Article:
Laleau, Regine and Polack, Fiona orcid.org/0000-0001-7954-6433 (2008) Using formal metamodels to check consistency of functional views in information systems specification. Information and Software Technology. pp. 797-814. ISSN 0950-5849

https://doi.org/10.1016/j.infsof.2007.10.007

Using formal metamodels to check consistency of functional views in information systems specification

Régine Laleau [a,*], Fiona Polack [b]

[a] *Research Laboratory LACL, University of Paris 12, IUT Fontainebleau, Route forestière Hurtault 77300 Fontainebleau, France, Telephone: (+33) 1 60 74 68 40, Fax: (+33) 1 60 74 68 02.*

[b] *Department of Computer Science University of York, York, YO10 5DD, UK. Telephone: +441904432722, Fax: +441904432767*

Abstract

UML notations require adaptation for applications such as Information Systems (IS). Thus we have defined IS-UML. In this article, we propose an extension to this language to deal with functional aspects of IS. We use two views to specify IS transactions: the first one is defined as a combination of behavioural UML diagrams (collaboration and state diagrams), and the second one is based on the definition of specific classes of an extended class diagram. The final objective of the article is to consider consistency issues between the various diagrams of an IS-UML specification. In common with other UML languages, we use a metamodel to define IS-UML. We use class diagrams to summarize the metamodel structure and a formal language, B, for the full metamodel. This allows us to formally express consistency checks and mapping rules between specific metamodel concepts.

Key words: Information System Design, Unified Modelling Language Notation, Metamodel, Formal Notation

* Corresponding author.
Email addresses: laleau@univ-paris12.fr (Régine Laleau), fiona@cs.york.ac.uk (Fiona Polack).

1 Introduction

This paper reports part of an ongoing research programme bringing formality to the development of information systems (IS) [13]. This section motivates our work, then provides a general introduction to the relevant aspects of IS and formal methods.

Formality, or the addition of mathematical notations to conventional diagram-based modelling, allows us to prove the consistency of the model at each stage of development. This improves the verification of models, bringing verification activities into the development process, rather than relying on implementation testing.

Our work has two aspects. The first is to provide a rigorous underpinning to IS modelling. This entails identifying and modifying UML modelling notations that are appropriate to IS, and extending the UML metamodel to represent the IS-UML. A complete formal metamodel is also prepared. The second aspect is to provide automatic translation of IS-UML models into formal models. Elsewhere [16,15], we have shown that, with the help of these formal models, it is possible to construct tools that can automatically generate a consistent implementation of the core functionality (ie a relational database schema) of an IS.

1.1 Information systems and their modelling

IS are not complicated computational systems; they just manage a large amount of data. Data is often a commercially-critical resource; its protection and reliability are essential. Data security is concerned with confidentiality (or secrecy), integrity, and availability. Data integrity is paramount; it is facilitated by the structural properties of relational databases (the main target implementation of most IS developments). Additional IS static integrity constraints represent the business or enterprise rules of the system, proposed by clients.

Data modification has the potential to break data integrity. At the implementation level, an IS generally relies on a database management system (DBMS), which uses *transactions* to ensure that operations on the database leave the database in a state that is consistent with its definition. A transaction is made up of low-level operations on particular data. It includes all the low-level operations needed to make the necessary changes to data values, and to return the database to a consistent state. In addition, the transaction may provide authorization, audit and other security functions, and various interface functions.

The low-level operations of the database (eg. SQL statements) do not preserve integrity. For example, a creation operation simply adds a new *tuple* (row) to a single database table, without inserting any required links to other tables. Where tables have mandatory links, such a creation operation leaves the database in an inconsistent state — a structural constraint is broken. The DBMS guarantees atomic execution of transactions; if a component low-level operation cannot execute, or fails to complete, then the whole transaction aborts, and the database is restored to its (consistent) state before the transaction started.

In the work presented here, we consider *update transactions* (as opposed to read-only transactions that do not change the state of the database). Our models express static constraints, but do not consider dynamic constraints (such as temporal and ordering constraints).

IS development uses graphical notations for its models, justifying this with the assertion that diagrams help in communication with clients. For data modelling, it has been traditional to use variants of semantically well-defined entity-relationship (E/R) diagrams. More recently, UML class diagrams have been widely used, implicitly adopting an E/R semantics. UML class diagrams include concepts that are not required for database modelling, and lack some important aspects such as keys. Rigorous use of UML for data modelling requires some modification of conventional UML class diagram metaconcepts.

IS functional specification is often neglected, or is achieved by inelegant use of techniques designed for procedural program design (data flow diagrams, Jackson structures such as entity life histories). Within UML, there are a number of models that can be adapted to IS behavioural modelling. We focus on use cases, state diagrams (a class or table view of behaviour) and collaboration diagrams (a transaction view of behaviour). Even in conventional UML, the semantics of these notations are not always transparent, and the relationship between metaconcepts of different models are not well-defined. For example, the links between operations on classes and state diagram actions, between messages and events, and between the parameters of events, messages and operations, are obscure.

1.2 IS-UML for formal modelling of IS

Model-based formal methods such as B or Z are well suited to the expression of models with important static constraints, and to the verification of these constraints across state changes. In our work, we have favoured B: in [11] we justify our use of B for IS-UML language definition.

A B model can be built incrementally from simple *machines*, which, for IS,

can model a single IS entity or table and the low-level operations on it. Transactions can be modelled as assemblies of simple machines that provide the integration and interfacing needed for the low-level operations in the simple machines. The Appendix A gives a summary of B notations used in this paper. For a full account, see [1].

A further advantage of using B is the development support. Commercial tools such as AtelierB (used widely in France, for example in the development of transport control systems [3]) provide modelling and proof support. The tool checks the type-correctness of the formal model, then generates the proof obligations (POGs) needed to prove the consistency of each model. In analysis of B models of IS, we find that these POGs are almost always discharged automatically [17].

To introduce formality to IS development, we have developed the IS-UML notation and tool [8,9]. Our language definition in B is supplemented by graphical summaries of the abstract syntax (AS) of IS-UML, using UML class diagrams. (When the IS-UML work was started, UML metamodels were the only graphical form available.)

The graphical summaries allow comparison with the Object Management Group (OMG) metamodels for UML, MOF etc. Like OMG, our approach to metamodelling requires expression of the abstract syntax of the modelling language. In the UML community, OCL [28] is used to elaborate diagrammatic models with static constraints. OCL provides a means of referring to diagram components, of navigating links in diagrams, and a basic set-based logic for expressing constraints. IS-UML does not use OCL. Although OCL syntax and semantics is being addressed with some rigour, the language is not stable; it does not express the whole UML model, only the additional constraints; and it is not supported by full formal analysis tools. We present the IS-UML definition in B. This B metamodel adds predicates to complete the AS definition and to give the static semantics of the notations. We believe that it would be possible to add facilities for working with OCL to the IS-UML metamodels, to support the whole of the relevant parts of UML.

For an IS-UML development, we take UML class diagrams and various UML behavioural diagrams and specialize them for IS specification. We use our defined translation rules to extract from these diagrams a B model that permits consistency checking of the specification. The method has three main elements:

(1) Generate a B specification of the static structure and elementary operations of the system – this starts from a class model extended with added IS features such as static constraints [10].
(2) Generate transactions from IS-specialized UML state and collaboration diagrams. The IS-UML diagrams are annotated with formal definitions

derived from the formal model of the static structure.

(3) Prove the consistency of the model, by discharging the consistency proofs derived by the B prover (here, Atelier-B).

The graphical approach suits traditional IS developers. It is also a good medium for summarizing the structure of each view of a model and facilitates readability and thus comprehension of the specification. For instance a single class diagram generates several pages of B notations which have to be read line by line, page by page, whereas the graphical notations present several dimensions of reading on one page.

To use our IS-UML tool, a diagram tool with serialisable output is needed. We use the ROSE environment to construct the UML diagrams. In order to apply our B translation tool, we must ensure that the diagrams conform to the IS-UML notations. However environments such as ROSE do not provide elaborate consistency checking either between or within diagrams — the translated B model could be meaningless or produce type checking errors that are difficult to interpret. Instead, we verify the graphical model before translating it to B. To do this, we need to formally define the syntax and semantics of the IS-UML notations, and to express formally the correspondences between concepts in different diagrams.

