Specification of Integrity-Preserving Operations in Information Systems by Using a Formal UML-based Language

Régine Laleau
CEDRIC-IIE(CNAM),
18 allée Jean Rostand, 91025 Evry, France
tel +33 1 69 36 73 47; fax +33 1 69 36 73 05
laleau@iie.cnam.fr
&
Fiona Polack
Department of Computer Science
University of York, York, UK
fiona@cs.york.ac.uk

November 6, 2001

Abstract

Information system specification requires careful consideration of data integrity. We present an approach to the definition of a variant of UML that allows the rigorous specification of operations and transactions which enforce data constraints. The approach to structural language definition, summarised in Appendix A, extends that used for conventional UML, comprising metamodels of the abstract syntax of structural aspects, with denotational semantics. We add operation semantics using a formal notation, B. Issues relating to the tool-assisted specification of integrity-preserving transactions are discussed.

Keywords: information system specification; UML; formal language definition; information system transactions; information system integrity.

1 Introduction

CEDRIC-IIE is engaged in a programme of research into the rigorous specification of information systems (IS). The long-term aim is to provide support tools to assist in the construction of formally-correct IS specifications, based on existing graphical notations. The group has already published its guideline-based approach for deriving a formal specification of an IS (in the B notation[1]) from object-oriented class and behavioural models[2, 3, 4, 5, 6]. This allows the formal investigation (proof, etc) of the IS specification, but cannot guarantee either the

correctness or the equivalence of the object-oriented models. To permit fully-rigorous IS specification, the modelling notations, as well as the existing translation rules, need to be rigorously defined. This is being undertaken for an IS-focused variant of UML.

UML is widely used in commercial software engineering. However, the language is still evolving. One problem for a user of UML is to determine the subset of notations which is relevant to their particular project or level of abstraction. The Object Management Group (OMG), which oversees the development of the UML, has recognised the need to allow specialisation of the language for different types of system. The mechanisms of extension are not static or agreed (see OMG's Requests for Proposals (RFPs), http://www.celigent.com/omg/adptf/wgs/uml2wg.htm). For the IS UML, we follow the general approach of the pUML group (eg[7, 8]), defining a small core language and extensions.

In general, the structural aspect of a system describes the system state. The behavioural aspect includes functional features, namely operations, and dynamic features, for example liveness or concurrency management. Because an IS manages large amounts of highly structured data, its specification depends on a well-defined structural model, with a large number of data constraints defining the integrity conditions of the IS. The structural aspect is an important determinant of the architecture of the final implementation. The functional aspect of the IS comprises transactions which correspond to user requirements and enforce the integrity constraints. In IS, dynamic features are of limited importance, compared to, for example, real-time systems. Their management is largely determined by the software used to implement the IS (eg, database management system).

We have already defined most of the structural aspects of UML required for IS specification. This work is summarised in Appendix A. In this paper, we show how the structural language definition can be extended to support the specification of integrity-preserving operations and transactions. First, we discuss approaches to the formal definition of a modelling language. We briefly explain the conventional approach using metamodelling to define the abstract syntax of structural and functional aspects, and a denotational semantics of the structural aspect. This approach is not appropriate for the semantics of the functional language, which we define using an existing formal language. We summarise the metamodel structures needed to express operations and transactions in IS UML, and explain how the resulting formalised language allows us to address issues that assist the designer in the specification of operations which enforce data integrity. A simple example shows how the practical specification of an IS can be achieved in the IS UML.

2 Approaches to the Definition of a Language

An ideal definition of a language such as UML should fix the meaning of core concepts and allow extension and specialisation for application-specific concepts. The visual representations (concrete syntax) and the meaning of concepts should be separable. This allows the use of different visual representations. An abstract syntax defines the correct use of the notations, independently of any concrete syntax, and is needed for tool development. It is also necessary to define the meaning of the concepts, the semantics.

In computing, language definition generally focuses on the meaning of textual programming languages. Semantics may be expressed in denotational, operational, axiomatic or algebraic ways (amongst others)[9]. A denotational approach is well-adapted to the expression of the semantics of a graphical notation, since it defines the symbolic representations of things, the meaning of the things denoted, and a mapping between the two.

2.1 Issues in Language Definition for UML

There are a number of current attempts to define the syntax and semantics of UML. These use a denotational approach to define core structural concepts (eg classes, associations, objects). For example, pUML[7] uses metamodels to define the UML concrete and abstract syntax, and the semantic domain of structural concepts. The denotational semantics are established by mappings between metamodels (figure 1). UML diagrams and OCL constraints are used throughout.

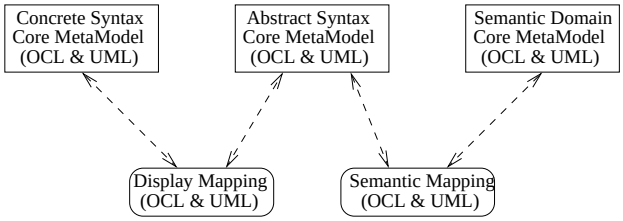

Figure 1: pUML Metamodelling Method(based on[7, Fig 1])

The pUML work includes a prototype tool implementing these definitions. This tool can verify UML class models (in diagrams and OCL) against the metamodel. For structural aspects, this tool could underpin a formalised UML, in which the graphical notations form a usable interface to a precise specification. However, such a tool cannot provide assistance in the design of a system.

These existing attempts are incomplete, and can only capture the syntactic representation of the

functional aspects of a system. The denotational approach is not sufficient for the modelling of operation contents, namely preconditions and the definition of what the operation does (eg postconditions, substitutions, etc). Moreover, it would be difficult to apply a denotational approach to the behavioural semantics of operations. For example, a semantic domain for operations might have to establish the meaning of an operation in terms of its execution.

For IS UML, we present an alternative approach, based on a formal language with well-defined semantics. We retain the diagrammatic metamodels as these conveniently summarise the formal language definition. (This will allow comparison of IS and general UML metamodels, and will help us to define the IS UML as a UML profile.) The formal (B) definition of the language starts with the derivation of a formal representation of the diagrammatic metamodels, using the guidelines for representing object-oriented models in B, defined in our earlier work[2]. B invariants are added to complete the syntax and to express the formal denotational semantics of these structural concepts. The B invariants re-express all the relevant OCL constraints used in existing pUML (and OMG) metamodels for UML, and add properties relating to IS constraints (see Appendix A).

For functional aspects (operations and transactions), we extend the graphical metamodel to express operation signatures. We use the B formal language to describe operation contents. The semantics of operations are given by the semantics of the formal language and the translation rules from IS UML to the formal language (figure 2).

