept
ext – PSL extension concept
processes, allowing the logic of processes to be more easily represented. Although IDEF3 has this advantage; it lacks an unambiguous semantic description for its notation. By contrast, PSL has a well-defined ontology and formal semantics but lacks graphical notations. Users cannot easily define processes in this language without proper training in logical languages. FBPM combines aspects of both IDEF3 and PSL to obtain the advantages of their different aspects. It provides a simpler and more pragmatic modeling language suitable for workflow system design.

3 Devising a logic-based Workflow Language for FBPM

FBPM is a visual and conceptual language. It captures and describes the business processes of an organisation. Besides describing a model, it also allows BPMs to be analysed, redesigned, and checked. Through FBPM, the tasks and operation of an organisation can be more easily understood. In order to provide the properties described above, FBPM has a declarative reading (understood independently of any particular computational procedure) as well as an operational reading when combined with a particular computational procedure. These are exactly the principles that FBPM embodies.

While FBPM gives precise execution logic for processes, it allows multiple versions of implementation of the workflow engine. This is because the FBPM specifies how processes should be enacted but does not specify the workflow engine that enacts those processes. In this section, we have constructed a workflow language “FWFL” (the FBPM WorkFlow Language) based on FBPM. In section 5, one version of the workflow engine will be introduced based on “FWFL”. The definitions of FWFL will be introduced in the following section.

3.1 FBPM WorkFlow Language (FWFL) Design

Process: The predicate defines a process. It has six parameters: ProcessId, ProcessName, Pstate, TrigCond, PreCond, and Action.

```plaintext
process(ProcessId, ProcessName, Pstate, TrigCond, PreCond, Action)
```

ProcessId defines the ID of a process. It must be unique to all processes in the BPM. ProcessName describes the name of a process. It does not have any impact on the execution, but is used only for human interpretation. It’s purpose is to help the user understand what process is. Pstate records the status of a process. There are two types of status in a process – “Triggered” and “Completed”. “Triggered” means that the process is already triggered and is temporally qualified to be executed. “Completed” means that the process has already been executed properly, i.e. it is already finished. TrigCond defines the trigger condition(s) for the process. It is composed of a triggered event or values about a state, e.g. attributes and their values of an entity. PreCond defines the pre-condition(s) for the process. It is composed of attributes and their values of an entity that a process manipulates. A simple example will be used to illustrate this later. A user can understand what the process looks like simply by reading through its description. Example 1 shows that process “a” needs an event to trigger the process. The content of this trigger event can be read simply through the description.

Actions For Processes: In FWFL, eight different types of action are provided. They are create_entity, get_newValue_from_usr, get_upValue_from_usr, add_attribute, delete_attribute, update_attribute, refer_attribute_of_entity and delete_entity.

The action create_entity creates a new entity and adds it into the entity database. To carry out this action – the attribute and its value for that entity are needed at execution time. The action get_newValue_from_usr gets a value from the user at run time, as a workflow system may have interactions with other systems as well as with the user. This action provides an interface through which a workflow system can obtain new data from a user. The action get_upValue_from_usr is similar to get_newValue_from_usr. The only difference is that it gets an updated value to update an existing attribute. The actions add_attribute, delete_attribute or update_attribute add, delete or update attributes in an existing entity. The action refer_attribute_of_entity refers to the value of an attribute provided by another entity. The action delete_entity deletes an existing entity. Sometimes the actions being carried out by one process may conflict with another process. The workflow system needs to deal with this situation and provide appropriate warning messages.

Example 1:

```
process(a, receiveCustomerReq, Pstate, [true],
    custom_req_for_pc_spec,
      created, attribute(Attribute))],
   [true],
   [create_entity(attribute(Attribute))]).
```

Instance: A process instance is an actual running process at execution time. A process (type) may have more than one instance depending on the number of events. The definition of an instance is inherited from process, such as TrigCond, PreCond and Action. The only difference between a process and an instance is that process’ variables ProcessId, ProcessName and Pstate are used instead of InstanceId, InstanceName and Istate, but the definitions of these parameters are the same. In addition, an instance has a parameter – BeginT/EndT that is different from process. It records the start and end time of that instance. As we know, an instance is the actual executed process, therefore, its variables are instantiated at run time. The workflow system needs to record this information and carry out some checking at execution time. A predicate – “instance” is therefore:

```plaintext
instance(InstanceId, InstanceName, Istate, TrigCond, PreCond, Action, BeginT/EndT)
```

Attribute: Attributes are used everywhere in FWFL such as “TrigCond”, “PreCond”, “Action”, “entity” and “event”. Attributes are defined in predicate:

```
attribute(EntityName = AttributeName/AttributeValue)
```

This means that the attribute, AttributeName belongs to a particular entity, EntityName. Its value will be assigned at execution time. For example, “customer-name/NameV” means the entity “customer” has an attribute “name”. The value has not been assigned yet.
FWFL also allows two other different kinds of attributes to be used. These two attribute definitions are slightly different from the previous one. The first type is the multiple-value attribute. For example, “ioBoard-capability(normal-graphics-long)” means that the entity “ioBoard” has an attribute “capability”, and “capability” has multiple values which are “normal”, “graphics” and “long”. These three values appear at the same time in the attribute “capability”. The second type is the alternative-value attribute. For example, “box-color[white,silver,black]” means that the entity “box” has an attribute “color”. The values of color are only one of “white”, “silver” or “black”.