There are two possible levels of consistency checking. The first level consists of simply checking that the syntax of the diagrams is valid. For this, our formalization of metamodels is sufficient. It gives well-formedness rules for the different diagrams and allows checks of intra and inter diagrams consistency. The next level consists of verifying the "behaviour" of the specified system. This implies describing the dynamic semantics of the metamodels. The formal expression of the behavioural semantics of structural concepts comprises the specification of the semantic domain and of the links between the concepts and the semantic domain. It is difficult to directly specify or validate a formal dynamic semantics of transactions. However, a semantics of the functional aspects is already given by the translation from IS-UML to B; the discharging of the B proofs demonstrates that a transaction semantics is valid.

In this paper, we focus first on the specification of functional aspects of the IS, elaborating our previous published accounts, and then on the way of ensuring a consistent global specification. We present a simple example in UML, explaining its direct translation into B (Section 2). We then outline how we have formalized class diagram notations (Section 3.1), and extend the formalization to state and collaboration diagrams (Sections 3.2, 3.3). Consistency rules and relationships between all the diagrams are considered further in Section 4, whilst overall evaluation and conclusions appear in Section 5.

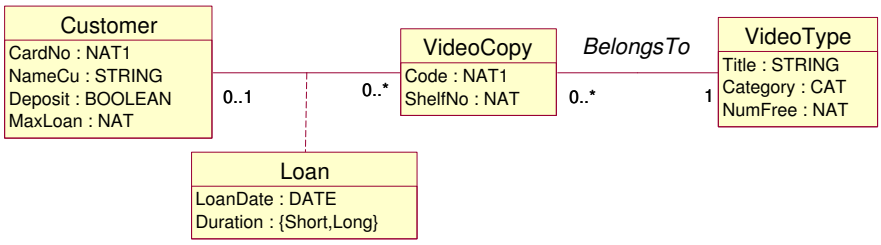

Fig. 1. Class Diagram of the Video-club Example

2 IS-UML diagrams and their B representation

This section introduces a simple example (used throughout the paper). We use this example to review our earlier work on class diagrams in IS-UML and their translation into B. We only present the basic concepts needed to understand our overall approach. The section summarizes static aspects and develops different ways of modelling transactions. It outlines the restrictions and extensions of UML that can be applied and how we use UML with B to present a full, verified model of an IS.

Example

A video club's loan system is modelled as a UML class diagram (Figure 1). The video loan system comprises *Customer*, *VideoCopy* and *VideoType* classes. The association between *VideoCopy* and *VideoType*, called *BelongsTo* models a *VideoCopy* belonging to exactly one *VideoType*; each *VideoType* can have many *VideoCopy*s. Between *Customer* and *VideoCopy*, there is an association class, *Loan*, modelling the current loan of a *VideoCopy* by a *Customer*; a *Customer* can loan many *VideoCopy*s, and a *VideoCopy* can be loaned at a time to one *Customer*. In the model, *CardNo* identifies a customer instance, *Code* identifies a video copy instance, and *Title* identifies a video type instance. On *Customer*, the *Deposit* attribute is true if a customer has made a deposit, which allows him to benefit from a long duration loan, and false otherwise, which restricts him to a short duration loan. The *MaxLoan* attribute determines the maximum number of loans a customer can have at the same time. On *VideoType*, the *NumFree* attribute models the number of actual video copies for a particular *Title* that are available for loan at a particular time. Other attributes are self-explanatory.

2.1 B generation of static structure and elementary operations

Our approach starts by formalizing the class diagram. Here, we summarize the mapping rules (for full treatment, see [21] for class constructs and [11,16] for allocation of formal statements to specific B machines).

For each IS-UML class A, we define an abstract set S_A of all possible instances of A. A variable v_A represents the set of existing instances of A, a subset of S_A. Each attribute Att of the class A is modelled by a variable v_{Att} which is a binary relation between v_A and the type T_{Att} of the attribute. Depending on the multiplicity of the attribute, this relation may become a function ($\rightarrow$), an injection ($\rightarrowtail$), etc. We impose the rule that UML basic types correspond to B basic types. An association Ass between two classes X and Y is modelled by a variable v_{Ass} defined as a relation between the existing instances of the two classes v_X and v_Y. The form and the directionality of this relation depends on the multiplicity of each association end (or *role*). An attribute of an association is formalized like an attribute of a class. A summary of association multiplicities and their corresponding B translations is given in Appendix B.

The B specification is built up from B machines; classes are defined in the lowest level machines, which include elementary operations to create, delete and modify instances. Associations, user-defined operations and transactions are built on top of these machines, which are referenced by the B *USES* or *INCLUDES* mechanisms as appropriate. The Appendix A summarizes the B machine syntax used in this paper. Within each machine, abstract sets are defined as B *SETS*; variables representing classes, attributes and associations are defined as B *VARIABLES*. The B *INVARIANT* expresses typing, associates attributes to classes, and models associations. For example, the B machine for the class *Customer* in Figure 1 would contain the clauses,

SETS
$CUSTOMER$
VARIABLES
$Customer$, $CardNo$, $NameCu$, $Deposit$, $MaxLoan$
INVARIANT
$Customer \subseteq CUSTOMER \ \wedge$
$CardNo \in Customer \rightarrowtail NAT_1 \ \wedge$
$NameCu \in Customer \rightarrow STRING \ \wedge$
$Deposit \in Customer \rightarrow BOOLEAN \ \wedge$
$MaxLoan \in Customer \rightarrow NAT$

The invariant states that the *Customer* variable, which represents the set of known instances of customers, is a subset of the set of all possible customers, *CUSTOMER*. The remaining four clauses bind the attributes, shown in figure

1, to *Customer* and give their type. Note that *CardNo*, which is the primary key, or unique identifier, of the class, is defined using a total injective function, capturing the fact that there must be a value for each customer's *CardNo*, and that each value is unique to one customer.

The associations *Loan* (and its attributes, *LoanDate* and *Duration*) and *BelongsTo* are defined in machines with a B *USES* relationship to the *customer*, *videoCopy* or *videoType* machines, as appropriate. The associations are defined by the following invariant:

INVARIANT
$BelongsTo \in VideoCopy \rightarrow VideoType \ \wedge$
$Loan \in VideoCopy \nrightarrow Customer \wedge$
$LoanDate \in Loan \rightarrow DATE \ \wedge$
$Duration \in Loan \rightarrow \{Short, Long\}$

BelongsTo is a function between two sets of class instances, representing the $0..*$ *to*1 association in figure 1. The association class *Loan* is first defined as the function between *Customer* and *VideoCopy*, and then its class attributes are added in the same way as for an ordinary class, above.

In IS, all operations are built from elementary create, delete and update operations. These have a generic form, namely a standard precondition and a standard effect – for a create operation, the precondition is always that the instance does not exist, and the effect is to add the instance to the set of known instances. Thus, the elementary operations descriptions can be automatically generated from the class diagram[11]. We generate operations for inserting or deleting objects of a class or links of an association and for modifying attributes. The form is illustrated in the following B operation specification, to add a new instance of customer.

$B\_Add\_Customer(cu, nu, na, de, ml)$
PRE
$cu \in CUSTOMER - Customer \wedge$
$nu \in NAT_1 \wedge$
$nu \notin ran(CardNo) \wedge$
$na \in STRING \wedge$
$de \in BOOLEAN \wedge$
$ml \in NAT$
THEN
$Customer := Customer \cup \{cu\} \ ||$
$CardNo := CardNo \cup \{cu \mapsto nu\} \ ||$
$NameCu := NameCu \cup \{cu \mapsto na\} \ ||$
$Deposit := Deposit \cup \{cu \mapsto de\} \ ||$
$MaxLoan := MaxLoan \cup \{cu \mapsto ml\}$

END

The five parameters represent input to the operation – a customer instance (*cu*), and its four attribute values. The B **PRE** clause, or precondition for the operation, gives the required types for the five parameters, and also states that the input customer and the value for the *CardNo* attribute, *nu*, must not already exist (a new customer is not an existing customer, and it should not be possible to create a new customer using an existing card number). The operation is specified as the parallel composition of five B substitutions, each of which assigns the relevant parameter to the system state. Thus, the customer, *cu* is added to the set of existing customers, *Customer*, and each attribute value is coupled to this customer instance, and added to the set of attribute couplings.

Thus, the preconditions enforce the invariant of the machine representing the class, and always include the assertion of the parameter types. The operation adds appropriate instances to each state variable.

So far we have used only information extracted from the class model to derive the B specification. In order to address system-level behaviour (as opposed to elementary operations), we need to consider how database transactions can be specified, and what UML notations can be adapted to provide material for a full behavioural specification.