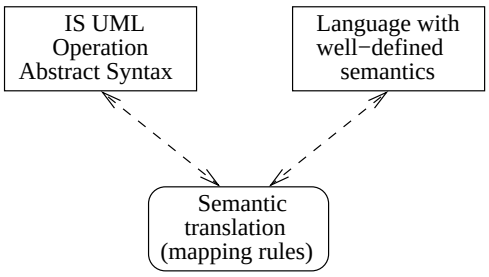

Figure 2: The Language Definition Method Used for IS UML Operations

2.2 Justification of the B Notations

Conventional work on UML semantics uses UML's diagrammatic notations and OCL constraints. Although OCL is intended for the expression of object constraints, it is not a mature language, and does not have a formally-defined semantics. A significant drawback of using OCL in language

definition is the need to derive a definition of the language of expression at the same time as using it to express the syntax and semantics of the modelling concepts. A practical problem in the use of OCL is the current lack of commercial tools for checking OCL expressions. OCL provides more rigour than a natural language, but less rigour than most mature formal languages.

In our research, we use the well-defined B method (based on Dijkstra's guarded command language, abstract machine notation, etc). B semantics are used to give meaning to IS UML concepts. B is a model-based formal language (other such languages include Z and VDM). It provides the power of expression required to specify the extensive structural aspects of IS. The functional specification of a system is built directly on the structural specification. (It would be difficult to express dynamic properties such as liveness but this is not of concern here.)

Languages such as B provide relational operators and types such as sets and sequences that facilitate precise, readable expression of data integrity constraints. However, B also provides a well-defined structuring mechanism, using an architecture of abstract machines, and routine tool support for a fully-proven development from specification to implementation. B tools generate, and assist in the discharge of, all proof obligations, including those relating to the enforcement of the state invariant. We have used the proof facilities in IS specification[5], and will use the B structuring to complete the support for operations and transactions in forthcoming work.

3 Language Structures for Operations in IS UML

This section presents language structures required to support the rigorous specification of IS functionality. (Appendix A gives diagrams summarising the definition of relevant IS UML structural concepts and extracts of the IS UML syntax and semantics expressed in B.) First, the nature of IS functional components is outlined. Then, diagrams are used to extend the abstract syntax of the structural aspects with details of operations and preconditions. Finally we discuss the approach used to define the semantics of the language. For brevity, only summary diagrams are given. The derivation of the formal metamodels is as for the structural aspect.

3.1 Characteristics of IS Functional Components

In IS, a *transaction* is a component of an application designed to support a user role. A transaction must enforce the global integrity of the system, but cannot directly change the state of the system.

A transaction is made up of *basic operations*. A basic operation is related to a single class or association. In an IS, the specification of many basic operations (here referred to as *generated operations*) can be deduced from the structural specification, independently of any transaction content: each class and association must have an operation to create and delete objects, and to replace the values of each changeable attribute. These are the only operations that can change the contents of a system class or association.

We introduce an intermediate level between transactions and generated operations. *User Operations* allow us to define a context of use for generated operations, and can be used by the designer to specify, for example, additional preconditions dependent on the application. A user operation is a basic operation and is a combination of basic operations on a single class or association that enforces the integrity constraints related to this class or association.

In practice, the use of basic operations allows consistency proofs of the system to be established in an incremental way: once the consistency proof of a basic operation has been established, it can be "reused" for the consistency proof of transactions built from basic operations.

3.2 Abstract Syntax of Operations

Figure 3 defines the abstract syntax of *Transaction* and *BasicOperation*. These are subclasses of *Operation*, which has attributes *opName* and *opParams*. A *BasicOperation* represents operations on one class or association, whereas a transaction, composed of basic operations, acts on many classes and/or associations. (Details of the class and association concepts are in Appendix A.)

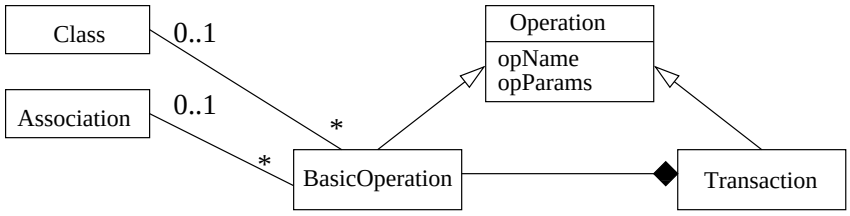

Figure 3: Metamodel Extract for the General IS Operation Abstract Syntax

The diagrammatic abstract syntax of operations allows us to define the signature of any operation. The signature is sufficient for a user to know how an operation is called. The type of *opParams* is defined in B as a sequence $seq(TPARAM)$. This is an indexed structure (mathematically, it is an injection from natural numbers to the sequence type), and is used here to model an ordered list of parameters. The type $TPARAM$ is a triple of a name, a type, and an

indicator of whether the parameter is input (i) or output (o).

$$TPARAM \in (NAME \times TYPE \times \{i, o\})$$

Generated and user operations are subclasses of *BasicOperation* (figure 4). *UserOperation* is also an aggregation of other user and generated operations.

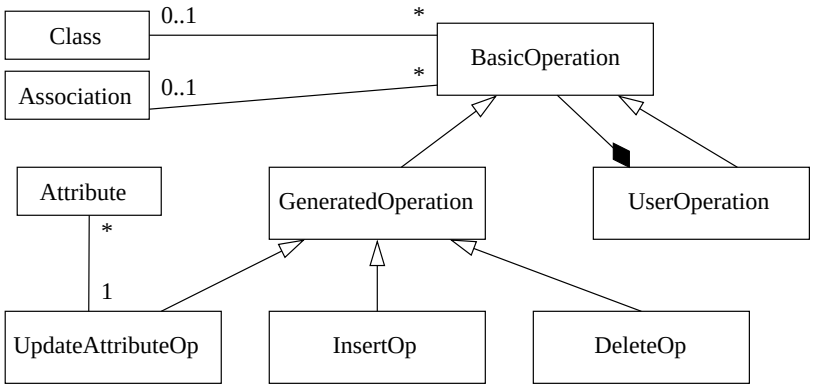

Figure 4: Metamodel Extract for the IS Basic Operation Abstract Syntax

3.3 Developing the Semantics of IS UML Operations in B

We need to define semantics for operations so that we can provide conceptual linkages between the structural and functional aspects. This will also allow us to develop a tool to assist in the design of user operations and transactions.