**Attribute Domain and Attribute Value:** The domain of an attribute is defined by

\[
\text{all\_domain} \langle \text{Class, Attribute Name, Domain} \rangle
\]

The attribute, \text{AttributeName}, indicates that “AttributeName”. The field \text{Domain} defines the domain of values for the attribute. For instance, \text{all\_domain} \langle \text{processor, type, [t1, t2, t3, t4]} \rangle means that Class “processor” has an AttributeName “type”, and its domain is “t1”, “t2”, “t3” or “t4”.

**Attribute Value** is defined by

\[
\text{all\_value} \langle \text{Class, Attribute Name, Value} \rangle
\]

It means “AttributeName” belongs to class “Class”. The value of this AttributeName is “Value”. For example, \text{all\_value} \langle \text{customer, orderNo, value(1)} \rangle means that Class “customer” has an AttributeName “orderNo”, and its value is “1”.

**Entity:** An entity represents a class in the world. It defines the properties of an entity. The predicate is:

\[
\text{entity} \langle \text{Entity Name, Entity Id, Entity State, Entity Attribute} \rangle
\]

The \text{Entity Name} is the name of the entity. The \text{Entity Id} is the ID of the entity. The same entity type will have the same entity name but different IDs. Thus, the ID is unique. The \text{Entity State} records the state of the entity. There are two types of the state: “valid” and “invalid”. The entity is “valid” when it is created; it is “invalid” when it is deleted or after it reaches a particular time point. The \text{Entity Attribute} contains the attributes and their values for that entity. At execution time, every occurrence of this entity must be bound by these properties. An entity occurrence is defined as

\[
\text{entity\_occ} \langle \text{Entity Name, Entity Id, Entity State, Entity Attribute} \rangle
\]

which is inherited from the entity definition.

**Event:** An event defines the properties of a triggered event. It is represented as

\[
\text{event} \langle \text{Event Id, EventType, Event Flag, Event Attribute, Time} \rangle
\]

The \text{Event Id} indicates the ID of an event. The \text{EventType} is the type of an event. The \text{Event Flag} records the state of an event \text{“ ”}. The \text{Event Attribute} contains the attributes and their values for the event. \text{Time} records the trigger time for this event. The event predicate defines the properties that are needed in an event. In the same way, the event data is represented as an event occurrence. An event may have different occurrences. We use

\[
\text{event\_occ} \langle \text{Event Id, EventType, Event Flag, Event Attribute, Time} \rangle
\]

Two flags – “new” and “created” are defined in FWFL.

to express it.

**Junctions and Models:** The predicate

\[
\text{junc} \langle \text{Model Id, Junction Type, Pre Processors, Post Processors} \rangle
\]

is used to represent a junction. For example, if we want to represent a junction \text{OrSplit} which is connected with “b” and “c,d,e,f” for Model “m1”, we can represented is as \text{junc(m1, or\_split, [b], [c, d, e, f])}

**4 Validation and Verification Support Framework for Business Process Modeling**

In this research, a three-level framework is provided to analyse the BPM. The framework includes “model behavior”, “detailed model testing” and “instantiation of business scenario – case study.” This framework is intended to give users a more systematic structure when analysing a BPM. The framework first addresses the process flow control issues and then focuses on the detailed process analysis. The analysis of this framework may also be divided into two categories: syntactic and semantic [Sadiq and Orlowska, 1996]. The invalid use of a business process modeling language results in syntactic errors. Semantic errors occur due to incorrect modeling of business processes or getting into erroneous situations because of unanticipated combinations of task execution [Sadiq and Orlowska, 1996]. Based on these two types of error, we define our two types of critique as syntactic critique and semantic critique. We verify them and give advice in the three-level framework.

Figure 2 demonstrates our three-level framework and shows the relationships between level 1, 2 and 3. Level 1 considers the overall model behavior to find the appropriate topology for the BPM and carries out the syntactic critiques. Level 2 captures the topology features from level 1, carries out the semantic critiques and eliminates impossible execution sequences. Level 3 executes the BPM using business scenarios (entity data) in a particular domain and attempts to validate the model. Because the examined problem space for possible execution paths is reduced by each level as more information is presented by the model, we present our three-level approach in a three-layered oval graph. The detailed descriptions will be introduced in the following sections.