2.2 Different views of user transactions in IS-UML

Now the specification of the example IS is extended to model behaviour. To specify the functional aspects of the information system, we use two views to highlight different aspects of transactions. The first uses UML behavioural diagrams, state diagrams and collaboration diagrams. The second allows transactions to be directly specified in an extended class diagram.

The two views are designed after the generation of B specifications from the class diagram since they use the names from the class diagram and the generated operations.

Before considering the two views, we note that, in IS-UML, use cases give a general view of user transactions. A use case contains all the transactions related to the same business sub-system. For example, the *Loan_Management* use case gathers all the transactions creating or deleting a loan, changing its duration, etc.

2.2.1 UML behavioural diagrams for IS behaviour

A transaction can be specified by state diagrams and collaboration diagrams. IS-UML imposes some constraints on their use, to capture specific features of IS and to facilitate the B translation. First, we present examples of both models in graphical form. We then illustrate their use in the B translation of the example system.

Graphical representation

State diagrams model the permitted sequences of states of an instance, from creation to deletion. There is at most one state diagram per class or association. Each arc represents a state transition, and is labelled,

EventName [Guard] / Action

The event that triggers a transition may be generated internally or externally.

In formalizing the information from the state diagrams, there are two ways to handle state information. The first, which we do not favour, is used in much action-based modelling. In this approach, a state variable is included in each class with the sole purpose of determining the state of each instance. Thus, in figure 2, there would be states, *LongLoan* and *ShortLoan*. Transitions can then be written to change the value of the state variable. This approach is not natural in a model-based approach, and we prefer to distinguish states by their characteristics – each state in a state diagram represents can be defined in terms of the range of values that the variable can take whilst an instance is in that state. Thus, for the class *Loan*, the *LongLoan* state applies to all instances of the class for which the value of the "Duration" attribute (a native attribute of the class) is "Long".

Through this definition of states, it is easy to see that state in the state diagram, and, indeed, guards on state transitions, can be written as Boolean conditions on variables, and can be expressed as predicates, in B invariants. We can also associate such Boolean statements with a name, akin to defining the state variable, as illustrated in the *Create_Loan*() operation, below, in the B **DEFINITION** clauses. Thus, IS-UML imposes two rules to assist the B translation of state diagrams:

(1) States and guards must be defined as well-formed B predicates.
(2) The name of an action on a transition must correspond to the name of a B operation of the relevant class.

Figure 2 is the state diagram for the video club's *Loan* association class. A loan instance is created with either a long or a short duration, depending on the

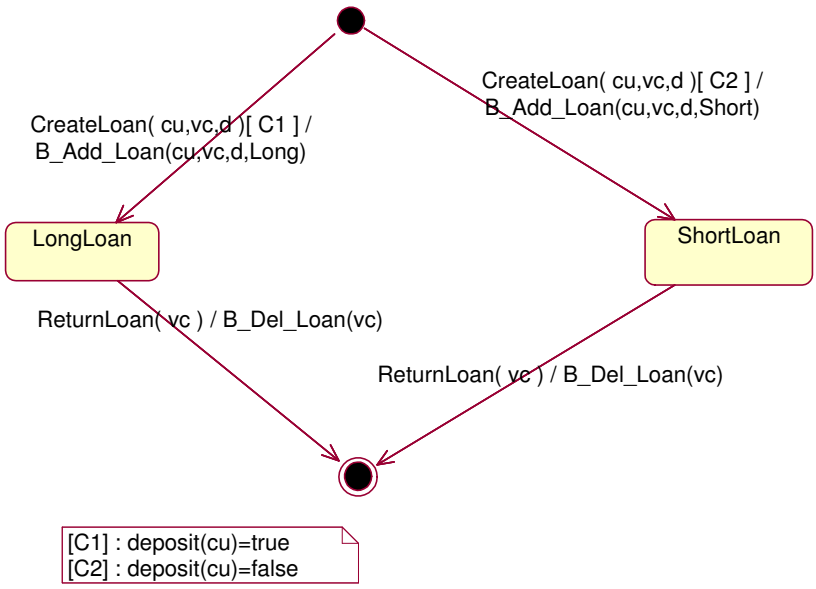

Fig. 2. State Diagram of the association class *Loan*

value of the *deposit* attribute of the customer that makes the loan. The event *CreateLoan(cu, vc, d)* can trigger either of the two transitions; the transition actually fired depends on the guard (*C1* or *C2*). In IS-UML, we impose the condition that exactly one guard is true at any time that an event occurs. A guard must be expressed by the user as a B predicate. For example, the condition corresponding to *C1* is:

$$deposit(cu) \;=\; TRUE$$

A loan instance is created by the B operation *B_Add_Loan*, called with parameters *(cu, vc, d, x)* where x stands for Short or Long. The first three parameters are provided by the triggering event. Once created, the loan instance is only deleted on receipt of the event, *ReturnLoan(vc)*.

Collaboration Diagrams are used to express IS transactions; these typically incorporate operations on more than one class, and may have extra preconditions expressing business rules. In the example, there is a simple transaction to create a loan instance, Figure 3.

An IS-UML collaboration diagram belongs to exactly one use case. We require that this is represented by a controller "class", which receives the input message that triggers the transaction. In this simple transaction, the controller is responsible for generating the internal message calls to create the loan and update the number of free copies of the video type corresponding to the requested video copy. Here, one of the subsequent triggers is the internal *CreateLoan(cu,vc,d)* event, which appears in the state diagram, Fig. 2 above. The other trigger is a direct call to the B operation *B_Change_NumFree*, spec-

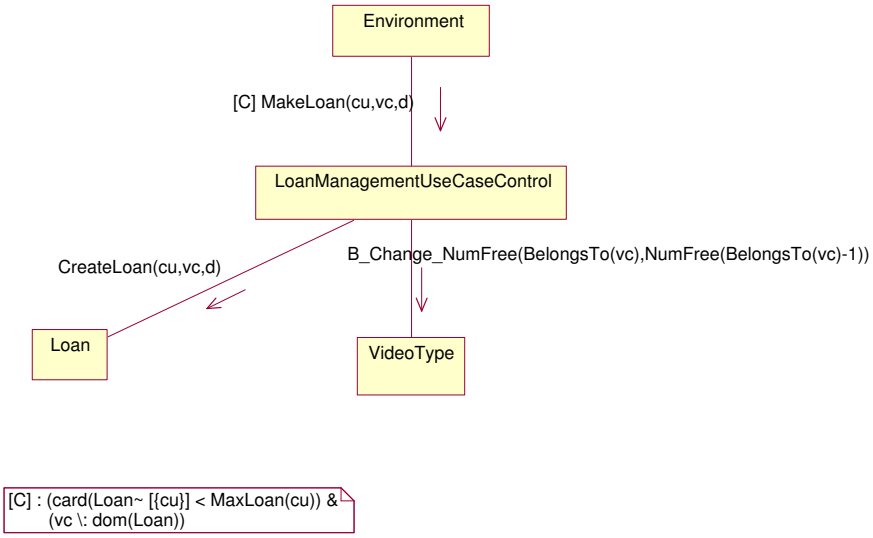

Fig. 3. Collaboration Diagram of the Transaction *MakeLoan*

ified on the *VideoType* class. The message that triggers the transaction can be guarded. Thus, the condition

$$(card(Loan^{-1}[\{cu\}]) \ < \ MaxLoan(cu)) \ \wedge \ (vc \ \notin \ dom(loan))$$

tests first that the cardinality of the set of loans for the customer (obtained by applying the inverse of the function defining the *Loan* association to the relevant customer) is less than the maximum number of loans authorized for the customer; secondly, the constraint tests that the requested video copy is available.

To understand the full transaction, the information in the collaboration, class and state models must be combined. Thus, the state model indicates that *CreateLoan(cu,vc,d)* results in the direct addition of the loan instance, with a duration value corresponding to the deposit of the customer. From the collaboration diagram, we see that the transaction to create a loan not only create a new loan instance but also decreases the number of free copies available for the video type of the requested video copy.

B representation

The B translation of behaviour is built on the lowest level B machines, the B translation of the class model (see [11] for details).