In [10], we have proposed a method that uses UML collaboration and state diagrams to express IS transactions, and then translates these into B specifications. The method relies on a particular style of state and collaboration diagrams, and is applicable to most IS transactions, but cannot be fully generalised. In this paper, we use an alternative, generally applicable approach, writing transaction specifications directly in B, in a form that matches the requirements of the user.

The IS UML functional semantics are expressed via the semantics of the B specification language used to describe the operation contents (preconditions and effects). At the meta-level, figure 5 uses the UML class notation to summarise how the relevant IS metamodel components are represented in B concepts. The *Operation* meta-class is represented by one *B_Signature*, one *B_Expression* and one *B_Substitution*. In the B notations, a *B_Expression* is an expression which evaluates to a boolean (true or false). It is used to express operation preconditions and also

invariant conjuncts. A *B_Substitution* describes the effects of an operation. It might change the state, calculate an output value or call another operation.

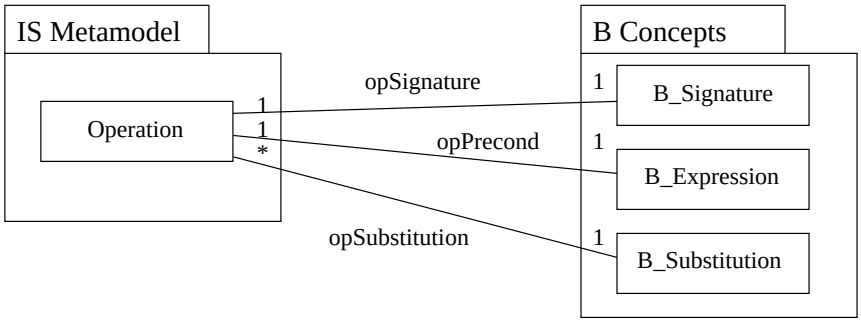

Figure 5: B Concepts Representing IS Metamodel Concepts Relevant to Operations

The semantics of a IS operation are expressed in these three B constructs. To be more precise, the complete semantics are given by the rules for deriving the full B specification from a full IS-UML specification. This includes expression of operations into B and also the B definition of the system structure and the links between the various B components. The rules for expressing operations in B, fully defined for the approach that defines B operations from collaboration diagrams and state diagrams[10], are summarised in Appendix B.

We also define a kind of "semantic property" of IS operations with respect to integrity enforcement. Recall that a consistent operation must establish the integrity constraints related to the state components on which it acts. The ability to prove this is a vital component of rigorous specification. In B, there is a formal proof obligation,

$$P \ \wedge INV \ \Rightarrow [S]INV$$

This states that, for an operation with a non-trivial precondition, P, if the precondition and the relevant invariant, INV, are true, then the substitution, S, corresponding to the operation, re-establishes the invariant[1, p236].

For an IS, the B invariant includes the translation of integrity constraints; the formal proof obligation thus implies an obligation on each operation and transaction to maintain integrity of the data, through its preconditions and its effects. Operation semantics are thus interrelated with structural concepts on which they act. This is illustrated in the following section.

4 Illustration of the Language through a Simple Example

The concepts in section 3 are illustrated for the UML class model in figure 6, and selected operations. The class model is translated into B variables and invariants, using the rules summarised in the Appendix B. We introduce the B given set *EMPNO* for the attribute *empNo* and *TELNO* for the optional attribute[1] *telNo*. Other attributes use the *NAME* and *STRING* given sets, and the B type *NAT*. The graphically-expressed inter-association constraint (*includes*) specifies that the head of department is an employee of that department.

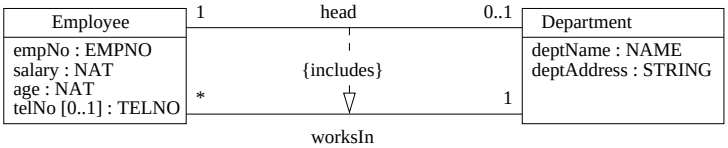

Figure 6: Example IS UML Class Model

4.1 Definition of Generated Operations

Generated operations are the only operations in the IS specification that are allowed to modify the system state. The designer is not permitted to modify these operations. B does not have a simple concept which could be used to define generic insert, update and delete operations, unlike Z[11]; we use systematic rules to construct these operations[2]. The contents of the generated operations are derived from the metamodel definition of the structure of and constraints on the system state. The automatic construction of generated operations in B defines their semantics.

Parameters of generated operations are deduced from the system structure recorded in the metamodel. For example, an operation to insert a new object in a class must have a parameter representing each mandatory attribute of the class. Typing preconditions for the parameters come from the system structure. The remaining preconditions come from intra-object constraints because such operations act on only a single object or link. The form of the substitutions derives from the B representation of the system structure.

We give two examples to demonstrate generation. These are insert operations for an object and a link.

[1] The default multiplicity of a UML attribute is [1..1]. An optional attribute has the explicit multiplicity [0..1].

Operation to Insert an Object

InsertEmployee is an operation to insert a new employee object. Its signature comprises an input parameter, *newEmp*, representing an object of the class, *Employee*, and input parameters, p_1, p_2 and p_3 which give values for the three mandatory attributes *empNo*, *salary* and *age*.

The rules for generating insert operations define the general precondition to establish that the new object is not already in the system. Here, this is,

$$newEmp \in EMPLOYEE - Employee$$

where *EMPLOYEE* is the type of the class, ie the set of all possible employee objects, and *Employee* is the set of objects that already exist in the system. The B set semantics used here are more powerful than the customary UML definition that does not distinguish between the type of a class and the set of objects that exist at any point.

The preconditions derived from typing invariants establish the type of each parameter:

$$p_1 \in EMPNO \wedge p_2 \in NAT \wedge p_3 \in NAT$$

The general form of substitution for a generated insert operation is a set union. This is used to insert *newEmp* in the set *Employee*, and to update each function that assigns mandatory attributes to the *Employee* set with a maplet from *newEmp* to the relevant parameter:

$$\begin{array}{l} Employee := Employee \ \cup \ \{newEmp\} \\ || \ empNo := empNo \ \cup \ \{newEmp \mapsto p_1\} \\ || \ salary := salary \ \cup \ \{newEmp \mapsto p_2\} \\ || \ age := age \ \cup \ \{newEmp \mapsto p_3\} \end{array}$$

The parallel substitution, ||, is used in B specification to indicate an undetermined order of execution. The completed operation establishes the relevant part of the invariant.