**4.1 Model Behavior Level**

Syntactic critiques are checked at the model behavior level. The types of syntactic critiques are shown in tables 2 and 3. This includes two parts: syntactic errors and syntactic warnings. “Model behavior” is also examined at this level. The model behavior indicates all the possible actions that the
model may carry out. The first level in this framework generates this kind of model behavior. At this level, all the possible triggered results and the resulting execution sequences are determined. The users may know all of possible behaviors of the BPM using this facility. This facility is useful in understanding the behavior of a similarly complicated model. Through this simple behavior enumeration, the user may have a rough idea about what kinds of flow may be executed. Particularly, the user may know the state enumeration and all possible execution sequences. Although the BPM that we deal with is written in FBPML, this approach is generic to other business process modeling languages.

### 4.2 Detailed Model Testing Level

The main purpose of level 2 — detailed model testing — is to provide semantic critiques for the BPM such as reachability analysis⁵, potential deadlocks and irrelevant nodes. As for syntactic critiques, it may be divided into semantic errors and semantic warnings. Table 4 and 5 summaries these critiques.

This detailed testing mechanism considers the logical meaning of the junctions, preconditions and actions of the processes together to verify the semantics of a BPM. It also figures out the possible execution results of the BPM.

At this level, not only the logical meaning of the junctions, but also the detailed preconditions and actions of the processes are considered. Neither level 1 nor level 2 consider trigger conditions, because all possible triggered results are listed at these two levels. The analysis of level 2 is based on the results from level 1. The analysis of level 1 enumerates all possible results of the BPM. Some detailed checking are added at level 2 to eliminate all impossible execution sequences while keeping all possible ones. A termination problem may also be detected at this level.

### 4.3 Instantiation of Business Scenario Level

At level 3, instantiation of business scenario – case study, the entity data is instantiated to a BPM. At this level, a user scenario adapted from AKT project⁶ in the “PC configuration” domain has been used to test these ideas. The BPM is shown in figure 3. It is different from level 1 and level 2 in that level 1 and 2 focus on overall business process model simulation, whereas level 3 focuses on a special case study. At level 3, a workflow system is executed and may create instances of processes at run time which depend on the given input data. The final flow is determined due to the attributes of this data. At this level, conflicting actions are also checked and are signaled by warning messages.

A workflow system directly mapped to FWFL is implemented at this level. A BPM described in FBPML is used at this level, and may be checked and run in this workflow system. The detailed system architecture and implementation are discussed in section 5. The three-level framework becomes complete because model behavior is tested and some model checking is provided in level 1 and 2, and a case is used to analyse the BPM at level 3.

---

<table>
<thead>
<tr>
<th>Error Types</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unreachability error</td>
<td>The precondition of the process can never be satisfied, and the process will never be executed.</td>
</tr>
<tr>
<td>Deadlock error</td>
<td>A process (A) waits for another process (B) and process (B) waits for process (A) at the same time.</td>
</tr>
<tr>
<td>Termination problem</td>
<td>Determining whether a workflow structure can reach a terminating state (ter Hofstede et al., 1996).</td>
</tr>
</tbody>
</table>

Table 4: Semantic Errors

---

<table>
<thead>
<tr>
<th>Warning Types</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Irrelevance warning</td>
<td>A process that does not use outputs from any other processes and does not produce inputs for other processes to use.</td>
</tr>
</tbody>
</table>

Table 5: Semantic Warnings

---

![BPM adapted from AKT Project](http://www.aktors.org/)

Figure 3: BPM adapted from AKT Project

---

⁵“Reachability analysis” is used to describe the construction of a state-transition model of a system from models of individual processes [Yeh and Young, 1991]

⁶Advanced Knowledge Technologies (AKT) Project is an IRC (Interdisclipined Research Council) project. This project is used to develop and integrate relevant AI techniques in the larger process of knowledge management. Various public web sites are available at: [http://www.aktors.org/](http://www.aktors.org/)
Case Study
The case used in this research is adapted from the AKT project [Chen-Burger, 2002a] in the “PC configuration” domain as shown in figure 3. A BPM and an entity database are needed for this workflow engine. The BPM is composed of a Process Specification, a Junction Specification and an Entity Specification. Definitions of processes are stored in the Process Specification. It is written in FWFL which provides direct input to the workflow engine. If a BPM is changed, the definitions of the processes written in FWFL change correspondingly. The purpose of Junction Specification is to record the logical connections of the BPM. Entity Specification stores the definitions of the entities and events (i.e. the event occurrences and entity occurrences are stored in the entity database). Because the user needs to determine how many steps that the flow needs to run, the flow can stop at a step which is stipulated by the user.

Example 1 in section 3.1 illustrates a part of the Process Specification. Junction Specification is shown in example 2.