The state diagram for a class is used to create internal operations built up from elementary operations, with user-defined constraints. For example, the B operation corresponding to the *CreateLoan(cu,vc,d)* event is defined in a

B machine, called *SD-Loan*. *SD-Loan* **INCLUDES** the *Loan* machine, which gives it access to the elementary operations of *Loan*. *SD-Loan* **USES** the *Customer* and *VideoCopy* machines, which allows it to access their state. B **DEFINITIONS** are used to express states, simplifying subsequent clauses of the *SD-Loan* machine. For example, the clause

DEFINITIONS
$Initial(vc, cu) == vc \in VideoCopy \ \wedge \ cu \in Customer \ \wedge \ (vc, cu) \notin Loan$
$LongLoan(vc, cu) == (vc, cu) \in Duration^{-1}[\{Long\}]$

states that, first, in the *Initial* state, the video copy and the customer referred to here do exist, secondly that the loan instance *(vc,cu)* does not already exist, thirdly that a loan instance *(vc,cu)* is in the *LongLoan* state if its *Duration* attribute is *Long*.

The B operation to create a loan is,

$CreateLoan(cu, vc, d)$
PRE
$Initial(vc, cu) \ \wedge$
$(deposit(cu) \ = \ TRUE \ \vee \ deposit(cu) \ = \ FALSE)$
THEN
SELECT $deposit(cu) \ = \ TRUE$
THEN $B\_Add\_Loan(cu, vc, d, Long)$
WHEN $deposit(cu) \ = \ FALSE$
THEN $B\_Add\_Loan(cu, vc, d, Short)$
END
END

The precondition (**PRE** clause) contains first the Boolean definition of the required state for this operation to take place (the starting state in figure 2). Secondly, from figure 2, we note that two different creations are possible – one guarded by each of the disjoined predicates on $deposit(cu)$ – this disjunction is a tautology and could be omitted from the B, but is retained to demonstrate this point. There is also an implicit precondition that establishes the preconditions of the called elementary operations.

Two proof obligations are generated. The first one is a standard B proof obligation and expresses the fact that the operation representing the action on the transition preserves the invariant of the machine. The second one is generated from the state diagram and allows us to proof that the operation establishes the predicates of the target states, here the *LongLoan* state or *ShortLoan* state.

To model full transactions, the relevant collaboration diagram is translated

into an operation on a high-level B machine which includes all the machines for relevant state diagrams, and any required basic machines that are not otherwise included. The transaction-level machines are the only B machines that specify an external interface; other machines, and the state and operation components that they specify, are internal to the system.

The *MakeLoan(cu,vc,d)* transaction from Figure 3 is specified as:

$MakeLoan(cu, vc, d)$
PRE
$card(Loan^{-1}[\{cu\}]) < MaxLoan(cu) \wedge$
$vc \notin dom(loan)$
THEN
$CreateLoan(cu, vc, d) \,||$
$B\_Change\_NumFree(BelongsTo(vc), NumFree(BelongsTo(vc)) - 1)$
END

The precondition of this operation is the guard of the triggering message but it also implicitly includes the preconditions of the called operations. Again, proof obligations are generated to ensure that all the operations, and thus the whole transaction, preserve the global invariant of the machine and its included machines.

2.2.2 Specifying IS transactions with extended class diagrams

When modelling IS with UML, there are things that cannot be expressed in state or collaboration diagrams. For instance, a common problem is modelling a transaction that affects a number of objects of the same class, or modelling global integrity constraints.

Rather than further adapting the UML behavioural models, we propose an alternative approach that extends the class diagram. UML *stereotypes* are used to add distinct *control* classes, dedicated to the management of transactions.

In UML metamodelling, stereotypes are used to define different forms of class, whilst properties allow definition of new metamodel attributes. Three stereotypes are used in the UML metamodels [25].

(1) *Entity classes* are used to describe the static part of a system. Here, the part of a UML class diagram that contains only the entity classes and their associations corresponds to the E/R semantics needed for IS class diagrams.
(2) *Boundary classes* are used to describe the system interface, and are not considered further here.

(3) *Control classes* allow the description and co-ordination of the functionality of a system. We use these classes to specify transactions.

We define two kinds of *Control* class, to facilitate B translation and proof. An *EntityControl class* is linked to one entity class [1] (or an association class). It allows control of access to the operations of the class (or association class) and checking of local integrity constraints. A *TransactionControl* class effectively describes transactions. It can be linked to *EntityControl* classes and/or directly to entity classes. Each *TransactionControl* class describes the transactions of a single UML use case. It can also be used to express integrity constraints involving several classes or associations.

The links that relate control classes to entity classes represent control flows, rather than conventional association links. Thus we define the control flows as UML dependency links with stereotypes. Our approach is a variation of that of Treharne [27], who uses *Control classes* for general system specification. The concept of association links all classes independently of their kinds (entity or control). This is inadequate for our models where two stereotypes are distinguished: $\ll$ op $\gg$ represents calls to operations of the target class; $\ll$ var $\gg$ represents read-only use of variables. In translation, these correspond to B *INCLUDES* and *USES* references to lower level machines, respectively.

Illustration of control classes

In the video club system, there is a transaction *LoanByCategory(cu,cat)*, that allows a customer to select for loan any video copy in the stated category. (This belongs to the use case, *LoanManagementUseCase*.) To model the transaction using *Control classes*, the original class diagram is extended as shown in Figure 4.

The first extension adds an *EntityControl* class to *Loan*. *LoanControl* has an operation *CreateLoan* that accesses the elementary operation *B_Add_Loan*. It also verifies multiplicity and integrity constraints, through access to the variables of *Customer* and *VideoCopy*.

The second extension adds a *TransactionControl* class for the required transaction. *LoanManagementUseCaseControl* gives access to the *CreateLoan* operation, and to the elementary operation *B_Change_NumFree* of *VideoType*. The relevant constraints are also verified.

We use the B language to describe operations, since our objective is to produce complete B specifications [2]. As before, a B operation has a precondition that

[1] We continue to call *class* an entity class, to simplify notations.

[2] In our work, B is a natural choice for these descriptions. OCL lacks object cre-

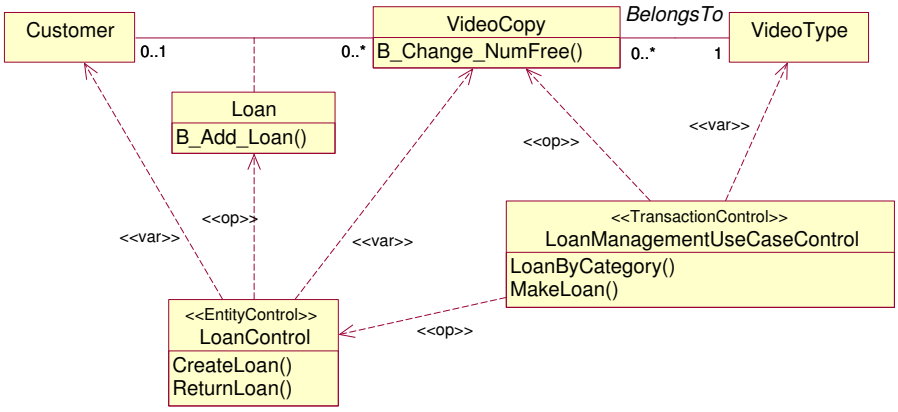

Fig. 4. The Example Class Diagram Extended to Model a Transaction

can express typing of parameters, business rules, and preconditions of the called operations. In the following operation, the first three preconditions are for typing, the next is the precondition of the called operation *CreateLoan*, and the last is a business rule, stating that, to make a loan, customers must have fewer existing loans than their limit.

$$
\begin{array}{l}
videoCode \leftarrow LoanByCategory(cu, cat, d) \\
\textbf{PRE} \\
\quad cu \in CUSTOMER \wedge cat \in CAT \wedge d \in DATE \wedge \\
\quad cu \in Customer \wedge \\
\quad card(Loan^{-1}[\{cu\}] < MaxLoan(cu) \\
\textbf{THEN} \\
\quad \textbf{ANY } vc \textbf{ WHERE} \\
\qquad vc \in VideoCopy - dom(Loan) \ \wedge \\
\qquad Category(BelongsTo(vc)) = cat \ \wedge \\
\qquad NumFree(BelongsTo(vc)) \geq 1 \\
\quad \textbf{THEN} \\
\qquad CreateLoan(cu, vc, d) \ || \\
\qquad B\_Change\_NumFree(BelongsTo(vc), (NumFree(BelongsTo(vc)) - 1)) \ || \\
\qquad videoCode := Code(vc) \\
\quad \textbf{END} \\
\textbf{END}
\end{array}
$$

Since the **ANY** substitution is non-deterministic, an additional precondition must be added to ensure the feasibility of the operation:

$$
\begin{array}{l}
\exists\, vc \in VideoCopy - dom(Loan) \bullet \\
\qquad Category(BelongsTo(vc)) = cat \ \wedge \ NumFree(BelongsTo(vc)) \geq 1
\end{array}
$$

ation facilities, whilst a language such as ASL [22] is not suitable, since it is not a declarative specification language.

