School of Informatics, University of Edinburgh

Centre for Intelligent Systems and their Applications

Informal Semantics for the FBPML Data Language

by

Jessica Chen-Burger

Informatics Research Report EDI-INF-RR-0154

School of Informatics **October 2002**
http://www.informatics.ed.ac.uk/

Informal Semantics for the FBPML Data Language

Jessica Chen-Burger

Informatics Research Report EDI-INF-RR-0154

SCHOOL *of* INFORMATICS
Centre for Intelligent Systems and their Applications

October 2002

unpublished.

Abstract :

Keywords : business process modelling, knowledge management, workflow systems

Informal Semantics for the FBPML Data Language

Jessica Chen-Burger
October 2002

This document informally defines the data language that has been used to describe data in the Formal Business Process Modelling Language (FBPML).

A Brief Introduction to FBPML

FBPML is a merging and adaptation of two recognised process modelling languages (PML): PSL [7] and IDEF3 [8]. PSL provides formal semantics for commonly shared process modelling concepts as well as theories such as situation calculus that support the use of such concepts. As it is designed to be an interchange language between different process languages, it covers the core concepts in PML, but does not provide visual notations or model development methods.

IDEF3 is originated from manufacturing environment and is one of the richest methods available in the process modelling community. It provides visual notations and a rich modelling method. Nevertheless, its semantic is informal and its models therefore may be open to interpretation.

It is therefore useful to combine the two different methods, i.e. to gain advantages from the rich modelling method from IDEF3 and provide it with a formal semantics and necessary theories from PSL, so that reasoning mechanism and formal analysis may be carry out on those models. FBPML is such a modelling language.

FBPML Data Language

The FBPML Data Language (FBPML-DL) is first-ordered. This document uses the syntactic convention that has been used in Prolog to present the FBPML-DL.

Four parts are included in the FBPML data language:

- Foundational Model;
- Core Data Language;
- Extension Data Language;
- Meta-predicates of FBPML.

The foundational model includes the use of logic [1][2] and the fundamental concepts of set theory that has been used primarily via the class relation, *subclass_of*, and membership function, *in/2*. Mathematical theory on the manipulation of integer, rational and real numbers are also included.

Concepts, predicates and functions of background theories that are used in the language are included in the Foundational Model and described in Part I. Primitive predicates are introduced here and will be used to define other predicates.

Core data language introduces core predicates and functions to the FBPML data language. They are fundamental and common concepts that may be used for all applications of the language. Their definitions are given in this documents using natural language. These semantics may also be

defined in terms of foundational or core FBPML language, given that they have been defined properly and have not been recursively defined – i.e. a definition path that leads to itself. Core FBPML is given in Part II of this document.

The extension data language includes predicates and functions that are in additional to the core data language that has been provided by FBPML. They are usually defined by the user and are often application and domain-dependent. This information is given in Part III.

Meta-predicates give definitions for other predicates and may define axioms of an application model. They are defined in Part IV.

Part I. Foundational Model: Concepts, Predicates and Functions for the Data Language:

1. Concept: number/1

This language allows all numbers that may be expressed using Prolog language.

2. Concept: constant/1

A valid constant may be of any number or any unbroken sequence of symbols beginning with a lower-case letter, i.e. a-z [1].

3. Concept: variable/1

A valid variable is denoted by an unbroken sequence of symbols start with an upper-case character, i.e. A-Z, or underscore, _, followed by any numbers of characters, underscores or numbers.

4. Concept: list/1

A list is a (Prolog) list consisted of an arbitrary numbers of constants, variables, numbers, lists and valid FBPML terms. A list may also be an empty list.

5. Concept: string/1

A string is a (Prolog) string of arbitrary length of characters, including the use of a-z, A-Z, numbers, underscores, hyphens, brackets, semi-columns and columns, separated by spaces that are being quoted by two single quotes. A string may also be an empty string, i.e. ‘’.

6. Concept: term/1

A valid term is a constant, variable, number, list, string or another predicate that has been defined by the foundational, core and extension data language of the FBPML.