We assume that the specification of the class *Employee* includes two intra-object integrity constraints. The first restricts the age of employees, and the second restricts the salary of younger employees. These constraints are expressed in two conjoined invariant clauses, in the following *B_Expression*s :

$$\begin{array}{ll} (i) & \forall \, e \in Employee \bullet age(e) \ \geq \ 18 \wedge \\ (ii) & \forall \, e \in Employee \bullet age(e) < 30 \Rightarrow salary(e) < 3000 \ euros \end{array}$$

To enforce these constraints, the precondition must be completed by the two following expressions :

$p_3 \geq 18 \wedge$
$(p_3 < 30 \Rightarrow p_2 < 3000)$

Operation to Insert a Link

The *InsertWorksIn* operation creates a link between employee and department objects. It requires two parameters (*emp* and *dep*) to represent the existing objects to be linked. The inserted link is an instance of the association, *worksIn*, represented in B as a total function from *Employee* to *Department*.

The generated precondition depends on the multiplicity of the association. The multiplicity for *Employee* is 1; the link is created only when the employee is created. Thus the precondition includes :

$$emp \in EMPLOYEE - Employee$$

The multiplicity for *Department* is *; that means that the link is created either when the department is created or for an existing department. Thus the precondition includes :

$$dep \in DEPARTMENT - Department \vee dep \in Department$$

which can be simplified as :

$$dep \in DEPARTMENT$$

There are no other preconditions, as intra-object constraints do not exist for links. The general form of the substitution to add a link to the system gives,

$$worksIn := worksIn \cup \{emp \mapsto dep\}$$

Note that a formal check that the B representation of *worksIn* is still a total function cannot be achieved by proof of this operation alone, as we do not have enough information on the existence of the department and employee. It is achieved in a transaction that inserts an employee or a department, below.

4.2 Definition of Transactions and User Operations

IS transactions are identified from user requirements. The details cannot be automatically generated, but must be added by the designer. The only difference between user operations and

transactions is that user operations relate to only one class or association, whereas transactions reference all the elements needed to express the user-required functionality.

The parameters of transactions and user operations include attribute parameters of the basic operations called by their substitutions. Other parameters may be added, for example, to represent the output of an operation. The substitutions include calls to basic operations and output calculations; the substitutions are not allowed to directly modify the state of the system.

Preconditions include all the preconditions of the basic operations called by their substitutions, plus designer-written preconditions reflecting both the integrity constraints on the parts of the state affected by the transaction or user operation, and environmental constraints that come from business rules. Specifications of transactions must enforce inter-association and general constraints, whereas specifications of user operations must enforce inter-object constraints.

In the example system, we require a transaction to insert an employee object, link the employee to an existing department, and assign a telephone number. A possible form of this transaction in B is:

$newEmp \leftarrow AddNewEmployee(no_t, sal_t, age_t, dep_t, tel_t)$
PRE
$no_t \in EMPNO \wedge sal_t \in NAT \wedge age_t \in NAT \wedge tel_t \in TELNO \wedge$
$age_t \geq 18 \wedge (age_t < 30 \Rightarrow sal_t < 3000) \wedge$
$dep_t \in Department \wedge$
$Employee \subset EMPLOYEE$
THEN
ANY e **WHERE** $e \in EMPLOYEE - Employee$
THEN
$InsertEmployee(e, no_t, sal_t, age_t)$
$\| InsertWorksIn(e, dep_t)$
$\| UpdateTelNo(e, tel_t)$
$\| newEmp := e$
END
END

Here, upper-case bold terms are B keywords. The signature of the operation defines the output parameter, *newEmp*, and five input parameters, deduced from the parameters of the called operations. The **ANY** substitution selects, non-deterministically, an object that meets its predicate ($e \in EMPLOYEE - Employee$). *UpdateTelNo* is a generated operation that updates the optional attribute. The last substitution calculates the output of the transaction. The precondition, $Employee \subset EMPLOYEE$, ensures that a new employee can be created, since $EMPLOYEE$ is the set of all possible employee objects and $Employee$ is the set of existing

employee objects. The totality *worksIn* is enforced by the precondition $dep_t \in Department$ and the call to both the generated operations, *InsertEmployee* and *InsertWorksIn*.

5 Approaches to a Tool Specification Assistant

Ultimately, we would like a rigorous IS specification tool that assists the designer in the construction of consistent operations such as those illustrated in section 4. Our semantics can help in calculating preconditions that are capable of enforcing the relevant clauses of the state invariant and thus the integrity constraints.

There are two possible approaches to assisted construction of preconditions. The first, a reactive (*a posteriori*) approach, presents an initial B specification of the system state and transactions to a proof tool such as AtelierB[12], which attempts to discharge the proof obligation. Assuming that the state invariant is sufficiently defined, the proof fails if either the precondition or the substitution of an operation is insufficient. A specification assistant could analyse the error messages from the proof tool to deduce the missing preconditions and/or substitutions. However, the messages are not generally sufficiently explicit for this to be fully automatable.

The second approach, which we favour, is a pro-active (*a priori*) tool that helps the designer to construct consistent operations before attempting to discharge the proof obligation. Such a tool would firstly identify the parts of the state invariant that are relevant to the operation, and secondly automatically generate preconditions and substitutions whenever possible. For this approach, we need to consider in more detail the sources of the invariant and the semantic relationship between the invariant and operation preconditions.

We can identify two sources of invariants:

- typing invariants for classes, attributes and associations, derived from the core IS metamodel (Appendix A, figure 8);
- invariants which express integrity constraints on the system specification, derived from the constraint metamodel (Appendix A, figure 9).

A tool should be able to retrieve all typing invariants and integrity constraints which pertain to the parts of a system structure affected by an operation. To do this, the tool would need access to the complete metamodel that records this information.

In theory, a precondition can be automatically derived from most invariants. This is practical only for typing invariants. For invariants that express integrity constraints, we assume that the designer can express (as a *B_Expression*) each precondition required to enforce the constraint.

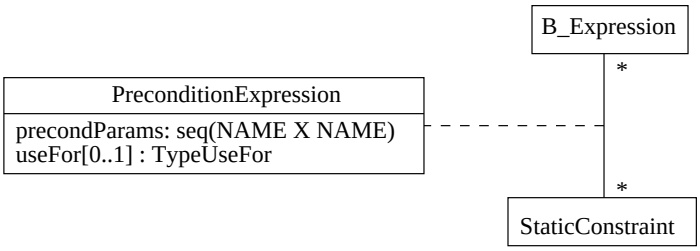

Figure 7: A Structure to Represent Expression of Preconditions

Figure 7 describes a structure to record all generated and designer-written preconditions. For each precondition of a particular constraint, the association class *PreconditionExpression* records the implied parameters. Each parameter is described by its name (of type *NAME*) and the element of the structural model to which it is associated (also of type *NAME*).