2.2.3 Transaction modelling discussion

It is well-known that designing different views of a system highlights different aspects and properties of the system. Thus a state diagram is local to a class and shows the effect of events on the different states of class instances. A collaboration diagram shows the internal structure of a transaction. The extended class diagram summarizes the static links that exist between the transactions of one use case and the relevant classes. There is also a correspondence between the two forms of control class and the state and collaboration diagrams. Each B operation generated from an event in the state diagram is added to the relevant *EntityControl* class. For instance, *CreateLoan* and *ReturnLoan* are derived from the state diagram of the association class *Loan* and added in class *LoanControl* in the extended class diagram, Figure 4. Similarly, each B operation generated from an input message of a collaboration diagram is added to the relevant TransactionControl class. Here the input message *MakeLoan* is added to class *LoanManagementUseCaseControl*.

Different kinds of checks can be achieved. For example, integrity constraints are translated into B state invariant conjuncts, thus they generate proof obligations. On a state diagram, the source state of a transition gives the precondition of the corresponding B operation. The target state gives additional postconditions, and thus proof obligations. It can be proved that guards associated with each event are deterministic (only one guard is true at a given time) and that their disjunction is always true. Moreover, model checking can be used, in addition to the B prover, to check liveness properties.

To properly verify a specification, we must demonstrate that each operation maintains the state invariant, and that the state invariant is not always false. The tools associated with the B method automatically generate the required proof obligations and have powerful mechanisms for automatically discharging most of them. Furthermore, research on the B method is continuously improving the automatic proof facilities [2]. For us, it is one of the major advantages of using B. In the IS domain, we have found that, if the B specification respects the characteristics of IS-UML, discharging the proof obligations is not problematic. The proofs that are not discharged automatically follow similar patterns and are not too difficult to discharge by hand. However, proofs can be simplified, and some errors found during proof discharge can be avoided, if the model is verified as far as possible before translation to B.

If we want to have a meaningful IS-UML specification, we have to check the consistency between the different views, for instance that a message in a collaboration diagram is either a generated operation or an event of a state diagram. To prove the consistency conjectures on an IS-UML model, it is essential that the class, state and collaboration diagrams are consistent. The following sections present our approach to this, based on rigorous metamodelling.

3 Formalizing IS-UML metamodels

To deal with the issue of consistency of views, we must have a formal system model. This provides a framework for reasoning about the relationships between the different views. For each kind of diagram, we present a metamodel in the B specification, and a summary in UML class diagrams to highlight links between the concepts. This at least ensures syntactical coherence, and also captures part of the static semantics. Note that in the metamodels, the correspondence between the state space in B and the UML static elements is the same as that defined for IS-UML models.

3.1 Formalizing IS-UML class diagram notations

Our formal metamodel for IS-UML class diagrams, that describe the entity classes and their relationships, is described elsewhere [10,11]. The parts of the class diagram metamodel relevant to the paper are shown in Figure 5. The main metamodel concepts are *Class* and *Association*. An association class is represented by a *Class* instance, an *Association* instance and an instance of the metamodel association *assocClass*. An *Attribute* is a *Characteristic* of a *Class*; an association end, *AssocEnd*, is that of an *Association*. Figure 5 also shows operations, focusing on the elementary database operations that were generated automatically in our B translation, *GeneratedOp*, and its three subclasses. The abstract superclass *Operation* defines the signature of an operation (attributes *opName* and *opParams*). For simplicity, this figure omits the meta-structures of inheritance and static constraints.

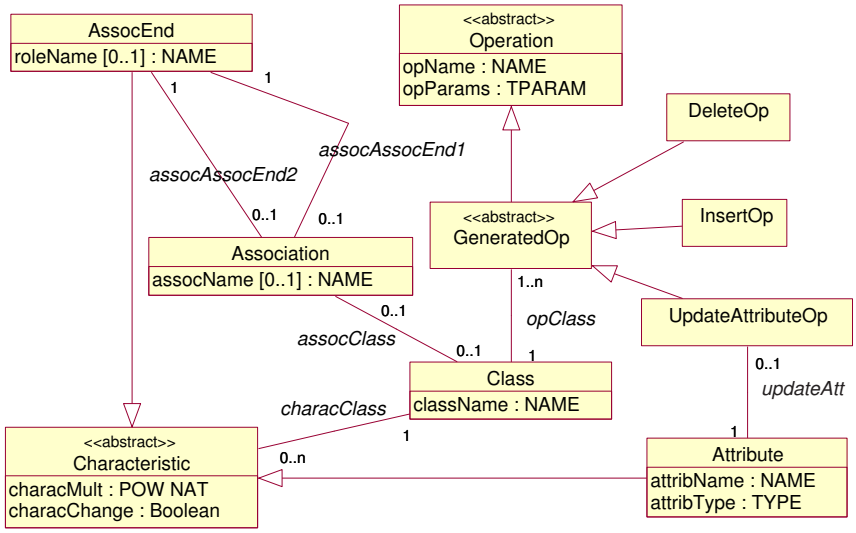

Fig. 5. Extract of the Metamodel for the IS Class Model

The B counterpart of the abstract syntax concepts is described in the following tables. Here, only the relevant clauses are given. The B metamodel requires a number of abstract sets, written in upper case, whereas names of *VARIABLES* correspond to the UML names.

Table 1 gives the typing invariants for each meta-class.

Subclasses have the same type as their superclass; the instances are a subset of the superclass instances. An invariant is added to express the disjointness of subclasses. For example,

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

Table 1
B Variables and Typing Invariants of IS Metamodel Classes

$Class$	$\subseteq$	$CLASS$
$Association$	$\subseteq$	$ASSOCIATION$
$Characteristic$	$\subseteq$	$CHARACTERISTIC$
$Attribute$	$\subseteq$	$Characteristic$
$AssocEnd$	$\subseteq$	$Characteristic$
$Operation$	$\subseteq$	$OPERATION$
$GeneratedOp$	$\subseteq$	$Operation$
$DeleteOp$	$\subseteq$	$GeneratedOp$
$InsertOp$	$\subseteq$	$GeneratedOp$
$UpdateAttributeOp$	$\subseteq$	$GeneratedOp$

Table 2 gives the invariants that express the typing of attributes and their mappings to classes. Mandatory attributes are represented by total functions.

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) [3]:

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

[3] There are many other ways of modelling parameters, including simply allowing a multi-valued attribute of type *TPARAM*. At some point, the type of the logical parameter needs comparing to the type of the actual parameter supplied, which may be, for example, the value of an attribute.

Table 2
B Variables and and Typing Invariants Associating Attributes to IS Metamodel Classes

$className$	$\in$	$Class \rightarrowtail NAME$
$characMult$	$\in$	$Characteristic \rightarrow \mathbb{P}(NAT)$
$characChange$	$\in$	$Characteristic \rightarrow BOOLEAN$
$attribName$	$\in$	$Attribute \rightarrow NAME$
$attribType$	$\in$	$Attribute \rightarrow TYPE$
$roleName$	$\in$	$AssociationEnd \nrightarrow NAME$
$assocName$	$\in$	$Association \nrightarrow NAME$
$opName$	$\in$	$Operation \rightarrow NAME$
$opParams$	$\in$	$Operation \rightarrow seq(TPARAM)$

The abstract set, *TYPE*, represents any relevant UML- and user-defined data types. The B abstract set, *NAME*, represents any name in the metamodel.

Metamodel associations are expressed as relations or functions, table 3.

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

$opClass$	$\in$	$GeneratedOp \twoheadrightarrow Class$
$updateAtt$	$\in$	$UpdateAttribOp \rightarrowtail Attribute$
$characClass$	$\in$	$Characteristic \rightarrow Class$
$assocClass$	$\in$	$Association \nrightarrow Class$
$assocAssocEnd1$	$\in$	$Association \rightarrowtail AssocEnd$
$assocAssocEnd2$	$\in$	$Association \rightarrowtail AssocEnd$

Additional constraints can be defined by B invariants: four of them are illustrated here.