7. Concept: in/2

in(M, X) is the set logical operator “membership” to indicate that M is a member of the Set X. In FBPML, a set is captured using the concept *List*. An axiom can therefore be described below:

$in(M, X) \Rightarrow term(M) \wedge list(X)$

8. Function: atom/1

atom(Term)

Atom/1 is a function that returns true, if Term is a valid FBPML constant; otherwise, it returns false.

9. Function: atomic/1

atomic(Term)

Atomic/1 is a function that returns true, if Term is a valid constant or a number; otherwise, it returns false.

10. Function: compound/1

compound(Term)

compound/1 is a function that returns true, if the Term is a valid term to the FBPML language with arity > one, i.e. it is either a list or a structure. It returns false, otherwise.

11. Function: is_list/1

is_list(X) return true, if X is a valid list to the FBPML language; false, otherwise.

12. Function (Truth-Functional Connective): or/1

or(X)

X is a list of terms that may have truth or falsity value separated by commas, where each comma denotes a conjunction. The list itself is written in Conjunctive Normal Form.

13. Function (Truth-Functional Connective): not/1

not(X)

not/1 denotes the negation of the truth-value of the term X.

14. Function (Truth-Functional Connective): , /2

[X, Y] where the comma ","denotes the conjunction of the 2 terms X and Y. This notation is only used within a list.

As normal logical connectives have been used in the language, the normal convention for them has also been adopted. The standard precedence ordering for connectives is therefore given below: [1]

Implication and equivalence connectives have the highest precedence; conjunction and disjunction have the next highest precedence; negation has the lowest precedence. In a formula where several connectives are presented, the connective with the higher precedence is the *Principal Connective* because it binds the formula together. Principal connective is applied last (or later) in a formula. In cases where two operators of equal precedence are used and there is no bracketing to impose an explicit ordering on the operators, then the operators furthest to the right is taken as the principal operator.

A Principal Connective (or Operator) in a formula, therefore, is the Operator that binds together the entire formula and/or has the higher precedence. [5] gives a brief discussion on the use of logical connectives and precedence.

15. Function: =/2

X = Y
=/2 returns true if X unifies with Y in Prolog terms.

16. Functions: </2, >/2, >=/2, =</2

X < Y, X > Y, X >= Y, X =< Y denotes the mathematical functions, smaller than, greater than, greater than or equals to, and smaller than or equals to, that operates on numbers the same way as mathematical theory for integer and real numerical theory. Since this theory is obvious, details are not repeated here.

17. Quantification: forall/1

forall(X) denotes the universal quantification for X in a formula.

18. Quantification: exist/1

exist(X) denotes the existential quantification for X in a formula.

19. Constant: true/0

The logical symbol, true, is always true.

20. Constant: false/0

The logical symbol, false, is always false.

21. Concept: constraints: e.g.

dynamic_constraint(Quantification, Precondition, Conclusion)

static_constraint(Quantification, Precondition, Conclusion)

Two types of constraints are provided, the static and dynamic constraints, and are described in details in Part II: the Core Predicates. The logical connectors are used in the context of the static_constraint/3 and dynamic_constraint/3 predicates.

Part II. Core Predicates for the Data Language:

1. class/1

class(X): X is a class that X is a string describing the name of the class.

2. subclass_of(Class_1, Class_2)

subclass_of(Class_1, Class_2)

Class_1 is a sub-class of Class_2. This is to say that all instances of Class_1 are also instances of Class_2. The relation subclass_of is consistent with AKT Support Ontology class relation "Subclass-of" [3] .

3. instance/1

instance(X) denotes that X is an instance of the represented domain, X may be a tangible or an intangible thing.

4. property_name/1

property_name(Y) denotes that Y is a property name for some classes in the domain.

5. property_value/1

property_value(Z) denotes that Z is a property value for a property of some classes, where Z is a term or a list.

6. attribute_name/1

attribute_name(Y) denotes that Y is an attribute name for some instances, where Y is a string of characters that may include underscores and hyphens.

7. attribute_value/1

attribute_value(Z) denotes that Z is an attribute value for a property of some classes, where Z is a FBPML term.