For some constraints, it is also possible to identify the types of operations for which each precondition is relevant. This is recorded by the optional *useFor* attribute, the type of which, *TypeUseFor*, is defined as:

$$\mathbb{P}\ \{Insert, Update, Delete\}$$

For instance, the instantiation of figure 7 for constraint (i) is:

$$B_Expression: \quad x \geq 18$$
$$precondParams: \quad \langle\ (x, age)\ \rangle$$
$$useFor: \quad \{Insert,\ Update\}$$

In generating the *InsertEmployee* operation, the tool would use the metamodel to seek *Insert* preconditions for the *Employee* class abd its attributes. The associated *B_Expression* would be added to the precondition, and the tool would equate x to the input parameter representing the attribute *age*, ie p_3.

Such simple solutions do not always exist. For transactions, the precondition required to enforce a specific constraint varies according to content of the transaction. For instance, suppose that there is a general constraint stating that the salary of an employee must not be greater than the salary of the head of the employee's department:

$$(iii) \quad \forall\, e \in Employee \bullet salary(e) \;\leq\; salary(head(worksIn(e)))$$

In the transaction *AddNewEmployee*, the designer must add the corresponding precondition:

$$sal_t \;\leq\; salary(head(dep_t))$$

However, for a transaction that inserts a new employee as the head of department, a different precondition is needed, to establish that the salary of the new head is at least as large as that of any other employee who works in the department:

$$sal_t \;\geq\; max(salary[worksIn^{-1}[\{dep_t\}]])$$

We consider that the effort required to produce a more powerful tool would not be justified. Such a tool would require an in-depth analysis of transactions and precondition expressions, and classification of every transactions according to the types of called generated operations. For each constraint, each of its precondition expressions would be matched to the relevant generated operation of a transaction. This would require an extension of the type of the *useFor* attribute. The linking of precondition expressions to generated operations would also have to impose a hierarchical ordering of preconditions since some of the preconditions may annul others: in the example above, a transaction to add a new head of department would both insert an employee object and insert the head link; however, it must only have the precondition, relating to head's salary, as that relating to the employee's salary would prevent the new head from having a higher salary than the old head.

Remark that not all constraints relevant to transactions can be directly enforced in preconditions. Some must be enforced by combinations of called operations. For example, the class model, figure 6, shows the *includes* inter-association constraint, of which the *B_expression* is:

$$head \subseteq worksIn$$

To enforce this invariant, a transaction that inserts a new head of department must include all the generated operations needed to ensure that the new head is an employee of the same department.

Reciprocally, there are, of course, preconditions that are not linked to a constraint but that come from business rules. It is the case of the precondition $dep_t \in Department$ in the transaction *AddNewEmployee* that states that the department must exist before inserting a new employee.

In summary, a tool to assist design of IS operations can provide automatic generation of simple

operations, and can assist in the specification of a transaction by listing all the constraints relating to the parts of the system structure on which it acts.

6 Related Work

There is a range of other work using formal approaches to specify systems. We are interested principally in approaches that relate to IS and database development.

1. Work deriving formal specifications from diagrams without defining a rigorous language includes the SAZ method, which derives a Z specification from structured state and behavioural models[13, 14], and Dupuy's derivation of a Z specification from OMT or UML models[15]. These are practical and adaptable approaches, but do not provide the rigour required for our tool, and do not allow direct reasoning about the diagrammatic specification.

2. As well as our work on object-oriented notations and B, there are a number of other attempts to define language metamodels and to use these to translate between diagrammatic and formal notations. Early approaches are summarised in [16]. Although this approach is well-suited to the construction of rigorous development tools, most work leads to tools for checking only diagrammatic models produced in various notations. For example, Bryant[17] reports a tool for checking Yourdon structured models using translation into Z.

3. Strictly formal but less practical IS specification research does not use a graphical notation. In [18], the derivation of relational database programs is presented. From TD, a conceptual modelling language that is already formal, B is used as an intermediate language to carry out a formal refinement. A similar approach is applied to object-oriented databases in [19]. [20] uses the formal language, TM, to design transactions and the prover, Isabelle/HOL, to discharge proofs. In these approaches, the use of two distinct formal languages complicates the work of the designer and hinders provision of tool support.

4. Work aiming to identify the preconditions for basic operations in IS includes that of Ledru using the Z/EVES theorem prover to detect missing preconditions[21]. This is similar to the reactive approach noted above. Hadj-Rabia and Habrias [22] describe a similar approach using NIAM, but do not make use of a tool. These approaches do not consider

more complex IS transactions, and do not provide any automatic generation facilities to assist the designer.

7 Conclusion

This paper explores the functional aspects of a rigorous language dedicated to IS specification and based on UML graphical notations, showing how different sorts of operations can be expressed and linked to structural elements. We have identified different ways of providing assistance to the designer of such operations, explaining which parts of the formal operation specification can be automatically generated, and which structural elements must be considered in order that operations enforce the integrity of an IS.

To define the rigorous language, we add to the abstract syntax that we have already defined for the structural aspects of IS UML structures representing different kinds of operations that exist in an IS and a metamodel of operation signatures. We explain why we cannot use denotational semantics, used to specify structural aspects, for specifying the semantics of operations. These are given by the semantics of the B language and translation rules from IS UML to B notations. We illustrate the formation of operations in the formal B notation, using the semantics of the B notation and the rules of translation to define the operation semantics.

A formal specification forms a contract between the developer and the designer. It allows us to reason about the IS specification at this level, rather than waiting to reason about the implementation (it is least costly to correct problems early). It also allows us to verify consistency between the structural and functional aspects of a model. With formal language tools, we can animate or prototype the design, generate test cases[23], and formally refine the IS specification, eg to derive relational database definitions[24].

A tool is under construction. This already supports the translation into B of a UML structural specification modelled using the Rose toolkit[25]. We use the OCaml[26] functional language, an extension of ML, to express the metamodel and translation rules[27]. The tool can automatically generate simple operations. The work presented here will allow the tool to be extended to assist in the design of consistency-preserving IS transactions.

In the future, we will complete a structural definition of IS inheritance and aggregation, and extend the metamodel to include the abstract syntax of other diagrammatic UML notations used for behavioural aspects of IS specification (state diagrams and collaboration diagrams). B is not

ideal for the dynamic aspects, but there is ongoing research into event-oriented extensions of B[28, 29]. We also intend to investigate the connections between B refinement and the design process in UML. These developments should allow us to define a complete IS profile for UML.