- Each instance of *AssocEnd* must be linked to exactly one instance of *Association*, either by association *assocAssocEnd1* or *assocAssocEnd2*:

$$ran(assocAssocEnd1) \cap ran(assocAssocEnd2) = \varnothing \wedge$$
$$ran(assocAssocEnd1) \cup ran(assocAssocEnd2) = AssocEnd$$

  This means that an instance of *AssocEnd* is either the first or the second end of an association but not both.

- *GeneratedOp* is an abstract class and its subclasses are disjoint :

$$\begin{array}{l} GeneratedOp \;=\; InsertOp \cup DeleteOp \cup UpdateAttributeOp \;\wedge \\ InsertOp \cap DeleteOp \;=\; \varnothing \;\wedge \\ InsertOp \cap UpdateAttributeOp \;=\; \varnothing \;\wedge \\ DeleteOp \cap UpdateAttributeOp \;=\; \varnothing \end{array}$$

- The uniqueness of operation names within a class 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:

$$\begin{array}{l} \forall\, c \;\in\; ran(opClass) \;\bullet \\ \quad \forall\, o_1, o_2 \;\in\; opClass^{-1}[\{c\}] \;\bullet \\ \qquad opName(o_1) = opName(o_2) \Rightarrow o_1 = o_2 \end{array}$$

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

$$\begin{array}{l} \forall\, op \;\in\; UpdateAttribOp \;\bullet \\ \quad updateAtt(op) \in allClassAttributes(opClass(op)) \end{array}$$

where *allClassAttributes()* returns all the attributes of a class.

Figure 6 synthesizes the abstract syntax metamodel of the new concepts and relationships introduced in the extended class diagrams. The stereotyped classes are defined as the two meta-classes *EntityControlClass* and *TransactionControlClass*. These are subclasses of *ControlClass*. The class *ExtendedOp* [4] contains the operations defined in *EntityControl* classes. The class *Transaction* contains the operations defined in all the *TransactionControl* classes. The class *BasicOp* is added to represent all the sorts of operations that act on only one class or association class. Its two subclasses are *GeneratedOp*, defined above, and *ExtendedOp*. The association between *GeneratedOp* and *ExtendedOp* specifies that an extended operation is constructed from generated operations. The association *BaOpClass* links each *BasicOp* instance to the class where it is defined; for a *GeneratedOp* instance, it corresponds to the association *opClass* described in Figure 5.

The association *useVarCl* maps *ControlClass* instances to *Class* instances. This allows the stereotyped dependency link $\ll var \gg$ to be defined. Similarly, the association *ECuseOpCl* models the stereotyped dependency links

[4] Our early work on operation metamodel [11] uses the term *UserOp*. We change it to *ExtendedOp* to avoid confusion with transactions which are also user operations but at the system level.

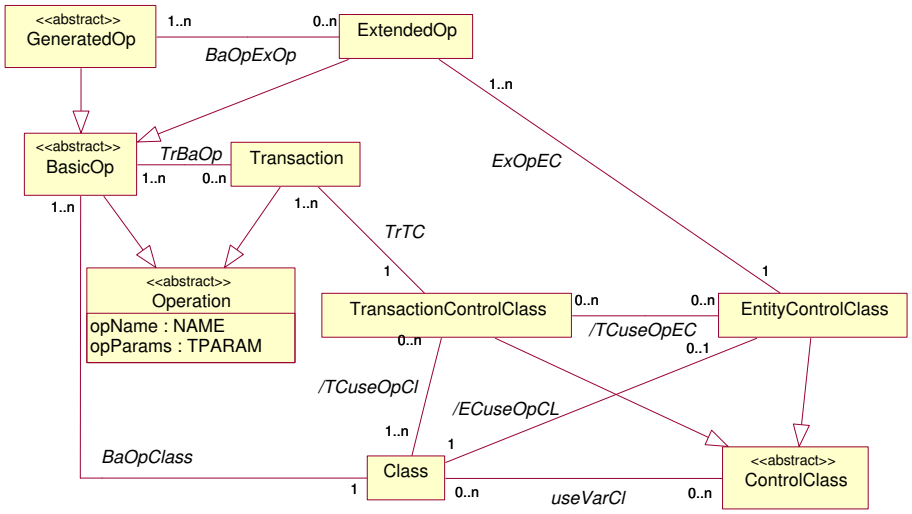

Fig. 6. Metamodel for Extended Class Diagrams

$\ll op \gg$ between *EntityControl Class* and *Class*. The associations *TCuseOpCl* and *TCuseOpEC*, between *TransactionControl Class* and *Class*, and between *TransactionControl Class* and *EntityControl Class* respectively, allow the definition of the $\ll op \gg$ stereotyped dependency links issued by an instance of *TransactionControl Class*. These associations are derived by navigation; this cannot be recorded graphically, but is included in the formal metamodel. Tables 4 and 5 give the formal specification of the new concepts of extended class diagrams.

$Transaction$	$\subseteq$	$Operation$
$ExtendedOp$	$\subseteq$	$BasicOp$
$ControlClass$	$\subseteq$	$CONTROLCLASS$
$EntityControlClass$	$\subseteq$	$ControlClass$
$TransactionControlClass$	$\subseteq$	$ControlClass$

Table 4
B Variables and Invariants for New Classes of Extended Class Diagrams

In table 5, the function representing the derived association is equivalent to the composition of the formal representations of the associations. For example, *ECuseOpCl* is obtained by the navigation from *EntityControlClass* to *ExtendedOp* and from *BasicOp* to *Class*. It is defined as:

$$ECuseOpCl \;==\; ExOpEC^{-1} \fatsemi BaOpClass$$

and we add the constraint:

$TrBaOp$	$\in$	$Transaction \leftrightarrow BasicOp$
		$\wedge\ dom(TrBaOp) = Transaction$
$BaOpExOp$	$\in$	$ExtendedOp \leftrightarrow GeneratedOp$
		$\wedge\ dom(BaOpExOp) = ExtendedOp$
$BaOpClass$	$\in$	$BasicOp \twoheadrightarrow Class$
$useVarCl$	$\in$	$Class \leftrightarrow ControlClass$
$TrTC$	$\in$	$Transaction \twoheadrightarrow TransactionControlClass$
$ExOpEC$	$\in$	$ExtendedOp \twoheadrightarrow EntityControlClass$

Table 5
B Variables and their Typing Invariants Representing New Associations of Extended Class Diagrams

$$ECuseOpCl \in EntityControlClass \rightarrowtail Class$$

This specifies that an *EntityControlClass* instance is linked to exactly one *Class* instance and, reciprocally, a *Class* instance is associated to at most one *EntityControlClass* instance.

Additional constraints express static semantics such as "all the generated operations referenced by an extended operation must belong to the same class".

3.2 Formalizing IS-UML state diagram notations

In UML, the semantics of state and collaboration diagrams is deliberately informal, to allow them to be used to model different kinds of system. Our objective is to define the concepts that are useful for IS specification.

As we discussed in Section 2.2.1, a UML state diagram models the possible states of an object of one class, the permitted transitions, and the events that trigger transitions; the IS-UML metamodel is summarized in Figure 7. The links between the state diagram concepts and the other metamodels are described in Section 4.

A *StateDiagram* is linked to one class. It comprises *Transition*s. A *Transition* may have a guard (*SDguard*) – a boolean expression defining the firing condition of the transition. A *Transition* may be triggered by a named *SEvent*. These events are either time-related (*TimeEvent*) or conventional (*OpEvent*). The latter may convey parameters.

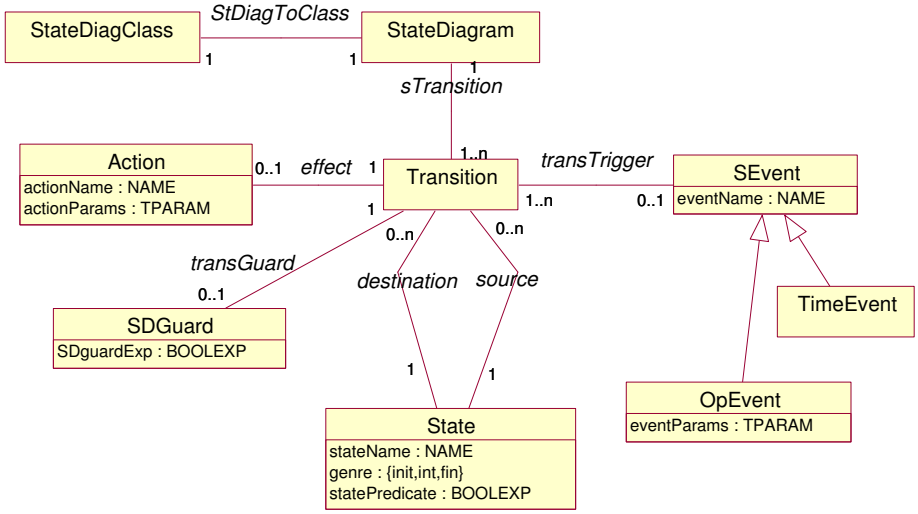

Fig. 7. IS-UML State Diagram Metamodel

Every *Transition* is associated with two instances of *State*, a source and a destination, that can be the same in the case of a reflexive transition. A *State* is named; its *genre* indicates whether it is only an initial state, only a final state, or an intermediate state in the life of an instance of the relevant class. Each *State* is defined by a boolean expression, the *statePredicate*, on the values of the attributes of the class or its associations.