8. class_property/3

class_property(Class, Property_name, Property_value)

This predicate stores the property of a class. The variable *Class* stores the class name, *Property_name* stores the name of the property, and *Property_value* stores the value of this property. Class properties and values are static and are common for all instances in the class. The property value is therefore not possible to be altered by instances of the class during dynamic operations.

class_property(Class, Property_name, Property_value)

=>
class(Class) ^
property_name(Property_name) ^
property_value(Property_value)

9. class_rel/3

class_rel(Rel, Class_1, Class_2)

Rel, Class_1, Class_2 are all a string of characters.

Class_rel/3 defines the relationships between two classes, Class_1 and Class_2, that is to say that class (Class_1) holds the relationship (Rel) with class (Class_2). The class_rel/3 predicate may only be used to define class level relationships.

The type of relationships, defined in Rel, may be defined by the user and are domain-depended. The actual relationship between two instances is stored in instance_rel/3. Currently, the class_rel/3 defines only binary relationships, this is under the assumption that all relationships with multiple participants may be re-organised and represented in multiple binary relationships.

10. instance_of/2

instance_of(Instance, Class):

instance_of/2 denotes that the Instance is an instance of the class Class.

The instance of a class, e.g. instance_of(accept_1, accept_student_application) denotes that accept_1 is an instance of class type, accept_student_application.

instance_of(Instance, Class)
=>
instance(Instance) ^
class(Class)

11. instance_att/3

instance_att(Instance, Att_name, Att_value).

This predicate stores an attribute value, Att_value, for attribute, Att_name, of the corresponding instance, Instance. The domain of Att_value is defined in the predicate att_domain/3.

∀ Instance, ∀ Att_name, ∀ Att_value. ∃ Class.
instance_att(Instance, Att_name, Att_value)
=>
instance(Instance, Class) ^
attribute_name(Att_name) ^
attribute_value(Att_value)

12. att_domain(Class, Att_name, Domain_for_att_value)

The attribute, Att_name, that is described in the predicate above indicates that Att_name is an attribute for all instances in the class, Class. Each instance, however, will have its own Att_value, for the attribute. The attribute value is recorded in the predicate instance_att/3.

The field Domain_for_att_value defines the domain for attribute value of attribute Att_name. This domain definition is applicable for all Att_name for all instances of the class, Class, where the domain for attribute value may be a domain of string, integer, term, or a numerated list of constants. The domain may also be further defined and refined in separate axioms, e.g. a domain may be restricted to a range of integers, or alphabets in an interval. It may also specify a pre-determined list of things, e.g. [high, medium, low], or a sub-list or subset of a pre-determined list or set of things. It may also be a type of structured terms; or a particular kind of template/schema which includes variables that may be instantiated by values at run-time.

$$\forall Class, \forall Att\_name, \forall Domain.\ att\_domain(Class, Att\_name, Domain)$$
$$\Rightarrow$$
$$class(Class) \wedge attribute\_name(Att\_name) \wedge$$
$$string(Domain) \vee integer(Domain) \vee list(Domain) \vee term(Domain) \vee compound(Domain)$$

13. instance_rel/3

instance_rel(Rel, Instance_1, Instance_2)

Rel, is a relationship that has been defined in the class_rel(Rel, Class1, Class2).

$$\forall Rel, \forall Instance\_1, \forall Instance\_2.$$
$$instance\_rel(Rel, Instance\_1, Instance\_2)$$
$$\Rightarrow$$
$$class\_rel(Rel, Class\_1, Class\_2) \wedge$$
$$instance\_of(Instance\_1, Class\_2) \wedge$$
$$instance\_of(Instance\_2, Class\_2)$$

instance_rel/3 defines the relationships between two instances, Instance_1 and Instance_2, that is to say that Instance_1 holds the relationship (Rel) with Instance_2. The instance_rel/3 predicate may only be used to define instance level relationships.

14. static_constraint/3

Static_constraint/3 defines a property that a valid goal state "must" have for a designated problem domain. A static_constraint is a "hard" constraint. As constraints are often domain- and application-dependent, they are often only meaningful when associated to a particular problem domain. In a problem domain, several static constraints may be applicable and defined by the user.

static_constraint(Quantification, Pre_condition, Conclusion)

Three variables are included: Quantification, Precondition and Conclusion, and all of them are a list.

The variable Quantification stores the quantification and the corresponding variables, i.e. forall/1 to denote the universal quantification and exist/1 to denote the existential quantification.

