Coming and going from UML to B : a proposal to support traceability in rigorous IS development

Régine Laleau[1] and Fiona Polack[2]

[1] CEDRIC-IIE(CNAM),
18 allée Jean Rostand, 91025 Evry, France
laleau@iie.cnam.fr
[2] Department of Computer Science
University of York, York, YO10 5DD, UK
fiona@cs.york.ac.uk

Abstract. CEDRIC-IIE is researching rigorous information system (IS) development. Previous work includes translation rules for deriving a B specification from object-oriented diagrams, metamodels of IS UML structural and functional concepts, and a prototype translation tool. Here we outline the traceability needs for a tool to assist in rigorous IS development, and provide meta-structures for the required links among B and IS UML concepts, in the context of existing translation rules and IS UML metamodels.

1 Introduction

There is currently an overdue realisation that computer systems development in general must conform to good engineering practice: it is not just critical systems that must be optimally robust and reliable, and all computer systems should be developed within cost and time constraints. Computer systems development must become a verifiable industrial process not an art form.

Increasingly, information systems (IS) are at the centre of commercially-critical systems, and their development needs more rigour than has typically been applied. Light-weight or back-door formalism is a possible solution. In our work, we seek to extend diagrammatic tool support to provide rigour via a partly-invisible and largely machine-generated formalisation. The work derives from the extensive but under-exploited integrated methods work of the 1990s[26, 13], but exploits the increased use of and reliance on tool support for modelling, the increasing sophistication of formal methods support, and the increasing capacity of development support computers to manage the development process. The CEDRIC-IIE work has already published a guideline-based approach for deriving a formal specification of an IS (in B) from object-oriented class and behavioural models[22, 21, 10, 11]. 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, are being

rigorously defined for an IS-focused variant of UML. Ultimately, this will allow a support tool to comment not only on the syntactic correctness of each graphical model, but also on the semantic validity of the models, including cross-references between models. (Other research in this area is summarised in [17].)

In this paper, we explore issues of traceability between the UML and B specifications. We believe that tools for rigorous development must allow the designer to move between the UML and formal models in both directions. This helps to improve readability and allows evolution of the specification, for example as validation (including proofs of properties of the formal model) identifies errors in UML and/or formal models. To our knowledge, the existing tools deriving B specifications from UML specifications [20] do not support mechanisms that allow navigation in both direction.

2 Context

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[1]. For the IS UML, we follow the general approach of the pUML group (eg[4, 5]), defining a small core language and extensions. 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. The use of diagrams to specify system concepts and OCL for invariants makes the specifications hard to check or prove.

In our research, the full IS UML metamodel is written in B; UML class diagrams summarise the abstract syntax. B has a well-defined semantics that is used to give meaning to IS UML concepts. B provides the power of expression required to specify the extensive structural aspects of IS, and the functional specification 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.) B also provides relational operators and types such as sets and sequences that facilitate precise, readable expression of data integrity constraints, and a well-defined structuring mechanism, using an architecture of abstract machines. We have used the B proof facilities in IS specification[11].

[1] see OMG's Requests for Proposals (RFPs), http://www.celigent.com/omg/adptf/wgs/uml2wg.htm

We have already defined most of the structural aspects of UML required for IS specification[9, 24], including a structure for IS constraints, which supports the construction of operations with complete, consistent preconditions. 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 structural metamodels are summarised in section 5.1.

More recently, we have established an abstract syntax and semantics for the IS functional aspect, comprising transactions that correspond to user requirements and enforce the integrity constraints in IS operations[17]. This is the principle component of the IS behavioural aspect. (In IS, dynamic features such as liveness and concurrency management are largely determined by the software used to implement the IS eg, database management system.) The operational aspect is summarised in sections 5.2 and 5.3.

The main contribution of this paper (section 6) is to model traceability links between IS UML and B concepts summarised in section 5. To set the context, we first describe the architecture of a prototype tool, indicating the traceability needs (section 3). The process of deriving a B specification from a UML class model is illustrated, and issues in the development of a tool assistant are noted (section 4).

3 A Tool to Support Rigorous IS UML Development

Conventional diagrammatic case tools can check specifications for syntactic precision, based on a built-in metamodel of syntax. However, it is hard to check diagram semantics, and to check that the components specified in one diagrammatic model are consistent with related components in other models. For example, a class model specifies the operations on each class, whereas the state model defines the events which cause a class to trigger an operation on a particular object. The preconditions of the operation bear a relationship to the guard conditions on the transition triggered by the event. We aim to provide these checks via an underlying formal metamodel and a formalisation of the specification which can be analysed by existing formal method tools.

Figures 1 and 2 outline the architecture of a tool to assist in rigorous IS specification, using a B specification derived from IS UML models. In the diagrams, solid arrows represent the current state of a prototype tool produced at CEDRIC-IIE [16, 9]. The dashed lines show the needed traceability links which are the main contribution of this paper. Traceability between UML and B models (and vice versa) is crucial for rigorous tool support. Without bidirectional traceability, errors can be identified using the B method, but cannot be automatically associated with omissions or errors in specific features of the UML models.