The B representation of the abstract syntax concepts is given in the following tables. Table 6 gives the typing invariants for each meta-class.

$StateDiagClass$	$\subseteq$	$CLASS$
$State$	$\subseteq$	$STATE$
$StateDiagram$	$\subseteq$	$STATEDIAG$
$Transition$	$\subseteq$	$TRANSITION$
$SEvent$	$\subseteq$	$SEVENT$
$OpEvent$	$\subseteq$	$SEvent$
$TimeEvent$	$\subseteq$	$SEvent$
$Action$	$\subseteq$	$ACTION$
$SDGuard$	$\subseteq$	$GUARD$

Table 6
B representation of state diagram metaclasses

The metamodel subclasses require additional exclusivity and coverage con-

straints in B:

$$OpEvent \cup TimeEvent = SEvent$$
$$OpEvent \cap TimeEvent = \varnothing$$

Table 7 gives the invariants that express the typing of attributes and their mappings to classes. Table 8 models the associations in the metamodels.

$stateName$	$\in$	$state \rightarrow NAME$
$genre$	$\in$	$state \rightarrow \{init, int, fin\}$
$statePredicate$	$\in$	$state \rightarrow BOOLEXP$
$SDguardExp$	$\in$	$SDGuard \rightarrow BOOLEXP$
$eventName$	$\in$	$sEvent \rightarrow NAME$
$actionName$	$\in$	$action \rightarrow NAME$
$actionParams$	$\in$	$action \rightarrow seq(TPARAM)$
$eventParams$	$\in$	$opEvent \rightarrow seq(TPARAM)$

Table 7
B invariants for state diagram metamodel attributes

$StDiagToClass$	$\in$	$StateDiagram \rightarrowtail\!\!\!\!\!\!\rightarrow StateDiagClass$
$sTransition$	$\in$	$Transition \rightarrow\!\!\!\!\!\!\rightarrow StateDiagram$
$source$	$\in$	$Transition \rightarrow State$
$destination$	$\in$	$Transition \rightarrow State$
$transTrigger$	$\in$	$Transition \rightarrow\!\!\!\!\!\!\!\rightarrow SEvent$
$effect$	$\in$	$Action \rightarrowtail Transition$
$transGuard$	$\in$	$SDGuard \rightarrowtail Transition$

Table 8
B representation of state diagram metamodel associations

The full metamodel includes B invariants capturing a wide variety of constraints and semantic details. Examples of the different kinds of constraints are as follows.

Unique names

UML requires various concepts to have unique names. In state diagrams, we illustrate this for state names. Within a state diagram *sd*, the values of

stateName are distinct (though they need not be distinct between diagrams):

$$\forall\, t \in sTransition^{-1}[\{sd\}] \bullet \\ \quad \forall\, s_1, s_2 \in source[\{t\}] \cup destination[\{t\}] \bullet \\ \quad\quad stateName(s_1) = stateName(s_2) \Rightarrow s_1 = s_2$$

Constraints on transitions

- a state is either a source or a destination of a transition, and can be both:

  $$State = ran(source) \cup ran(destination)$$

- there is always something associated to a transition: a guard or an event or an action:

  $$Transition = ran(transGuard) \cup dom(transTrigger) \cup ran(effect)$$

- if there is an action associated to a transition, then there is also an event.

  $$ran(effect) \subseteq dom(transTrigger)$$

The reason for the last constraint is technical: we need an event to give a name to the generated B operation that corresponds to the transition. Note that there are other transitions that have only a guard. They are generally called automatic transitions. For example, customers who have exceeded their loan date move automatically to a state "lateCustomer". In this case, there is no B operation generated for the transition.

Constraints on transitions associated to the same event

An *OpEvent* can trigger several transitions, depending on the source state and/or the guard. However all the transitions must belong to the same state diagram:

$$\forall\, ev \in OpEvent \bullet card(sTransition \fatsemi transTrigger^{-1}[\{ev\}]) = 1$$

Constraints on parameters

Actions and events ultimately provide the input parameters of operations, so cannot be output parameters on the diagram:

$$\forall\, n \in NAME \bullet \forall\, t \in TYPE \bullet \\ \quad \forall\, r \in \mathrm{ran}(actionParams) \bullet r \neq (n, t, o)$$

This means that the third element of the parameters cannot be "o".

A UML collaboration diagram describes how different objects interact, by exchanging messages, to complete a task. In the database part of an IS, there is no message passing (among tables); the task is achieved as a transaction under the control of a DBMS transaction manager. For IS-UML, we have modified collaboration diagrams to specify transactions from a use case. The diagram shows the classes of objects involved, and which operations are called under what conditions. The metamodel (Figure 8) simply comprises the messages that make up the diagram.

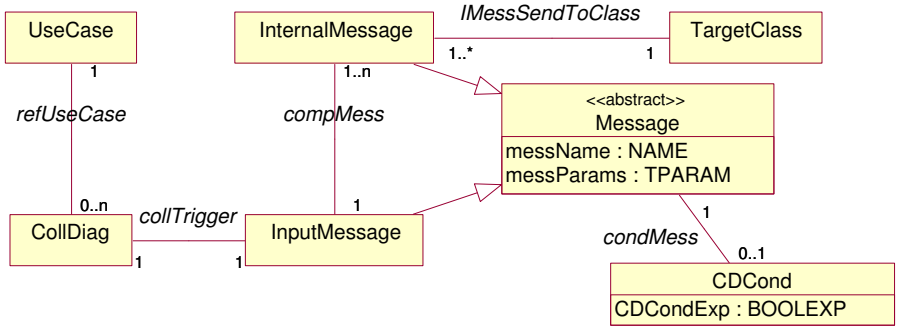

Fig. 8. IS-UML Collaboration Diagram Metamodel

The IS-UML collaboration metamodel has two subclasses of *Message*. An *InputMessage* is the triggering message of a collaboration diagram. An *InternalMessage* is triggered by an *InputMessage* and refers to a particular *TargetClass*. A *Message* can have an associated condition that is a boolean expression. Each *Message* is defined with a name and a parameter list. Tables 9, 10 and 11 give the B specification.

$UseCase$	$\subseteq$	$USECASE$
$CollDiag$	$\subseteq$	$COLLDIAG$
$Message$	$\subseteq$	$MESSAGE$
$InputMess$	$\subseteq$	$Message$
$InternalMess$	$\subseteq$	$Message$
$CDCond$	$\subseteq$	$CDCOND$
$TargetClass$	$\subseteq$	$CLASS$

Table 9
B representation of collaboration diagram meta-classes

The only specific constraints in a collaboration diagram express the uniqueness of message names inside a collaboration diagram, the kind of message

$messName$	$\in$	$Message \rightarrow NAME$
$messParams$	$\in$	$Message \rightarrow seq(TPARAM)$
$CDCondExp$	$\in$	$CDCond \rightarrow BOOLEXP$

Table 10
B invariants for collaboration diagram metamodel attributes

$refUseCase$	$\in$	$CollDiag \rightarrow UseCase$
$collTrigger$	$\in$	$InputMess \rightarrowtail\!\!\!\!\!\rightarrow CollDiag$
$compMess$	$\in$	$InternalMessage \rightarrow\!\!\!\!\!\!\rightarrow inputMessage$
$condMess$	$\in$	$CDCond \rightarrowtail Message$
$IMessSendToClass$	$\in$	$InternalMessage \rightarrow\!\!\!\!\!\!\rightarrow TargetClass$

Table 11
B representation of collaboration diagram metamodel associations

parameters (as in a state diagram) and constraints defined on *Message* subclasses:

$$InternalMessage \cup InputMessage = Message$$
$$InternalMessage \cap InputMessage = \varnothing$$

4 A consistent global view of an IS

Having outlined our modelling approach and the formally underpinned metamodels that allow us to check individual diagram consistency, we now explore what can be elaborated on a whole "IS-UML specification". There are two kinds of issues: consistency checks between metamodels and mapping rules between specific concepts of metamodels.