Typing information of each variable is stored in the variable Precondition. The variable Precondition and Conclusion together indicate a conditional constraint. It reads as following: "given the Quantification such that, if the Precondition is true, then the Conclusion must be true", i.e. for all valid goal states for the corresponding problem domain. Clauses that are stored both in the Pre_condition and Conclusion are written in Conjunctive Normal Form separated by commas or "or" predicates, where a comma, ",", represents a conjunction, and the "or/1" predicate represents a disjunction, the not/1 predicate denotes negation.

The variable Pre-condition and Conclusion of static_constraint/3 may be nested, i.e. it may take another or several static_constraint/3 as an argument.

15. dynamic_constraint/3

Dynamic_constraint/3 defines the property that a goal state "may" have for a designated problem domain.

dynamic_constraint(Quantification, Pre_condition, Conclusion).

The grammar rules that have been applied for arguments *Quantification, Precondition* and *Conclusion* in the predicate static_constraint/3 are also applied here for the corresponding arguments.

Different to static constraints, dynamic constraints are "soft" constraints that may or may not be obeyed by the desirable goal states for the problem domain. Similar to static_constraint/3, the Pre-condition and Conclusion of dynamic_constraint/3 may also be nested, i.e. it may take another or several dynamic_constraint/3 as an argument.

16. Mathematical Functions: equal/1, greater_than/2, greater_equal/2, less_than/2, less_equal/2

The above predicates correspond to the mathematical notation of =, >, >=, <, =<, e.g. the notation of less_than(X, Y) denotes that X is less than Y, where both X and Y are of type number.

Part III: Extension for the Data Language: (Application Specific Predicates for the PC Configuration Domain)

1. instance_rel(has, Instance_1, Instance_2)

instance_rel(has, Instance_1, Instance_2)

This predicate describes the fact that Instance_1 contains Instance_2; in other words, Instance_2 is a part of Instance_1. Both Instance_1 and Instance_2 are instances of some types. In the PC configuration domain, Instance_1 is of type board, and Instance_2 is of type slot.

instance_rel(has, Instance_1, Instance_2)
⇒
instance_of(Instance_1, board) ∧
instance_of(Instance_2, slot)

2. class_rel(next_to, X, Y)

class_rel(next_to, X, Y)
⇒
subclass_of(X, slot) ∧ subclass_of(Y, slot)

The instance relation next_to describes that one slot may be physically next to another slot.

3. instance_rel(next_to, Slot1, Slot2)

The instance relation next_to describes that instance Slot1 and instance Slot2 are physically next to each other. Both Slot1 and Slot2 must be both instances of class type “slot”. This predicate denotes that Slot1 is next to Slot2 on a certain platform, e.g. on a certain motherboard, i.e. if the slot is defined associated with a motherboard, as it is the case in the PC configuration domain. This is also described in the below axiom.

instance_rel(next_to, Slot1, Slot2)
=>
instance_of(Slot1, slot) ^
instance_of(Slot2, slot)

4. class_rel(allocate_next_to, X, Y)

class_rel(allocate_next_to, X, Y)
⇒
subclass_of(X, board) ∧ subclass_of(Y, board)

5. instance_rel(allocate_next_to, X, Y)

instance_rel(allocate_next_to, X, Y)
⇒
instance_of(X, board) ∧ instance_of(Y, board)

6. class_rel(allocate, Board, slot)

class_rel(allocate, B, slot)
⇒
subclass_of(B, board)

This predicate indicates the class relationship between board and slot.

7. instance_rel(allocate, X, Y)

instance_rel(allocate, X, Y)
=>
instance_of(X, board) ^
instance_of(Y, slot)

This predicate indicates that the instance Instance_1 has been allocated to instance Instance_2, where Instance_1 is of type board, and Instance_2 is of type Slot.

8. class(total_cost)

instance_of(ID, total_cost)
instance_att(ID, amount, X)

class(total_cost) denotes a summation of cost for a given PC configuration solution, where ID is an unique identifier for the cost and X is a number based on a predefined monetary unit for the application domain. The exact elements that contribute to the cost are determined dynamically by the solution that has been derived based on the corresponding requirements.