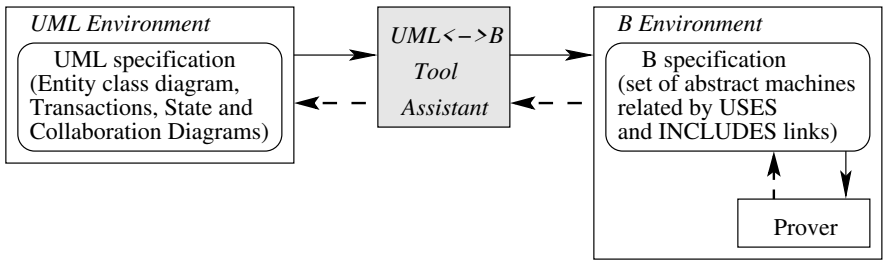

Fig. 1. Architecture of the Current Translation Tool

Figure 1 shows the context of the tool assistant. CEDRIC-IIE's prototype tool takes machine-readable UML models from the Rose case tool[6], and passes B machine structures to AtelierB[7], which performs formal analysis.

Figure 2 expands the architecture of the tool assistant. The machine-readable UML representation is verified against the metamodel definition of UML abstract syntax; it is then translated into B machines by the correspondence rules. The CEDRIC-IIE prototype performs these translations and verification using an OCaml implementation of the metamodel and rules. Figure 2 shows, as dashed arrows and boxes, the traceability mechanisms that are needed for the complete analysis of errors detected in translation, verification and AtelierB formal analysis. Thus, for example, if AtelierB finds an unprovable assertion, the traceability links are used to determine which parts of the UML model give rise to inadequate constraints or operations preconditions.

4 Deriving a B Model from a UML Class Model

This section illustrates the process of rigorous IS specification. Note that interaction between the system and the users would be modelled as operation calls (events), the signature of which are specified. The realisation of the user interface would be considered in design and implementation phases. The process is that automated in the CEDRIC-IIE prototype. The structure of the B specification that we produce mirrors the UML specification, to assist traceability. The process has five steps.

1. The IS UML entity class diagram is constructed. The diagram expresses the structure of data that is persistent over time. In IS, the class model is crucial, as it provides the basis of the final implementation structure.
2. The B specification corresponding to the diagram is derived. Primitive operations can be automatically generated once the B specification of the state has been defined. These low-level operations, generic to any IS, comprise update, insert, and delete operations for each class and association, changing the state of the object(s) and link(s). The operations are added to the IS

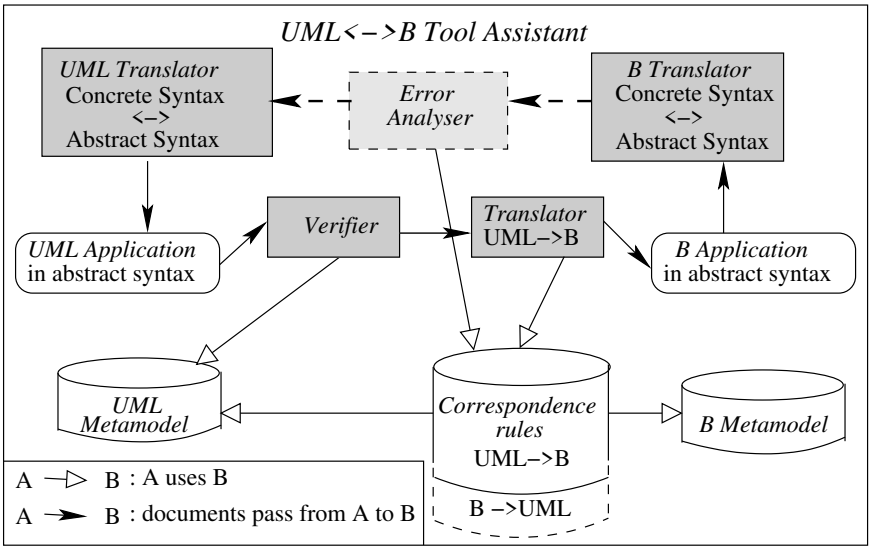

Fig. 2. Architecture of the Current Translation Tool

UML entity class diagram. The B specification is presented in an architecture of basic B machines that contain all the state variables of the system with the relevant primitive operations.

3. The user requirements and system business rules are analysed to identify the signatures of input/output events. In IS UML, the event effects can be specified in UML state and collaboration diagrams or can be written directly in B, as a transaction corresponding to the event. The latter approach is similar to Catalysis actions[8] or Meyer's contracts [19], in that the transaction must enforce all system integrity constraints.

4. The B machine architecture is completed to express the whole specification. New machines are defined to provide necessary state and operation structures. These machines have no variables, and must call generated operations from included basic machines in order to update the system state. A top-level B machine forms the interface of the B system specification. It contains one B operation for each event.

5. The properties of the B specification can be proved using a B tool. In order to "validate" the specification animators can also be used. If the proof or animation identifies problems with the specification, the B and UML specifications need to be modified accordingly. This is where traceability is a key issue.