Example 2:
```java
junc(m1,start,[],[],{a}).
junc(m1,link,[a],[b]).
junc(m1,or_split,[b],[c,d,e,f,g]).
junc(m1,and_joint,[c,d,e,f,g],[h]).
junc(m1,link,[g],[h]).
junc(m1,end,[h],[i]).
```

Example 3 illustrates a part of the Entity Specification.

Example 3:
```java
event_occ(e1, new, attribute(entity-id='e1',
  customer-name/"John",
  customer-orderNo/"1-06-07",
  customer-gender/male,
  customer-tel/"0131-5563432",
  spec([box-color/white,ioBoard-length/short,
    ioBoard-capability/(fast-_A-_B)],1),
  entity_occ(ioBoard,io1,valid,attribute(
    [ioBoard-type/io1,ioBoard-slot/3,
    ioBoard-capability/[fast-graphics-short]]))).
```

The BPM in example 3 describes a process flow for “PC configuration for the customer” as shown in figure 3. The trigger conditions of process c, d, and e are “true” (i.e. they may be triggered automatically), so they must be triggered when the flow is running. The user can easily understand the process because a PC must consist of these three parts. Processes “f” and “g” are different. They are triggered only if the customer has a special requirement. The Or_Split junction is used here. It indicates that not all following processes need be triggered. The And_Joint is used in the next junction. It indicates that all the triggered processes must be finished at some time (i.e. the workflow must find a solution for the customer). The testing result about a case is shown in example 4.

Example 4:
```java
Model: Modell
customer: (Mary, John)
```

Special Requirements: John’s special requirement is satisfied, Mary’s is not satisfied.

John’s special requirement is:
```java
sepc([case-type/c1,box-color/black])
```

The Execution Sequence:
```java
al-i-(e1-John),al-i-(e2-Mary),bl-i-(e1-John),bl-i-(e2-Mary),
f1-i-(e1-John),el-i-(e1-John),el-i-(e1-John),cl-i-(e1-John),
g1-i-(e2-Mary),el-i-(e2-Mary),dl-i-(e2-Mary),cl-i-(e1-John),
h1-i-(e1-John)
```

The Execution Result:
The process f1-i-e2-Mary cannot be executed. The reason may be the precondition cannot be satisfied or the action has errors! The requirement of customer -- Mary cannot be satisfied. The solution for customer -- John is [customer-orderNo/1,box-color/white, diskController-type/dc1,ioBoard-type/io1, processor-type/p3].

The flow is finished at step 10. We find that John’s special requirements include “case”, so processes c, d, e and f are triggered and finished. On the other hand, Mary’s special requirements include “case” and “box”, so processes c, d, e, f and g are triggered. But as Mary’s special requirement for box-color/black can not be found in the entity database, the solution can not be provided to Mary. The flow stops at step “10” in example 4 because the termination “step” defined by the user is “10”.

5 FWFL Workflow Engine
5.1 System Architecture
The system architecture of the FWFL workflow engine is shown in figure 4.

![System Architecture of the FWFL Workflow Engine](image)

A BPM and an entity database are needed for this workflow engine. The BPM is composed of a Process Specification, a Junction Specification and an Entity Specification. The definitions of the processes are stored in the Process Specification. It is written in FWFL which provides direct input to the workflow engine. If the BPM is changed, the definitions of the processes written in FWFL is changed correspondingly. Because of the direct mapping, it does not require much effort to deal with these changes. The purpose of the Junction Specification is to record the logical connections of the BPM; it also follows the FWFL. The Entity Specification stores the

7 We construct a BPM here, but FBPL may also be used to construct a manufactory process model.

8 An entity is a data stored in the entity database.
definitions of the entities and events. Event occurrences and entity occurrences are stored in the entity database.

The user needs to determine how many steps that the flow needs to run. The flow may stop at a step which is stipulated by the user. After accepting the input data, the workflow engine starts running the processes based on the definitions of the processes and the BPM. It checks the triggered events first and triggers all the processes whose trigger conditions are satisfied. The logical meanings of the junctions direct the flow.

When processes are triggered, instances of those processes are created and executed. In each tick\(^9\), all the possible instances are run within the given time until they are finished. Sometimes some instances may not be executed and finished in the current tick; the workflow engine keeps them until the next tick. The engine rechecks all remaining instances and runs them again in the next tick. This procedure loops until the flow reaches the step defined by the user at the beginning or until it reaches the “END” of the BPM. The time and flow state are updated in each tick, thus the user can know the time point and flow state in each step.

BPM 2002 Market Milestone Report[Group, 2002] classifies workflow processes into three categories “process-to-process”, “person-to-process”, and “person-to-person”. The FWFL workflow engine implemented in this research captures formal descriptions of the processes and provides a structured method to capture the business process. In addition to handling “person-to-person” or “process-to-process” workflow processes, it also provides interaction between the user and the workflow system. As well it can deal with “person-to-process” workflow processes because it has some exception handling so the user can choose when he/she is asked for input data from outside the workflow system. In most situations, however, the workflow still works automatically.