9. class(solution)

instance_of(Instance, solution)
instance_att(+Instance,
solution_content,
[+Customer, +Requirement, +Old_sol, -New_sol, +Difference])

This predicate records solutions that have been created/found for the user requirement, Requirement. Variables in the solution_content are explained below:

- Customer is a String, i.e. a Customer Name or ID;
- Requirements is a List of customer requirements described using constraint predicates, i.e. dynamic or static constraint predicates;
- Old_sol is a List: it may be an empty list (when there is no previous solution) or a previous solution for the given requirements. To begin with when there is no previously solutions available, the Old_sol is an empty list, when a solution has been revised, this new solution is recorded in the New_sol;
- New_sol is a List that records the newly devised solution, given new requirements described in the Difference variable;
- Difference is a list of changes, [Add, Deleted], that contains two sub-lists of “added” and “deleted” predicates. It records all information about the differences between the new and

old problem space, e.g. the removal of "violated" constraints for the new solution and/or the added new "relaxed" constraints that make the new solution valid. If one updates the old problem space according to the Difference list, one may derive the new solution based on the new problem space. (The Difference variable has not been used in the PC configuration application yet.);

- The solution_content is used for communication between two systems. Values of all of the + signed variables in the predicate are not to be changed during the communication; whereas the – signed variables are un-instantiated that indicates a return value is expected from the receiver.

10. class(issue)

subclass_of(Issue, issue)

Using the Core Predicate, subclass_of, to indicate that a particular issue, Issue, is a type of issue. As the different types of issues are determined depending upon the application domain, Issue is a variable that will be instantiated with a string that describes the issue.

instance_of(Instance, Issue)

This predicate indicates a particular issue instance.

instance_att(Instance, status, S)

This predicate indicates the status of the issue instance, e.g. S may be created, cancelled, finished.

instance_att(Instance, priority, P)

This predicate indicates the priority (P) of the issue instance, e.g. low, high, medium.

instance_att(Instance, time, T)

This predicate indicates the time duration of the existence of an issue instance, T = Begin_time/End_time. Begin_time is the time when an issue instance is created, End_time is the time when it is cancelled or finished.

instance_att(Instance, communicator, (Requester/Requester_type, Provider/Provider_type))

The field communicator records the information about the two parties involved in the issue, i.e. the service requester (the party that raises the issue) and service provider (the party that tries to resolve the issue).

instance_att(Instance, issue_content, Issue_details)

The value of issue_content, Issue_details, describes the details of an issue. In the PC configuration application domain, it is a structured term that records information from a *solution instance*, as given below:

Issue_details = instance_att(+Instance,

solution_content,
[+Customer, +Requirement, +Old_sol, -New_sol, +Difference])

where details of this predicate is described in the corresponding instance_att predicate for solution in this document in point 5 class(solution).

The issue predicate described above is compatible issue that has been described in the I-N-C-A framework [4]. An I-N-C-A issue instance includes any issue types that requires to be resolved. The different types of issues are described in an "Issue Ontology" of the corresponding process model. A set of issues are known to the process model that the model may or may not have the appropriate means to deal with it. When a process model encounters an issue that it can not resolve, it normally passes it to others (e.g. agents) via its communication processes.[5] FBPML therefore is suitable to be used in a distributed environment.

11. Communication Language: class(issue_packet)

class(issue_packet)
instance_of(Instance, issue_packet)

The variable Instance records an unique ID for the issue_packet.

instance_att(Instance, issue_packet_content,
[(Sender/Sender_role, Receiver/Receiver_role),
[Issue_instance, Issue_name, Issue_status, Priority, Time,
(Requester/Requester_type, Provider/Provider_type),
World_state, Packet_content]
])

An issue packet is dynamically constructed that stores communication information with other parties. The content of an issue packet contains two parts: the header and detailed packet content. The header is a bracketed construct that describes the sender (ID) of the packet, the role of the sender, the receiver (ID) of the packet and the role of the receiver, i.e.

(Sender/Sender_role, Receiver/Receiver_role).

The detailed issue content is a list construct that records all of the details of an issue, i.e.

[Issue_instance, Issue, Issue_status, Priority, Time,
(Requester/Requester_type, Provider/Provider_type),
World_state, Issue_content].