APPENDIX A: The IS Core Structural Concepts

This appendix illustrates the existing metamodels for the structural concepts of the IS UML[30]. We present only structural details relevant to our operation definition. We do not consider inheritance in this paper. Aggregation and composition concepts are not yet well-defined in UML.

A1: The IS UML Metamodel

In the abstract syntax of the structural IS UML concepts, figure 8, attributes and association ends are modelled as subclasses of the class, *Characteristic*. Attributes in IS UML require explicit multiplicity (*characMult*) and changeability(*characChange*), and are thus identical in form to association ends, except that *Attribute* also has a type, represented as the class, *Type*. A mandatory attribute or association end must not have a multiplicity containing zero. Classes are modelled as having many characteristics, but each characteristic is directly linked to only one class. Note that an association class is represented by a link of the association *assocClass*, plus objects of the classes *Class* and *Association*. The concepts modelled are essentially those of the UML core, omitting details which are not relevant to IS specification (visibility, concurrency etc).

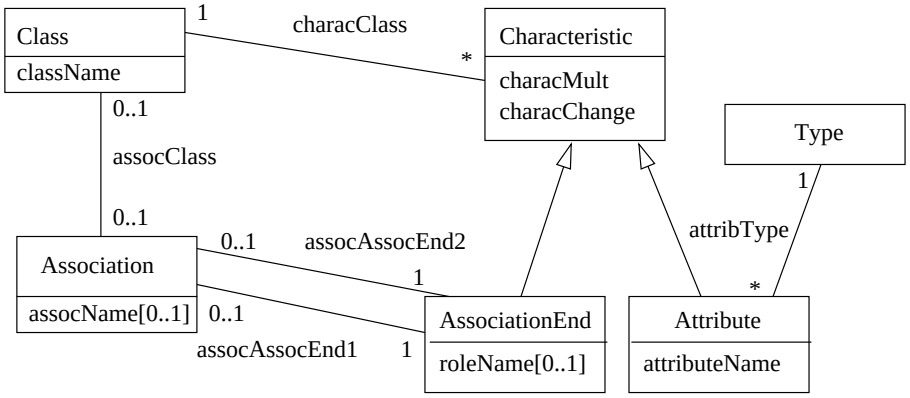

Figure 8: IS UML Metamodel for Structural Concepts

In [6], we define a hierarchical metamodel for IS structural constraint concepts. Figure 9 gives

the abstract syntax of the constraint superclass, *StaticConstraint*, and illustrates the associations between constraint subclasses and other structural concepts.

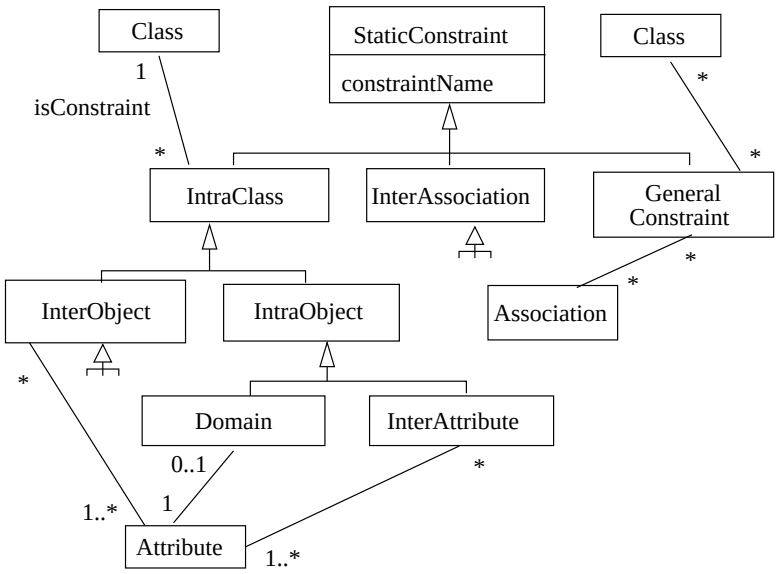

Figure 9: IS UML Metamodel Extract Summarising Static Constraint Concepts

The *GeneralConstraint* subclass represents constraints determined by the system requirements, such as business rules relating to the system classes and associations. The *IntraClass* subclasses model constraints on and among objects of a specific class. The leaf classes of this hierarchy have various associations with the *Attribute* class. Not shown here are the inter-object constraints required to model the IS functional dependencies and the constraints relating to IS keys, and inter-association constraints on the form and participants of associations and association ends.

A2: B Representation of Abstract Syntax, Semantic Domain and Semantic Mappings for IS Class Model Concepts

The denotational approach to the structural definition of the IS UML is illustrated with parts of the B model of abstract syntax, B variables and invariants which elaborate the abstract syntax, and B variables and invariants stating the semantic domain and semantic mapping.

Class Type Structures in B

Each metamodel superclass is of a different type; a variable defines the set of known instances. An invariant (table 1) defines the variable to be a subset of the class type.

Class	$\subseteq$	*CLASS*
Association	$\subseteq$	*ASSOCIATION*
Characteristic	$\subseteq$	*CHARACTERISTIC*
Type	$\subseteq$	*TYPE*

Table 1: B Variables and Invariants for Sets of Known Instances of Metamodel Classes

Data types (written in upper case) can be equated to mathematical sets. The B Method incorporates integers (and integer-related proof rules), and has a variety of mechanisms for introducing new sets. For the IS UML definition, most types are defined as *given sets* about which most details remain unelaborated[2]. We use the given set, *TYPE*, to represent any relevant UML- and user-defined data types.

Subclasses have the same type as the superclass. Additionally, the subclass instances are represented by a variable which is a subset of the superclass variable. The following invariant expresses both these properties for the class *Characteristic*:

$$Attribute \subseteq Characteristic \wedge AssociationEnd \subseteq Characteristic$$

The disjointness of the subclasses of *Characteristic* is defined by the invariant,

$$Attribute \cup AssociationEnd = Characteristic \wedge Attribute \cap AssociationEnd = \varnothing$$

Structural Definitions Using B Relations and Functions

Any kind of association can be modelled as a binary relation[31] (table 2). A binary relation is a set of pairs, comprising an element of the source set and an element of the target set[1, p77-9]. Participation of the domain (source) and range (target) elements can be constrained by mathematical functions[1, p86].

Functions are used to model the associations in the metamodel (table 3). They are also used to associate attributes to classes (table 4). Note that only the metamodel class *Class* must have a unique name; the names of other concepts are rendered unique by their association to *Class*. This is defined in B constraints (not shown).