There are two types of interaction in the FWFL workflow system that deal with the “person-to-process” workflow process. First, the workflow engine asks for new values (or updated values) from the user. In this case, the flow may stop and wait for input from the user. Second, the workflow engine may also detect conflicting actions which happen when different instances try to deal with the same entity data simultaneously. Warning messages are provide to the user, and the system asks for a decision.

Figure 5 shows the flowchart of the FWFL workflow engine which is implemented and based on this workflow meta-interpreter.

Table 6 shows the summary of the predicates used in the workflow system (meta-interpreter).

<table>
<thead>
<tr>
<th>Predicate Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute_flow</td>
<td>Control the whole process flow</td>
</tr>
<tr>
<td>check_event</td>
<td>Check the new event</td>
</tr>
<tr>
<td>do_junction_process</td>
<td>Execute the junctions of the BPM</td>
</tr>
<tr>
<td>execute_process</td>
<td>Execute the instances of the processes</td>
</tr>
<tr>
<td>update_time</td>
<td>Update the time point</td>
</tr>
<tr>
<td>update_step</td>
<td>Update the step counter</td>
</tr>
<tr>
<td>flow_state</td>
<td>Indicate the flow state at time T</td>
</tr>
<tr>
<td>check_mstate</td>
<td>Check the model state</td>
</tr>
</tbody>
</table>

\(^9\)A tick indicates a time point. It is a counter in the workflow system (i.e. after all the possible processes are executed properly, the tick increases by one.)

5.2 State Transition and Dynamic Behaviors

Once a workflow is implemented, we need to monitor its progress. We can do this by checking the status of a workflow [Georgakopoulos et al., 1995]. The FWFL workflow system records the flow state using the predicate *flow state(F state, T)* which indicates the flow state at time

![Figure 5: Flowchart of the FWFL Workflow Engine](image)

Figure 6 shows an example of the state transition of the workflow system.

![Figure 6: State Transition of the FWFL Workflow Engine](image)

In figure 6, suppose we have a simple BPM:

- junk(m1, start, [a, [b]]).
- junk(m1, link, [a], [b]).
- junk(m1, end, [b], []).

and two trigger events at time point t1. What will be the state transition? In this case, the initial state is “flow state([[]], t0)”. The model state indicates the instances of the BPM for all trigger events. At the initial time, no trigger event occurs, so no instances of the BPM are created. The initial model state
is "[I]". When two events – “customer-John” and “customer-Mary” are triggered, two instances of the model are created:

[start/john/[a]/[b],[link/john/[a]/[b]],
[start/mary/[a],[link/mary/[a]/[b]]]  

After the instances of the model are created, the flow starts to run. The instances of the first process for each model may also be created and executed as long as the preconditions are satisfied. In this example, the precondition of process instance – “customer-John” is satisfied but the precondition of process instance – “customer-Mary” is not satisfied. Perhaps this is because the required input data is not available immediately. The instance of the process – “customer-Mary” is left in the execution queue and waits for data. These kinds of instances are run again at a later time point. The whole process flow of “customer-Mary” may stay at this stage until this instance is finished. On the other hand, the instance of the first process – “customer-John” – is executed and finished at time point t1 (Tick 1). The instances of the BPM change to

[link/john/[a]/[b],[link/mary/[a]/[b]]]

The flow state records the executed result – [[a1-i-(e1-john)]] and t1, and the process agenda stores the remaining instance – [a1-i-(e2-mary)] which may be executed later. The data created have been saved in the entity database, and that data may also be modified or removed later. The flow continues running, and the flow state changes at each stage until it reaches the end of the flow. There are two ways to reach the end of the process flow. First, when “ModelState” is an empty list and the “process agenda” is also an empty list, it indicates that the whole business process has executed completely. Second, the flow reaches a step defined by the user. In this case, the whole business process may or may not be finished.

5.3 Validation and Verification to Workflow System

Errors and warnings may happen in two situations: First, two or more instances try to deal with the same entity data. Second, the instances may be run again at a later time point. The whole process flow of “customer-Mary” may stay at this stage until this instance is finished. On the other hand, the instance of the first process – “customer-John” – is executed and finished at time point t1 (Tick 1). The instances of the BPM change to

[link/john/[a]/[b],[link/mary/[a]/[b]]]

The flow state records the executed result – [[a1-i-(e1-john)]] and t1, and the process agenda stores the remaining instance – [a1-i-(e2-mary)] which may be executed later. The data created have been saved in the entity database, and that data may also be modified or removed later. The flow continues running, and the flow state changes at each stage until it reaches the end of the flow. There are two ways to reach the end of the process flow. First, when “ModelState” is an empty list and the “process agenda” is also an empty list, it indicates that the whole business process has executed completely. Second, the flow reaches a step defined by the user. In this case, the whole business process may or may not be finished.