Information about the corresponding issue such as *status, priority, time*, can be found and taken from the corresponding attribute predicates, e.g. instance_att/3, of the particular issue instance. *World_state* records the part of dynamic system state that is relevant to resolving this issue. *Issue_content* records the content of the issue. In the PC configuration example, it is the relevant solution(s) for resolving the issue. Details of Issue_content is given below:

Issue_content = instance_att(Solution_instance, solution_content,
[Customer,
Requirement_constraints,

Old_sol, New_sol, Difference]).

Content of this predicate is described in great details in instance_att(Instance, solution_content, Content), therefore is not repeated here.

Part III/1: Below describes domain classes that have been used in the FBPML: all of the below classes will have instances, and instance attributes predicates. As it is obvious for the reader, it is not enumerated. The definition for instance_of and instance_att predicates are given in Part II, Core predicates.

12. class(board).

This class is the superclass of all boards, except motherboard.

13. class(motherboard).

14. class(processor)

This is the class for CPU processor.

15. class(disk_controller)

This is the class for Hard Disk Controller.

16. class(io)

This is the class of boards that talks to IO devices.

17. class(option_I).

This is the (on-offered) board option 1, e.g. graphic card/board.

18. class(option_II).

This is the board option 2, e.g. graphic and video card/board.

19. class(option_III).

This is the board option 3, e.g. sound card/board.

Part IV: Meta-predicates:

1. def_predicate/2

def_predicate/2 is a meta-predicate that is used to introduce new domain-specific predicates. The new predicate may be defined using other existing previously defined predicates, such as foundational, core or extension predicates. The definition of a predicate however should not be recursive and must be grounded (eventually) on the foundational or core FBPML predicates, otherwise, errors and ambiguities may be incurred as a result. The def_predicate/2 is given below:

def_predicate(Predicate, Definition_for_predicate).

The variable Predicate stores the predicate (name) that are being defined by the Definition_for_predicate. The variable Definition_for_predicate is a list that defines the semantics of Predicate using first order logic statements. Grammar rules for those logic statements are the same as those defined for the field "Conclusion" for the predicate static_constraint/3.

2. axiom/2

axiom(Conclusion, Hypothesis)

The predicate axiom/2 defines properties of the already existing predicates. It does not introduce new predicates.

The axiom is described in the Hypothesis and Conclusion variables. The axiom predicate defines that if the Hypothesis is true, then the Conclusion is true.

The axiom itself is a horn-clause, i.e. there is only one single conclusion. Grammar for describing the Hypothesis is written in conjunctive normal form and is the same as it is defined for the Definition_for_predicate variable for the predicate def_predicate/2.

Reference:

[1] Dave Robertson, "An Introduction to Logic" Lecture Note, The University of Edinburgh. September 1992.

[2] Alan Bundy, "The Computer Modelling of Mathematical Reasoning", Academic Press Inc Ltd. 1983.

[3] Enrico Motta, AKT Support Ontology v1.0 and AKTive Portal Ontology v1.0, November 2001.

[4] Austin Tate, "<I-N-C-A>-Shared Model for Synthesized Artefacts", http://www.aiai.ed.ac.uk/project/ix/inca/.

[5] Yun-Heh Chen-Burger, Austin Tate, and Dave Robertson, "Enterprise Modelling: A Declarative Approach for FBPML", European Conference of Artificial Intelligence, Knowledge Management and Organisational Memories Workshop, 2002.

[6] Dave Robertson and Jaume Agusti, "Software Blueprints: Lightweight Uses of Logic in Conceptual Modelling", ACM Press, 1999, pp 41-43.

[7] C. Schlenoff and M. Gruninger and F. Tissot and J. Valois and J. Lubell and J. Lee, The Process Specification Language (PSL): Overview and Version 1.0 Specification, ISTIR 6459, National Institute of Standards and Technology, Gaithersburg, http://www.nist.gov/psl/, 2000.

[8] Richard Mayer and Christopher Menzel and Michael Painter and Paula Witte and Thomas Blinn and Benjamin Perakath, Information Integration for Concurrent Engineering (IICE) IDEF3 Process Description Capture Method Report, Knowledge Based Systems Inc. (KBSI), http://www.idef.com/overviews/idef3.htm, Sep, 1995.

Appendix I. The PC configuration domain model described in UML Class Diagram