Most attribute types are UML types[32, section 2.7]. *NAME* is an implicit subset of the OMG's catch-all *STRING* type. The range of *characMult* is the B finite powerset of natural numbers ($\mathbb{P}$

[2] A B given set may have enumerated values, or may simply be an (implicitly) non-empty, finite set[1, sections 2.2 and 4.17].

$\leftrightarrow$	*relation*	$*$ *to* $*$ *mapping*
$\rightarrow$	*function*	$*$ *to* 1 *mapping, all domain takes part*
$\rightarrowtail$	*injection*	1 *to* 1 *mapping, all domain takes part*
$\nrightarrow$	*partial function*	$*$ *to* 1 *mapping, not all domain takes part*
$\nrightarrow\!\!\!\!\rightarrowtail$	*partial injection*	1 *to* 1 *mapping, not all domain takes part*
$\twoheadrightarrow\!\!\!\!\!\rightarrow$	*partial surjection*	$*$ *to* 1 *mapping, not all domain, all range takes part*

Table 2: Example Mappings Modelled by Relation and Functions

$assocClass$	$\in$	$Association \rightarrowtail Class$
$assocAssocEnd1$	$\in$	$Association \rightarrowtail AssociationEnd$
$assocAssocEnd2$	$\in$	$Association \rightarrowtail AssociationEnd$
$characClass$	$\in$	$Characteristic \rightarrow Class$
$attribType$	$\in$	$Attribute \rightarrow Type$

Table 3: B Variables and their Typing Invariants Representing Associations in the IS Metamodel

NAT). This can express any multiplicity, with B's upper limit ($maxint$) equating to UML's $*$.

Semantic Domain and Semantic Mapping of the IS UML Language

The semantic domain introduces the concepts of object, value, and link. In B, we define a set of known objects, whose type is the given set, $OBJECT$:

$$Object \subseteq OBJECT$$

The given set, VAL, represents all possible values in the model. An object maps to many values:

$$objVal \in Object \leftrightarrow VAL$$

A link is represented as a pair of *object*s:

$$link \in Object \leftrightarrow Object$$

The denotational semantics of structural concepts is established by the semantic mapping between the semantic domain and the abstract syntax[6, 30], summarised here.

A class maps to a set of objects, such that each object is one instance of exactly one class:

$$objectClass \in Object \rightarrow Class$$

CLASS	*ATTRIBUTE INVARIANT*
Class	$className \in Class \rightarrowtail NAME$
Characteristic	$characMult \in Characteristic \rightarrow \mathbb{P}\ NAT$ $characChange \in Characteristic \rightarrow BOOLEAN$
Attribute	$attributeName \in Attribute \rightarrow NAME$
AssociationEnd	$roleName \in AssociationEnd \nrightarrow NAME$
Association	$assocName \in Association \nrightarrow NAME$

Table 4: B Variables and Invariants Associating Attributes to IS Metamodel Classes

The definition of *link* is used to instantiate an association as a set of links among objects. There may be more than one association among the same objects:

$$linkAssociation \in Association \rightarrow link$$

The semantic mapping between attributes and the object evaluations is defined by,

$$attributeObjectValue \in Attribute \rightarrow objVal$$

APPENDIX B: Summary of UML to B Translation Rules

Figure 10 summarises how each IS UML structural concept is represented in B. Each class in a specification is represented as a specific *B_GivenSet* that models the set of all possible instances. *B_Variable*s represent the sets of existing instances (objects, links) of each class and association. Each attribute is also represented as a *B_Variable* and has a particular *B_Type*. *B_Type* is the superclass of all typing concepts in B. A given set that is used to represent a system class cannot also represent an attribute.

Each *B_Variable* is associated with an B invariant that defines its type. Additional constraints define links between the invariant, the kind of IS UML concept and the associated variable, as illustrated for metamodel attributes and associations in tables 4 and 3, above. Finally, each static constraint is represented as an invariant.

A B specification is composed of a set of abstract machines. One abstract machine is generated for each class. If an association contains specific attributes and/or operations, an abstract machine is generated, otherwise the association is included in the machine of one of the classes involved in the association. Each of these machines contains the B variables and invariant related to the class or association that it corresponds to and the relevant generated operations. These

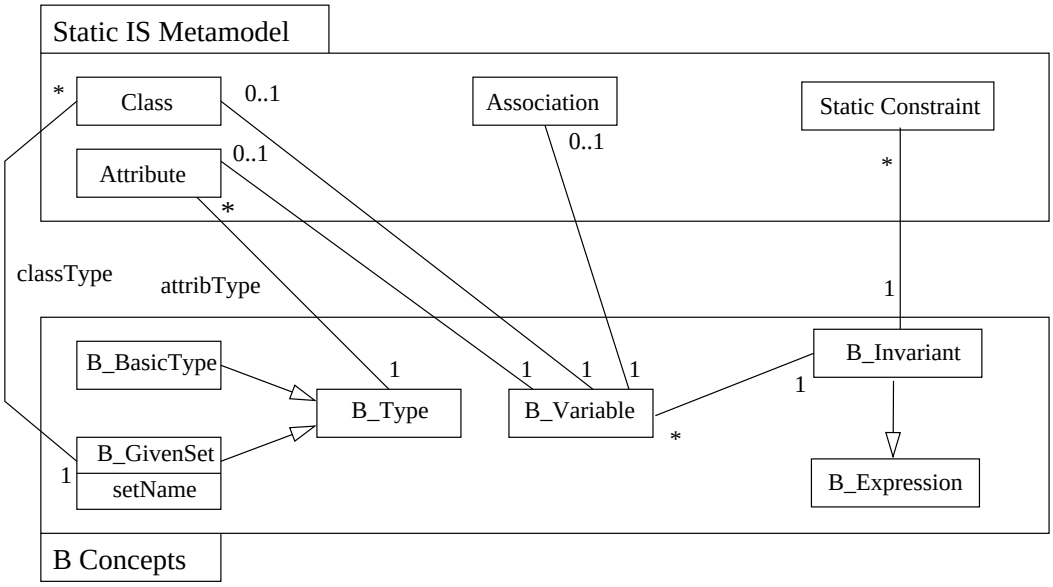

Figure 10: The Core IS UML Structural Concepts and Associated B Concepts

machines are called *Basic Class/Association Machines*.

The set of user operations associated with a class or association is included in an abstract machine called *User Class/Association Machine* which has no variable but includes the basic machines that provide access to variables and generated operations. Transactions are defined in an abstract machine that includes all the necessary user class/association machines and basic class/association machines. Each transaction is translated by a B operation. The machine has no variables but can contain invariants corresponding to general constraints or inter-association constraints. This top-level machine forms the interface of the global B specification.