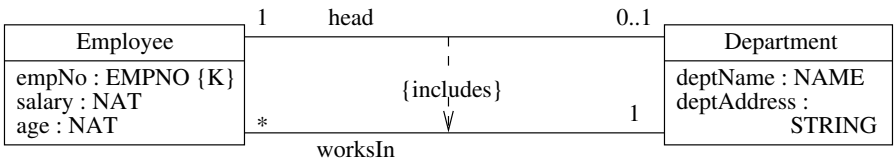

Fig. 3. Example of an IS UML Entity Class Diagram

4.1 Illustration of UML-to-B Translation on an Example

The first four steps of the translation process are demonstrated for a simple system of employees and departments. Steps 2 to 4 employ our existing translation rules[22, 21, 10], as implemented by the prototype tool assistant.

Step 1: Constructing the entity class diagram Figure 3 shows the entity class diagram for the example. There are two associations between the *Employee* and *Department* classes. Every employee *worksIn* exactly one department, and, in each department, exactly one employee is designated as the *head* of the department. The links of the latter are included in the links of the former association. The keyword $\{K\}$ against the *Employee* attribute *empNo* indicates that this is a key (unique identifier) of employees.

Step 2: The initial B representation The B model represents an entity class as a B given set. A variable represents the set of instances of the class via a typing invariant. In the example, thus, the *Employee* class is represented by the given set, *EMPLOYEE*, the variable, *Employee*, and the typing invariant,

$$Employee \subseteq EMPLOYEE$$

The B representations of attributes and associations are isomorphic. Each is modelled by a B variable and a typing invariant. For the attributes, we use B given sets *EMPNO*, *NAME* and *STRING*, and the B type *NAT*. The invariant links the variable representing the attribute to its class. For *Employee*, this gives:

$$\begin{array}{l} empNo \in Employee \rightarrowtail EMPNO \wedge \\ salary \in Employee \rightarrow NAT \wedge \\ age \in Employee \rightarrow NAT \end{array}$$

For associations, the B model captures the whole association semantics. A B variable represents the named association. A typing invariant models the association end classes and their multiplicity:

$$\begin{array}{l} worksIn \in Employee \rightarrow Department \wedge \\ head \in Department \rightarrowtail Employee \end{array}$$

The inter-association constraint is represented in a B invariant,

$$head \subseteq worksIn$$

Constraints are represented in B invariant expressions. Here, there are requirements that all employees are aged at least 18, that employees under 30 years old earn under 30000 euros per annum, and that the salary of the head of a department is at least as great as that of any employee of the department:

$$C1 : \forall e \in Employee \bullet age(e) \geq 18 \wedge$$
$$C2 : \forall e \in Employee \bullet age(e) < 30 \Rightarrow salary(e) < 30000\ euros \wedge$$
$$C3 : \forall e \in Employee \bullet salary(e) \leq salary(head(worksIn(e)))$$

There is one generated operation each for inserting and deleting an object (link) of each of class and association, and for updating each mandatory attribute of each class. The detail of each operations depends on the definition of keys, the integrity constraints, the multiplicity of the associations, and the machine in which an association is defined (for more details see [22]). Three operations are given here:

$UpdateSalary(emp, sal)$
PRE
$emp \in Employee \wedge sal \in Nat$
$sal \in Nat$
$THEN$
$salary(emp) := sal$
END

$InsertWorksIn(e, dep)$
PRE
$e \in EMPLOYEE - Employee \wedge$
$dep \in DEPARTEMENT$
$THEN$
$WorksIn := WorksIn \cup \{e- > dep\}$
END

$InsertEmployee(e, no_t, sal_t, age_t)$ PRE
$e \in EMPLOYEE - Employee \wedge no_t \in EMPNO - ran(empNo) \wedge$
$sal_t \in NAT \wedge age_t \in NAT \wedge age_t \geq 18 \wedge$
$THEN$
$Employee := Employee \cup \{e\} \parallel empNo := empNo \cup \{e- > no_t\} \parallel$
$salary := salary \cup \{e- > sal_t\} \parallel age := age \cup \{e- > age_t\}$
END

The B specification of each entity class is presented in a B machine. A B machine is also created for any association that is subjected to independent operations (ie operations on links between existing objects or on the attributes of the association). **USES** links are defined from the association machine to the machines of the associated entity classes. If the association has no independent operations, it is defined in one of the entity class machine, according to the multiplicity, with a **USES** link to the other machine[9, 22]. Generated operations are included in the relevant basic machine. Thus, *UpdateSalary()* and *InsertEmployee()* are defined in the machine *B_Employee*; *InsertWorksIn()* is defined in the machine *B_WorksIn*. The machine structure comprises the upper four machines in figure 4, below.

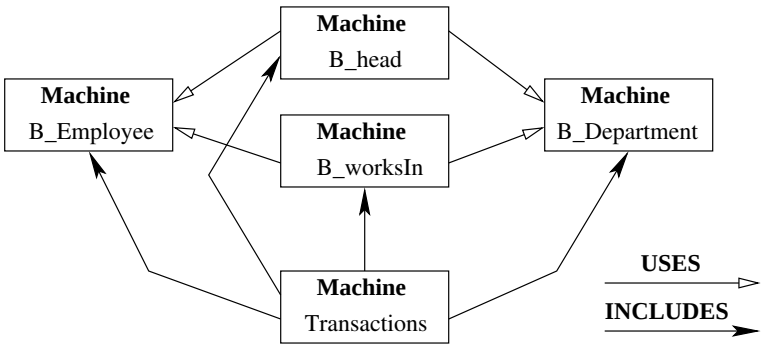

Fig. 4. Architecture of the final B specification

Step 3: Expressing input/output events In the example, the users require a transaction to insert an employee object and link the employee to a department. Here, we express this directly in B syntax:

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

Transactions such as *AddNewEmployee* are defined in a top-level *Transactions* B machine (figure 4). In general, it is possible to have an intermediate level of IS operation, between generated operations and transactions, *User Operations*, that define a context of use for generated operations. In essence, user operations differ from transactions only in their restriction to a single class or operation. A user operation can be used to specify additional preconditions dependent on the application, and to enforce the integrity constraints related to the class or association.