<table>
<thead>
<tr>
<th>Error Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type I</td>
<td>Two or more different instances add, delete or update the same entity data at the same time.</td>
</tr>
<tr>
<td>Type II</td>
<td>One or more instances refer to entity data and others try to add, delete or update it.</td>
</tr>
</tbody>
</table>

Table 7: Error Types of the FWFL Workflow Engine

When the workflow engine encounters these cases, the flow stops, a message is displayed and the system waits for a response from the user. Example 1 shows the message for a Type I warning. It shows that the instance of process b-i-id(e1-John,e1-John) is already delayed for more than n-ticks (time). The system will show the warning message to ask whether this delayed instance of the process should be kept or deleted. The user then makes a decision taking into account the impact of the decision. Serious problems may occur in this case. The flow may not be able to continue.

Example 1:

There is a delay in Process=> b-i-id(e1-John,e1-John)
Do you want to continue?(y/n)

Continue......

Keep Process=>b-i-id(e1-John,e1-John)
There is a delay in Process=> b-i-id(e1-John,e1-John)
Do you want to continue?(y/n).

delete current process......

6 Complexity of Business Process Models

The complexity of the BPM is discussed in this section. The complexity we calculate here is the complexity of the verification of the program using simulation/world state stepping techniques. The result shows that this program will be in practice need to carry out an exhaustive search to execute of all the possible permutation enactments of a business process model and find if there is any inconsistency. It also leads to a conclusion that “Or_split” junctions make a process model substantially more complicated as it allows rapid growth of different dynamic behaviours. In the following paragraphs we will show how we calculate the complexity of this program.

6.1 Complexity of a Single Model

In this section, we will calculate the complexity of a single model. Definition of the variables used in the following complexity analysis are:

- n: The number of branches from an And- or Or-Split junction.
- m: When two models (such as figure 1 (a)+(d), (a)+(c), (b)+(c), (b)+(d)) are connected, n is the number of branches from the first model and m is the number of branches from the second model.

An “Or-Or” model is used to illustrate how to compute the complexity. Suppose an “Or_split” junction has n branches. It is possible that all n branches, or only n-1 branches, are being triggered. In total, there are n + (n−1) + (n−2) + ... + 1 different possible triggered results for each triggered process set, the execution sequence may also differ. The possible permutations of these processes should be considered at the same time. The possible execution sequences of the “Or-Or” model are therefore:

\[ \sum (each\ possible\ triggered\ result \times all\ permutations\ of\ these\ triggered\ processes) \]

The second term in the product comes from:

\[ \sum (the\ number\ of\ the\ finished\ processes\ before\ the\ second\ junction \times permutation\ of\ the\ remaining\ processes) \]
Based on the result computed by “Mathematica”. We find that when “n=10”, the value of formula 1 is $88 \times 10^7$ which is a very big number. The complexity of this kind of BPM is high and has a factorial rate of growth [Bard, 2001]. It is impossible to list all the possible execution sequences when a BPM becomes so complex. Table 9 lists the complexity of four different types of BPM.

As a result, we find the “Or-Or” model is the most complex model as it has the highest complexity. The “And-And”, “And-Or” and “Or-And” are special cases of the “Or-Or” model. The “And-And” model case occurs when n processes are triggered at the same time and all the triggered processes must be finished before executing the following process of the next junction. The “And-Or” model case occurs when n processes are triggered at the same time and at least one of the triggered processes is finished at a later time point. The “Or-And” model is another special case when all triggered processes must be finished before executing the following process of the next junction. The following results are also found:

- The Or_split and Or_Joint have the greatest influence on the complexity. For instance, the complexity of “And-Or” is n times higher than “And-And”.
- The complexity of these types of BPMs is very high, and it is difficult to carry out all of the possible execution paths.

### 6.2 Complexity of Combination Models

In this section, we try to combine some of the previous models and compute the complexity of them. They are categorized as:

1. A model finishing with an And_Joint junction.
2. A model finishing with an Or_Joint junction.

A model finishing with an And_Joint junction is considered first because it enables possible execution sequences of one independent model. In general, the complexity of this case is computed as:

$$O(n!)$$

Models that finish with an Or_Joint have many possible execution results, especially when more than one Or_Joint is combined. These models may become very difficult to carry out.

A simple example – the combination of And-Or and And-And models – is used to describe the complexity. An assumption has been made to simplify this problem. We also remove the assumption to show the extent of the complexity. The assumption is that all triggered processes must be finished before the final process of each connected model (whereas in a real process model, an unfinished process may “propagate” and execute parallel to a process in the latter model).