References

[1] J-R. Abrial. *The B-Book*. CUP, 1996.

[2] H. P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi–formelles*. PhD thesis, CEDRIC Laboratory, CNAM, Evry, December 1998. available from www.iie.cnam.fr/˜ laleau/thesePHNguyen.ps-gz.

[3] F. Monge. Formalisation du méta modèle des méthodes graphiques d'analyse et conception orientées objet. Master's thesis, DEA IIE(CNAM)-University of Evry-INT, Evry, September

1997.

[4] P. Facon, R. Laleau, and H. P. Nguyen. Dérivation de spécifications formelles B à partir de spécifications semi–formelles de systèmes d'information. In *First B Conference, Nantes, France*. IRIN, France, 1996.

[5] P. Facon, R. Laleau, and H. P. Nguyen. From OMT diagrams to B specifications. In M. Frappier and H. Habrias, editors, *Software Specification Methods. An Overview Using a Case Study*, pages 57–77. Springer, 2000.

[6] F. Polack and R. Laleau. A rigorous metamodel for UML static conceptual modelling of information systems. In *CAiSE2001: 13th International Conference on Advanced Information Systems Engineering, Interlaken, Switzerland*, volume 2068 of *LNCS*. Springer Verlag, June 2001.

[7] A. Clark, A. Evans, S. Kent, and P. Sammut. The MMF approach to engineering object-oriented design languages. In *LTDA2001: Workshop on Language Descriptions, Tools and Applications*, April 2001.

[8] T. Clark and A. Evans. Foundations of the Unified Modeling Language. In *Second Northern Formal Methods Workshop*, volume 1241 of *LNCS*. Springer Verlag, 1997.

[9] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing. MIT Press, Cambridge Mass., 1993.

[10] P. Facon, R. Laleau, H. P. Nguyen, and A. Mammar. Combining UML with the B formal method for the specification of database applications. Technical report, CEDRIC Laboratory, CNAM, Paris, September 1999.

[11] F. Polack and S. Stepney. System development using Z generics. In *FM99: World Congress on Formal Methods, Toulouse, France*, volume 1708 and 1709 of *LNCS*, pages 1048–1067. Springer Verlag, September 1999.

[12] DIGILOG. *Atelier B, Manuel de Référence*. DIGILOG groupe STERIA, BP 16000, 13791 Aix-en-Provence Cedex 3, France, 1996.

[13] K. C. Mander and F. A. C. Polack. Rigorous specification using structured systems analysis and Z. *Information and Software Technology*, 37(5):285–291, 1995.

[14] F. Polack, M. Whiston, and K. Mander. The SAZ project: Integrating SSADM and Z. In *FME93: First International Symposium of Formal Methods Europe, Odense, Denmark*, volume 670 of *LNCS*, pages 541–557. Springer Verlag, April 1993.

[15] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. Overview of Roz: A tool for integrating UML and Z specifications. In *CAiSE00: 12th International Conference on Advanced Information Systems Engineering, Stockholm, Sweden*, volume 1789 of *LNCS*. Springer Verlag, June 2000.

[16] L. T. Semmens, R. B. France, and T. W. G. Docker. Integrated structured analysis and formal specification techniques. *The Computer Journal*, 35(6), 1992.

[17] T. Bryant, A. Evans, L. T. Semmens, R. Milovanovic, S. Stockman, M. Norris, and C. Selley. Using Z to rigorously review a specification of a network management system. In *ZUM95: 9th International Conference of Z Users, Limerick, Ireland*, volume 967 of *LNCS*, pages 423–433. Springer Verlag, September 1995.

[18] T. Günther, K. D. Schewe, and I. Wetzel. On the derivation of executable database programs from formal specifications. In *FME93: First International Symposium of Formal Methods Europe, Odense, Denmark*, volume 670 of *LNCS*. Springer Verlag, April 1993.

[19] B. Matthews and E. Locuratolo. Formal development of databases in ASSO and B. In *FM99: World Congress on Formal Methods, Toulouse, France*, volume 1709 of *LNCS*, pages 388–410. Springer Verlag, September 1999.

[20] D. Spelt and H. Balsters. Automatic verification of transactions on an object-oriented database. In *DBPL-6: 6th Workshop on Database Programming Languages, Colorado, USA*, volume 1369 of *LNCS*, August 1997.

[21] Y. Ledru. Identifying preconditions with the Z/EVES theorem prover. In *ASE: 13th IEEE Conference on Automated Software Engineering*. IEEE Computer Society Press, October 1998.

[22] N. Hadj-Rabia and H. Habrias. Formal specification from NIAM models: A bottom-up approach. In *ISCIS96: 11th International ISCIS Conference, Antalya, Turkey*, November 1996.

[23] H. Treharne, J. Draper, and S. Schneider. Test case preparation using a prototype. In *B98: Second International B Conference, Montpellier, France*, volume 1393 of *LNCS*, pages 293–312. Springer Verlag, April 1998.

[24] R. Laleau and A. Mammar. A generic process to refine a B specification into a relational database implementation. In *ZB2000: Formal Specification and Development in Z and B*, volume 1878 of *LNCS*, pages 22–41. Springer Verlag, August–September 2000.

[25] Rational Software Corporation. *Rational Rose - Using Rational Rose 98*. Rational Inc., 1998.

[26] X. Leroy. *The Objective Caml System, Documentation and User's Manual*. INRIA, France, 1999.

[27] R. Laleau and A. Mammar. An overview of a method and its support tool for generating B specifications from UML notations. In *ASE: 15th IEEE Conference on Automated Software Engineering, Grenoble, France*. IEEE Computer Society Press, September 2000.

[28] J-R. Abrial. Event driven sequential program construction. In *AFADL01: Approches Formelles dans l'Assistance au Développement des Logiciels, Nancy, France*, June 2001.

[29] J-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In *B98: Second International B Conference, Montpellier, France*, volume 1393 of *LNCS*, pages 83–128. Springer Verlag, April 1998.

[30] P. Facon, R. Laleau, A. Mammar, and F. Polack. Formal specification of the UML metamodel for building rigorous CAISE tools. Technical report, CEDRIC Laboratory, CNAM, Paris, September 1999.

[31] P. Facon, R. Laleau, and H. P. Nguyen. Mapping object diagrams into B specifications. In *Methods Integration Workshop, Leeds, UK*, eWiCS. Springer Verlag, March 1996.

[32] Object Management Group. *OMG Unified Modeling Language Specification(draft)*, April 1999. BetaR1 Release, available on line, www.rational.com/uml.