Step 4: Composing the complete B specification The definitions from steps 2 and 3 give rise to the B architecture in figure 4. The B machine architecture is similar to the structure of the UML class model. The **USES** structures mirror the classes and associations; the **INCLUDES** structure builds the top-level transaction machine on the whole state.

The levels of B machine facilitate evolution of the specification. In IS, user requirements and associated business rules frequently emerge during the development. The changes can be made in higher-level machines, without modification of the base machine classes, associations and generated operations. The B architecture also allows incremental proof.

4.2 Issues in the Development of Assistance Tools

One of the characteristics of the systems we specify is the great number of integrity constraints that link the different entities and associations. They are translated as state invariants in the B specification. Thus it is not obvious how to specify transactions that satisfy the constraints.

Once the state invariant is complete, two approaches can offer assistance in the development of consistent operations[24, 17]. Pro-active tool assistance would identify the parts of the invariant that are relevant to the operation, and then automatically generate preconditions and substitutions whenever possible, as in [18]. Subsequently, the specification would be presented to a B proof tool, which attempts to discharge the proof obligation. Assuming that the state invariant is sufficiently defined, the proof will fail if either the precondition or the substitution of an operation is insufficient. Error messages from the proof tool point to missing preconditions and/or substitutions. However, at present, these messages are often hard to understand. To be of value to a designer, the messages returned by the proof tool should be interpreted in terms of the UML models. For example, the omission of the call to *InsertWorksIn()* from the specification of the *AddNewEmployee()* transaction would produce a B prover error relating to the failure to prove the totality of the B function, *worksIn*. This needs converting into a comment on the failure to satisfy the multiplicity of the *Employee* association end of the *worksIn* association. This kind of assistance is called a reactive approach.

At present, for each UML construct, it is possible to identify all the corresponding B elements. To provide assistance as described below, we need to complete traceability between the textual (B) and the graphical (UML) specifications. As in the work of Freire[14], we use metamodels to unify the different views of the specification, as represented in the metamodels of the UML and B concepts.

5 Core IS UML Metamodels

Before defining the links between B and UML concepts, we summarise our IS UML metamodels[9, 24, 17]. These define the abstract syntax and semantics of core IS UML concepts, giving a precise definition of the relevant UML graphical notations. The metamodels are formally specified in B, and are used in the prototype tool.

In this section, graphical summaries (UML class model notations[25]) are presented, with the B representations for just a subset of the metamodels relevant to operation specification. The formal B metamodel is derived from the

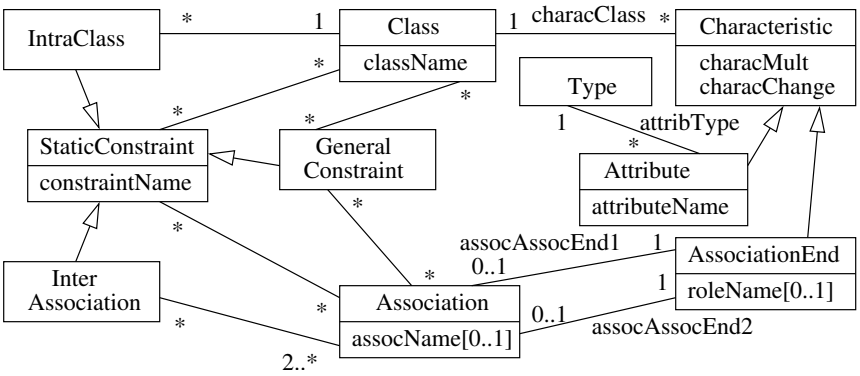

Fig. 5. IS UML Metamodel for Structural Concepts, from [9, 24]

diagrammatic representation using our translation rules, as in the illustrative translation, section 4, above. As above, the B expresses additional constraints on the structure.

5.1 Metamodels for the Structural Aspect

Figure 5 summarises the abstract syntax of the main structural concepts of IS UML. *Class*es have many *Characteristic*s, which are either *Attribute*s or *AssociationEnd*s. A *Characteristic* has attributes representing its multiplicity and whether its value can change; an *Attribute* also has a name, and is linked to an IS UML *Type*. An *Association* is linked to two distinct *AssociationEnd*s, each of which has a role name. *Class*es and *Association*s have many associated constraints (*IntraClass* and *InterAssociation* subclasses of *StaticConstraint*).