Based on this assumption, the complexity of this problem is:

$$
\sum_{k=1}^{n} \frac{n!}{(n-k)!} (m + n - k)! (\text{formula 2}) \approx O(m + n)!^{11}
$$

As a result, formula 2 only provides the complexity under the assumption. If we relax the assumption, an “assumption term” has to be added. Then the result is:

The previous formula + Those cases in which some processes are executed after the final process

It will be larger than formula 2. Its complexity remains in $O(n!)$. The assumption does not have a big influence on the complexity.

As n! has a factorial rate of growth which is bigger than polynomial rates, [Garey and Johnson, 1979] also shows that an NP-complete problem needs more than polynomial time to solve (i.e. it needs exponential time or greater). The growth rate of n! is bigger than the growth rate of $k^n$, so we know that the business process problem is at least an NP-complete problem. [Chen-Burger, 2002b] indicates that workflow systems may be required to handle over 300 processes based on a real military BPM. The possible execution sequences of such a large model may be more than $60! = 8.3 \times 10^{61}$. If there are many different types of the junction in the BPM (i.e. if there is a total of 300 processes and there are 5 junctions, then each junction will have approximately 60 branches). So the number of possible results may be enormous if we do not consider the detailed semantics of the processes.

### 7 Other Related Work

In order to show the difference between FBPML + FWFL and other workflow process languages, the application of Petri nets to workflow management [van der Aalst, 1998] is compared in this section. The most significant difference between “FBPML + FWFL” and this research is that “FBPML + FWFL” has a formal business process modeling mechanism which separates the business and the implementation logic. Hence the workflow system is more flexibly reactive to a dynamic environment.

A Petri net that models a workflow process definition is called Workflow net (WF-net). Comparing the formal method (FBPML + FWFL) that we provide, and Petri net (WF-net), the following results have been found (the details of WF-net are not explained here, we focus only on the comparison between them):

- Because a Petri net is a process modeling technique, it focuses on workflow processes which are designed to handle cases (called instances in FWFL). However, it does not focus on resources, such as people, machines or organization units, which may be involved in the workflow system. The concept of “Role” is ingrained in FBPML+FWFL in order to represent resources and use them in the BPM.

<table>
<thead>
<tr>
<th>Business process model types</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>And_split and And_joint model (And-And model in figure 1 (a)+(c))</td>
<td>$O(n!)$</td>
</tr>
<tr>
<td>And_split and Or_joint model (And-Or model in figure 1 (a)+(d))</td>
<td>$O(n \cdot n!)$</td>
</tr>
<tr>
<td>Or_split and And_joint model (Or-And model in figure 1 (b)+(c))</td>
<td>$\approx O(n!)$</td>
</tr>
<tr>
<td>Or_split and Or_joint model (Or-Or model in figure 1 (b)+(d))</td>
<td>$\approx O(n \cdot n!)$</td>
</tr>
</tbody>
</table>

Table 9: The Complexity of Different BPM Types

---

11 When m=0, this formula represents the complexity of the “And-Or” model.
• Tasks are modeled by transitions in Petri net, whereas in FBPML, activities are used to describe tasks. In Petri nets, cases are modeled by tokens, whereas in FBPML + FWFL, instances are used to describe cases.

• In Petri net, each condition is modeled by a “place” and it may represent a precondition or a postcondition of the process. In FBPML + FWFL, logical meanings of the junction and preconditions in the attributes of a process replace the usage of “place”.

• To define a process and a BPM, FBPML + FWFL has a formal, declarative semantics. A process described in WF-net does not have clear definitions of its attributes.

• To represent different tokens (cases or instances), there are two ways to represent and list them in Petri net (WF-net): 1. Use different colors in a high-level Petri net. 2. Use instances to describe them. The latter is the same in FBPML + FWFL, but instances in FBPML+FWFL are not listed in the BPM. Using FWFL, they are only described as instance occurrences inside the workflow engine.

• High-level Petri net has a time extension to describe the temporal behavior of the system. In FBPML + FWFL, “Precedence-Link” is used to indicate a temporal constraint between two processes. It is similar to Petri net.

• High-level Petri net provides a hierarchy construct called “subnet” which can be used to structure large processes. In FBPML + FWFL, a high-level process can be divided into two different types of low-level sub-processes (decomposition and alternation) representing a similar notion.

• Both Petri net and FBPML+FWFL have a “trigger” notion. The trigger condition is clearly defined as an attribute of a process in FWFL. The user may formally describe the trigger condition of this process. The declarative description in FWFL makes the trigger condition easy to understand.

In Petri net (WF-net) the trigger concept is distinguished by different symbols and focuses on the enabler (Automatic, User, Message, Time) which triggers the process. It does not focus on the data or world state conditions. In FBPML + FWFL, trigger conditions are flexible; the user may define any trigger condition as long as he/she follows the language.