In [24], we define a hierarchical metamodel for constraint concepts, categorised by the parts of the structure on which they act. *IntraClass* constraints are the most common, and include *InterObject* constraints that relate the values of several objects of a class (or links of an association), and *IntraObject* constraints that relate to attributes within an object of a class (or a link of an association). *InterAssociation* constraints are less common. They capture subsetting and exclusion constraints on associations and association ends. *GeneralConstraint*s represents business rules and other constraints requirements, and relate more than one system class and/or association.

5.2 Metamodels for the Functional Aspect

The metamodel of IS UML operations is summarised here; metamodels for state and collaboration diagrams[9] are omitted.

The hierarchy of operations (figure 6) inherits attributes from the superclass, *Operation*. These attributes together define the signature of an operation, ie its structural aspect.

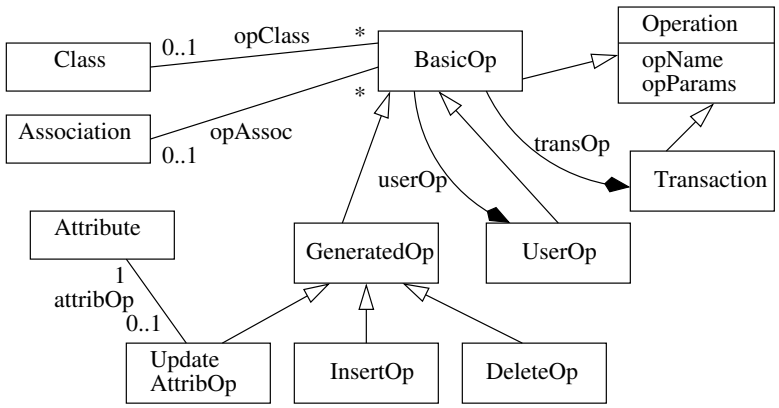

Fig. 6. IS UML Metamodel for Structural of Operations, from [17]

BasicOp and *Transaction* are subclasses of *Operation*. A *BasicOp* represents operations on one class or association, whereas a transaction, composed of basic operations, acts on many classes and/or associations. *BasicOp* is further specialised into *GeneratedOp* and *UserOp*. A *UserOp* is both a subclass of *BasicOp* and a composition of basic operations on a single class or association. *GeneratedOp*s are the generic, low-level operations of any IS, described above. The three subclasses, insert, delete, and attribute update operations, represent the only B operations that change the state; they cannot be modified by the designer.

5.3 B Representation of Abstract Syntax of Operation Concepts

The abstract syntax of the IS UML notations (figure 6) is represented in B according to step 2 of the process defined above.

The B metamodel requires a number of given sets, written in upper case. The given set, *TYPE*, is used to represent any relevant UML- and user-defined data types. The B given set, *NAME*, represents any class name in the metamodel.

Superclasses are represented in the same way as classes in section 4. Subclasses have the same type as their superclass; the instances are a subset of the superclass instances. The typing invariants for all these are given in table 1. An invariant is added to express the disjointness of subclasses. For example,

$$BasicOperation \cup Transaction = Operation \wedge$$
$$BasicOperation \cap Transaction = \varnothing$$

The *Operation* attributes are represented by total functions. These capture the fact that *opName* and *opParams* are both mandatory. Each could be associated with more than one operation, and a number of operations could have the same parameters:

$Operation$	$\subseteq$	$OPERATION$	$GeneratedOp$	$\subseteq$	$BasicOp$
$Class$	$\subseteq$	$CLASS$	$UserOp$	$\subseteq$	$BasicOp$
$Association$	$\subseteq$	$ASSOCIATION$	$DeleteOp$	$\subseteq$	$GeneratedOp$
$BasicOp$	$\subseteq$	$Operation$	$InsertOp$	$\subseteq$	$GeneratedOp$
$Transaction$	$\subseteq$	$Operation$	$UpdateAttribOp$	$\subseteq$	$GeneratedOp$

Table 1. B Variables and Invariants for Sets of Known Instances of Operations

$opClass$	$\in$	$BasicOp \nrightarrow Class$
$opAssociation$	$\in$	$BasicOp \nrightarrow Association$
$attribOp$	$\in$	$UpdateAttribOp \rightarrowtail Attribute$
$transOp$	$\in$	$Transaction \leftrightarrow BasicOp \wedge dom(transOp) = Transaction$
$userOp$	$\in$	$UserOp \leftrightarrow BasicOperation \wedge dom(userOp) = UserOp$

Table 2. B Variables and their Typing Invariants Representing Associations in the IS Metamodel

$$opName \in Operation \rightarrow NAME \wedge$$
$$opParams \in Operation \rightarrow seq(TPARAM)$$

The type of $opParams$, $seq(TPARAM)$, models an ordered list of parameters. $TPARAM$ is a triple of a name, a type, and an indicator of whether the parameter is input (i) or output (o):

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

Metamodel associations are expressed as relations or functions, table 2. Composition is not well defined in UML. Here, we simply translate the two compositions ($transOp$ and $userOp$) as if they were simple associations. Each is a relation, whose domain is constrained to be all the known instances of the class ($Transaction$, $UserOp$).

Three of the additional B invariants are illustrated here.

A $BasicOp$ (and thus any of its subclasses) must reference only one class or one association. The partial functions in the B representation of the associations require that a $BasicOp$ can be associated with at most one class or association. To ensure that it is associated with exactly one, the associations are defined to be disjoint and total with respect to $BasicOp$:

$$dom(opClass) \cap dom(opAssoc) = \varnothing \wedge$$
$$dom(opClass) \cup dom(opAssoc) = BasicOp$$

The uniqueness of operation names within each class (or association) is an example of the internal and mutual consistency properties of the metamodel.

The invariant states that, for any two operations of a particular class, if the operations have the same name then they are the same operation:

$$\forall c \in ran(opClass) \bullet$$
$$\forall o_1, o_2 \in opClass^{-1}[\{c\}] \bullet$$
$$opName(o_1) = opName(o_2) \Rightarrow o_1 = o_2$$

The third example requires that, for *UpdateAttribOp*s, the attribute updated must be of the class acted on by the superclass *BasicOp*:

$$\forall op \in UpdateAttribOp \bullet$$
$$attribOp(op) \in allClassAttributes(opClass(op))$$

where *allClassAttributes()* returns attributes of a class and its superclasses.

6 Modelling links between IS UML and B

To navigate from UML diagrams to B specifications and *vice versa*, we need to formally establish the correspondence between B concepts and the UML metamodel. The correspondence metamodels do not represent a complete metamodel of B concepts - B is fully defined in [1].

6.1 Modelling with B Machines

The practicalities of building an B specification from IS UML models led us to develop guidelines for B machine structures [12, 23], summarised in figure 7.

The instantiation of links from *BasicMachine* and its subtypes to the IS UML *Class*, *Association* and *GeneratedOp* are established during step 2 of the formalisation process. A *GeneratedOp* is always defined in a lowest-level basic machine, but cannot be accessed directly at this level.

The remaining links are established in step 4. All *BasicMachine*s are included in either *IntermediateMachine*s or *TransactionMachine*s. The *TransactionMachine* develops transactions required by users by combining (in parallel substitutions) the relevant operations defined on classes or associations.

In our operation metamodel, a *UserOp* is a combination of basic operations on one class or association. Figure 7 shows the ideal meta-structure, with *UserOp* represented in an B *IntermediateMachine* that **INCLUDES** the relevant basic machines. However, this is inhibited by the B rule that each operation of such a machine calls at most one operation of the included machine (because the parallel application of operations from an included machine may violate the invariant of the included machine[1, p317].) In effect, the including machine provides an interface and context for a single operation of the included machine. To take account of the B restriction, a further concept, a *BuiltOp* is added to the IS UML metamodel. This has no rationale other than to satisfy the needs of B specification.

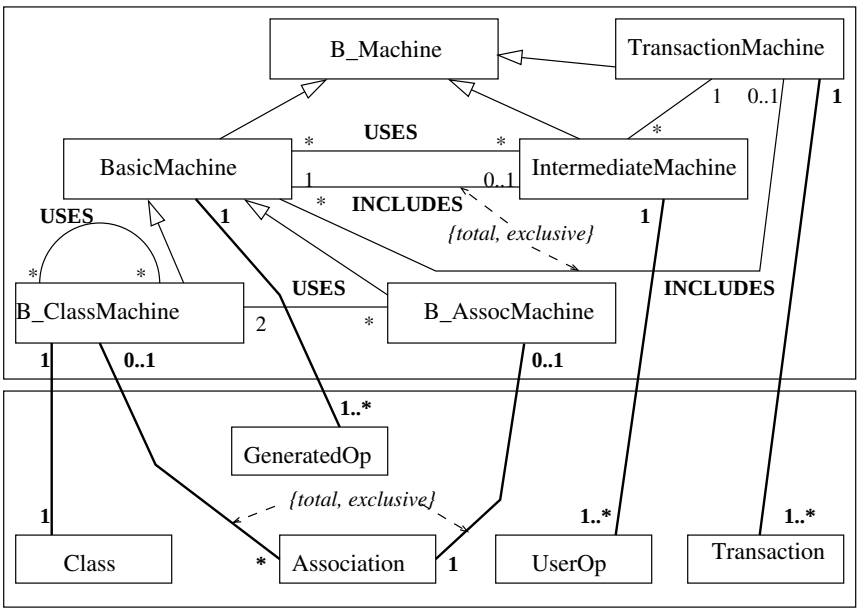

Fig. 7. B Machine Interrelationships and Links to IS UML Concepts

A *BuiltOp* is also required when a transaction needs to call several *GeneratedOp*s and/or *UserOp*s from one *BasicMachine* or *IntermediateMachine*. Thus a *BuiltOp* can be composed of either *GeneratedOp*s or *UserOp*s (figure 8).

A *BuiltOp* is automatically generated if the transaction requires such multiple operation calls. The *BuiltOp* is constructed by concatenation of the signatures of the required *BasicOp*s, conjunction of their preconditions and the parallel combination of their substitutions (resolving any duplication that occurs).