• Junction notions used in Petri net (WF-net) are modeled by ordinary transitions. Transitions are treated as control tasks. In FBPML + FWFL, each junction has its own formal definition. Logical meanings of \textit{And-Split} and \textit{And-Joint} are the same in FBPML+FWFL and Petri net (WF-net). However, in Petri net (WF-net) the “OR-split” is categorised into “implicit OR-split” and “explicit OR-split” (the same as “OR-join”). Although this may provide a clear definition of workflow modeling, it makes notation more complex. FBPML + FWFL does not distinguish this case so that the modeling language is easy to learn because it is more natural. Users do not need to remember specific meanings of notations.

• The WF-net is designed to require that there are no dangling tasks and/or conditions in WF-net. Every task (transition) and condition (place) should contribute to the processing of cases. The concept is similar to “irrelevant nodes” in our three-level framework.

• Triggers and workflow attributes are removed when analysing workflow in Petri net (WF-net). In our three-level framework, we also only consider trigger conditions at the level 3 – case study. The reason is the same as that described in [van der Aalst, 1998]. In our framework, in order to actually simulate possible model behaviors, some semantics and details of the process are considered in the analysis. This is different from Petri net (WF-net) analysis in which all verification and validation are done at a graphical level; although it has more formal definitions for verification than ours.

8 Conclusions

In this research, we try to bridge the gap between \textit{Enterprise Modelings} (EMs) and \textit{Software Systems} in order to provide support where EMs are used as a part of KM initiative. This gap exists primarily between the capabilities for gathering and presenting knowledge, and the capability for performing semantic-based automatic manipulation of this knowledge. Formality needs to be introduced to the informal or semi-formal enterprise modeling paradigm to provide precision and enable automatic support. A workflow system is built to bridge this gap, and allow domain knowledge to be checked for consistency and correctness during enterprise modeling. The following conclusions are made:

• FBPML is a merger of two standardised process modeling languages: IDEF3 and PSL. The benefit of merging the two languages is that the former has graphic notation but lacks formal process conceptualisation, whereas the latter provides formal process theory without any visualisation. Although the two languages are not equal, their core concepts overlap. Such core concepts are included in FBPML, and are carefully disposed so that the consistency of FBPML is maintained.

• The graphical notation used in FBPML is intended to make it easier for those unfamiliar with predicate logic to describe models in the language. Although we have not conducted extensive empirical evaluations of the FBPML graphical language, it is very similar in style to other graphical process modelling languages that have achieved widespread use. Our aim here is to conform

---

12A Petri net extended with color, time and hierarchy is called high-level Petri net.

13The author also says that there is no compelling need to distinguish between implicit and explicit OR-joins, but there is for ‘OR-split’.

14The reason is that it is impossible to model the behavior of the environment completely.
to currently accepted modelling practices. The graphical language in FBPML, however, translates automatically to a predicate logic description that supports both a declarative reading (helpful in checking the logical consistency of the model) and multiple operational readings (allowing different forms of enactment engines to execute a process model by interpreting the model description). As usual, we make our enactment engines generic for all forms of FBPML model so that we can freely change the declarative description without needing to alter the enactment engines.

- The workflow meta-interpreter is based on FBPML + FWFL. It accepts input specifications and executes the BPM directly. It may be implemented as a “person-to-process” workflow system so that some validation and exception handling may be confirmed by the user. This makes it a more useful tool for the user due to this flexibility. Most of the time, flow is executed automatically. Thus we have a flexible way to execute a business process flow in a dynamic environment.

- Our three level-framework provides a thorough test which is useful in analysing a BPM. At level 1, after carrying out syntactic critiques, the appropriate topology for a complex BPM is mapped out. At level 2, the details of the process are considered, and features from this topology are inferred. Because of this explicitness due to inference of the model, some impossible execution results are eliminated after performing semantic critiques. Problem size is therefore reduced. Level 3 corroborates inference of the model, some impossible execution results may be confirmed by the user. This makes it a more useful tool for the user due to this flexibility. Most of the time, flow is executed automatically. Thus we have a flexible way to execute a business process flow in a dynamic environment.

- Our three level-framework provides a thorough test which is useful in analysing a BPM. At level 1, after carrying out syntactic critiques, the appropriate topology for a complex BPM is mapped out. At level 2, the details of the process are considered, and features from this topology are inferred. Because of this explicitness due to inference of the model, some impossible execution results are eliminated after performing semantic critiques. Problem size is therefore reduced. Level 3 corroborates inference of the model, some impossible execution results may be confirmed by the user. This makes it a more useful tool for the user due to this flexibility. Most of the time, flow is executed automatically. Thus we have a flexible way to execute a business process flow in a dynamic environment.

Acknowledgements

This work is partially supported under the Advanced Knowledge Technologies Interdisciplinary Research Collaboration, which is sponsored by the UK Engineering and Physical Sciences Research Council under grant number GR/N15764/01.

References