The use of the *BuiltOp* concept is illustrated for an operation that both updates the name of a department, and changes its address. The *GeneratedOp*s to achieve this are,

$$
\begin{array}{l}
UpdateName(dep, name) \\
\quad PRE \\
\quad\quad dep \in Department \\
\quad THEN \\
\quad\quad deptName(dep) := name \\
\quad END
\end{array}
$$

$$
\begin{array}{l}
UpdateAddress(dep, address) \\
\quad PRE \\
\quad\quad dep \in Department \\
\quad THEN \\
\quad\quad deptAddress(dep) := address \\
\quad END
\end{array}
$$

The *BuiltOp* to perform both updates is,

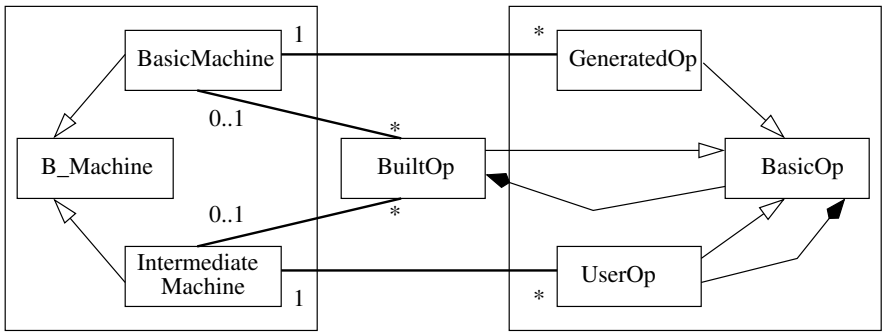

Fig. 8. B Machine and IS UML structures for User and Built Operations

$$
\begin{array}{l}
UpdateDept(dep, name, address) \\
\quad PRE \\
\quad\quad dep \in Department \\
\quad THEN \\
\quad\quad deptName(dep) := name \,||\, deptAddress(dep) := address \\
\quad END
\end{array}
$$

6.2 Links Between B Concepts for Structural IS UML concepts

Figure 9 presents the links between B concepts and structural IS UML components. The B concepts are used in the **SETS**, **VARIABLES** and **INVARIANT** clauses of basic machines. In addition, invariants defining integrity constraints appear in the **INVARIANT** of intermediate and transaction machines. The diagram omits the components of B **OPERATION** clauses, namely preconditions (*B_Expressions*) and substitutions[17, 9].

The links between B and IS UML concepts are self-explanatory. The B structures require some explanation. A *B_Expression* (and its subclasses) is any boolean expression, and is used to construct a *B_GlobalInvariant* by combining relevant *B_Invariants*. The *B_ElementaryInvariant* is required to fully define association end multiplicities in B: because tools such as AtelierB[7] do not have built-in proof rules for all function types, some functions are expressed by additional typing invariants on the domain, the range or the inverse of the function[7]. *B_Variables* represent class, attribute and association instances. The diagram models the links between these subclasses and the subclasses of *B_Invariant*. *B_ClassVars* and *B_AttribVars* are linked to *B_Types*; in IS specification, most types are represented as *B_GivenSets*, but the *B_BasicTypes* are also available.

7 Conclusion

The work presented here brings together the tool development and metamodelling components of our research into rigorous IS development. Whilst much of

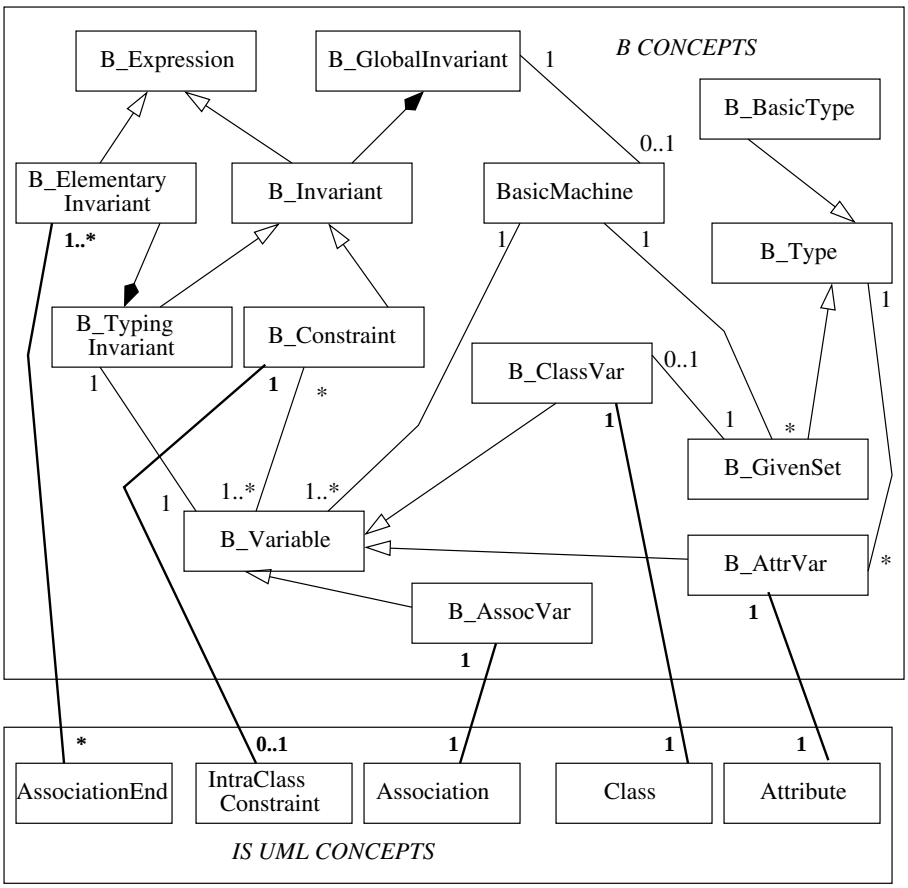

Fig. 9. B Concepts for Types and Invariants

the work is published elsewhere, the traceability links and their part in a tool architecture is new, and represents an important contribution to our long-term goal of a tool assistant for rigorous IS development.

Much of our research is forward-looking. Although commercial tools for diagrammatic and formal modelling have progressed in recent years, they do not yet provide all the features that we need. For example, tool interchange formats allow us to import to the prototype tool assistant textual representations of UML models, but we cannot specialise the UML tool to our IS UML syntax. Formal methods tools provide a wide range of verification, proof and animation facilities, but their user interfaces are inadequate for general use by non-experts. Tool assistants need to be capable of use by conventional designers with no more than a basic knowledge in formal specifications.

In this paper, we have not looked at how to refine an IS specified in IS UML and B. CEDRIC-IIE work has shown how relational database implementations can be formally derived from the B specification[15, 12]. Further research is planned into the connections between B refinement and the design process in UML. B is not ideal for the dynamic aspects, but there is ongoing research into event-oriented extensions of B[2, 3].

We have explored elsewhere[17] the relationship of our work to other combined notations for IS specification, and the relationship between our metamodels and the "standard" UML metamodelling attempts. Ongoing research is looking at how our B metamodels relate to these UML-OCL metamodels and metamodelling approaches. This involves working with members of pUML, with the aim of producing a fully-conformant IS UML, when the UML 2.0 "standard", with extension and specialisation mechanisms, is published. This will facilitate use of our tool assistant with commercial UML tools.

References

1. J-R. Abrial. *The B-Book*. CUP, 1996.
2. J-R. Abrial. Event driven sequential program construction. In *AFADL01: Approches Formelles dans l'Assistance au Développement des Logiciels, Nancy, France*, June 2001.
3. 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.
4. A. Clark, A. Evans, S. Kent, and P. Sammut. The MMF approach to engineering object-oriented design languages. In *Workshop on Language Descriptions, Tools and Applications, LTDA2001*, April 2001.
5. T. Clark and A. Evans. Foundations of the Unified Modeling Language. In *Second Northern Formal Methods Workshop*, volume 1241 of *LNCS*. Springer Verlag, 1997.
6. Rational Software Corporation. *Rational Rose - Using Rational Rose 98*. Rational Inc., 1998.
7. DIGILOG. *Atelier B, Manuel de Référence*. DIGILOG groupe STERIA, BP 16000, 13791 Aix-en-Provence Cedex 3, France, 1996.
8. D. F. D'Souza and A. C. Wills. *Objects, Components and Frameworks in UML: the Catalysis Approach*. Addison-Wesley, 1999.
9. 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, September 1999.
10. P. Facon, R. Laleau, and H. P. Nguyen. Dérivation de spécifications formelles B à partir de spécifications semi–formelles de systèmes d'information. In *1st B Conference, Nantes, France*, 1996.
11. 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.
12. 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, September 1999.

13. R. B. France and M. M. Larrondo-Petrie. A two–dimensional view of integrated formal and informal specification techniques. In *Ninth International Conference of Z Users, Limerick, Ireland, September 1995*, volume 967 of *LNCS*, pages 434–448. Springer Verlag, 1995.
14. J-C. Freire Junior. Pouvoir d'expression de modèles orientés objet. *Ingénierie des Systèmes d'Information*, 4(2):219–237, 1996.
15. R. Laleau and A. Mammar. A generic process to refine a B specification into a relational database implementation. In *Proceedings, ZB2000: Formal Specification and Development in Z and B, York, August-September 2000*, volume 1878 of *LNCS*, pages 22–41. Springer Verlag, 2000.
16. 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.
17. R. Laleau and F. Polack. Specification of integrity-preserving operations in information systems by using a formal UML-based language. Accepted for publication, Information and Software Technology, 2001.
18. 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.
19. B. Meyer. *Object-oriented software construction*. Prentice Hall, 1997.
20. E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. In *FM99, Toulouse, France*, volume 1708 and 1709 of *LNCS*, pages 875–895. Springer Verlag, 1999.
21. F. Monge. Formalisation du méta modèle des méthodes graphiques d'analyse et conception orientées objet. Master's thesis, Institut d'Informatique d'Entreprise, Conservatoire National des Arts et Métiers, Evry, September 1997.
22. H. P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi–formelles*. PhD thesis, Laboratoire CEDRIC, Conservatoire National des Arts et Métiers, Evry, December 1998. Available from www.iie.cnam.fr/~ laleau/thesePHNguyen.ps-gz.
23. F. Polack. Exploring the informal translations of OMT object models in B. Technical report, University of York, forthcoming, 2001.
24. 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.
25. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Guide*. Addison-Wesley, 1998.
26. L. T. Semmens, R. B. France, and T. W. G. Docker. Integrated structured analysis and formal specification techniques. *The Computer Journal*, 35(6):600–610, 1992.