4.1 Consistency among metamodels

Using the formal metamodels, there is a number of consistency checks among the metamodels that can be carried out.

Syntactic unification is achieved by linking corresponding concepts in the metamodels. For instance, each state diagram relates to a class that must exist in the class diagram.

$$StDiagToClass \subseteq Class$$

That is, *StateDiagClass* is a subclass of *Class*.

In the same way, the target class of an internal message in a collaboration diagram must exist in the class diagram.

$$TargetClass \subseteq Class$$

The main links among the metamodels are summarized in Figure 9, by way of the definition of derived associations, *ActRefOp*, *IMessRefGenOp*, *IMessRefOpEv*.

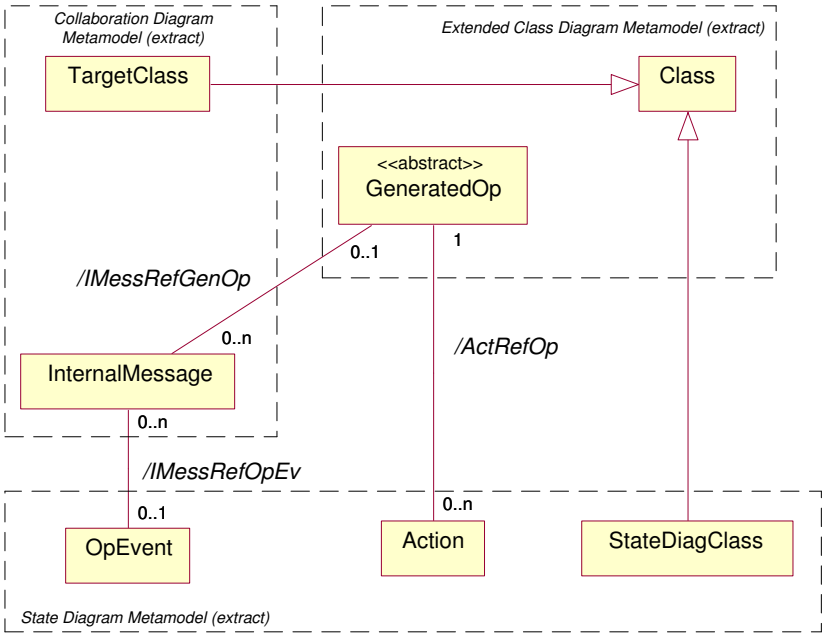

Fig. 9. Consistency Links between the Metamodels

An action must correspond to a generated operation, in that it is a call to such an operation. This constraint can be expressed by defining the derived association *ActRefOp*. Note that for the moment, we just consider name correspondence and do not take account of parameters.

$$\begin{aligned}
&ActRefOp \in Action \rightarrow GeneratedOp \wedge \\
&ActRefOp = \{ac \mapsto \ genop \mid ac \in Action \\
&\quad \wedge\ genop \in effect \fatsemi sTransition \fatsemi StDiagramToClass \fatsemi opClass^{-1}[\{ac\}] \\
&\quad \wedge\ actionName(ac) = opName(genop)\}
\end{aligned}$$

An internal message of a collaboration diagram refers to a particular class and concerns either a direct call to a generated operation of the class, or an event

of the state diagram of the class (if it exists). As previously, this is expressed by defining the two derived associations *IMessRefGenOp* and *IMessRefOpEv*:

$$
\begin{array}{l}
IMessRefGenOp \in InternalMessage \nrightarrow GeneratedOp \wedge \\
IMessRefGenOp = \{im \mapsto genop \mid im \in InternalMessage \\
\quad \wedge\ genop \in IMessSendToClass \fatsemi opClass^{-1}[\{im\}] \\
\quad \wedge\ messageName(im) = opName(genop)\}
\end{array}
$$

$$
\begin{array}{l}
IMessRefOpEv \in InternalMessage \nrightarrow OpEvent \wedge \\
IMessRefOpEv = \{im \mapsto ev \mid im \in InternalMessage \\
\quad \wedge\ ev \in IMessSendToClass \fatsemi StDiagToClass^{-1} \fatsemi \\
\qquad sTransition^{-1} \fatsemi transTrigger[\{im\}] \\
\quad \wedge\ messageName(im) = eventName(ev)\}
\end{array}
$$

We need to express that these functions are total and exclusive because an *InternalMessage* is either the concern of a *GeneratedOp* or of an *OpEvent* of a state diagram, but not both:

$$
\begin{array}{l}
\mathrm{dom}(IMessRefGenOp) \cap \mathrm{dom}(IMessRefGenOp) = \varnothing \wedge \\
\mathrm{dom}(IMessRefGenOp) \cup \mathrm{dom}(IMessRefGenOp) = InternalMessage
\end{array}
$$

One area of potential checking that we have not fully explored is the expression of guards, conditions and static constraints. In the metamodels, any type that is a predicate is expressed using the general *BOOLEXP*. We could express further constraints if we used a more complex type for these predicates.

4.2 Mapping rules between metamodels

Semantic unification occurs, owing to the fact that the same translation into B is used on one hand for events in state diagrams and input messages of collaboration diagrams and, on the other hand, for operations of *EntityControl* classes or *TransactionControl* classes. Figure 10 illustrates the mapping rules: two new associations are defined formally in Table 12. Each event of a state diagram leads to the creation of an extended operation (association *OpEventExtOp*) in the *EntityControl* class linked to the class where the state diagram of the event is defined. Similarly each input message of a collaboration diagram leads to the creation of a transaction (association *InputMessTransaction*) in the *TransactionControl* class that corresponds to the *UseCase*.

The two other associations of Fig. 10 are derived associations, with additional constraints.

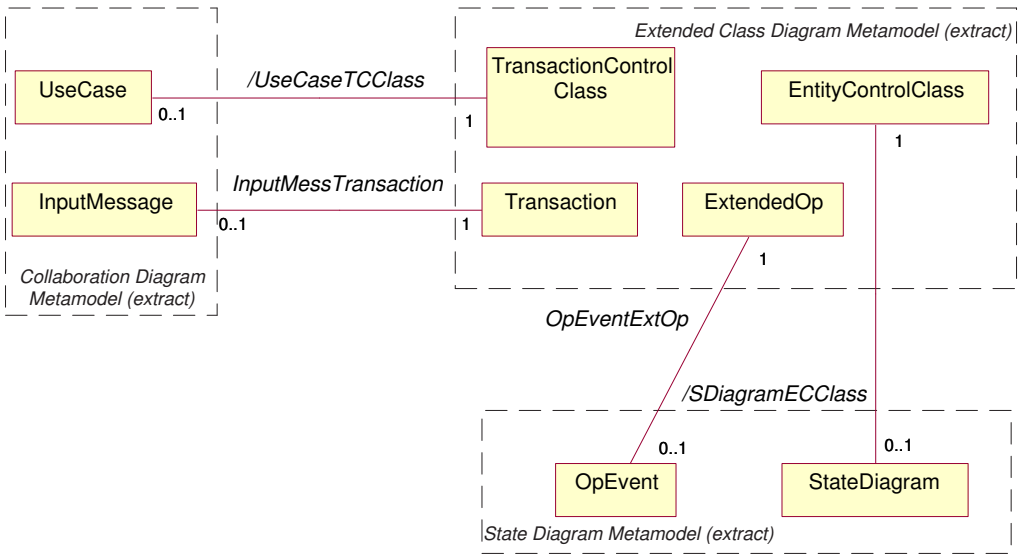

Fig. 10. Mapping Rules between the Metamodels

$OpEventExtOp$	$\in$	$OpEvent \rightarrowtail ExtendedOp$
$InputMessTransaction$	$\in$	$InputMessage \rightarrowtail Transaction$

Table 12
B representation of the mapping rules

$$SDiagramECClass == sTransition^{-1} \fatsemi transTrigger^{-1} \fatsemi OpEventExtOp \fatsemi ExOpEc$$

The following constraint states that each state diagram is necessarily associated with an entity control class and, reciprocally, an entity control class is associated with at most one state diagram:

$$SDiagramECClass \in StateDiagram \rightarrowtail EntityControlClass$$

The same format is used for $UseCaseTCClass$:

$$UseCaseTCClass == refUseCase^{-1} \fatsemi collTrigger^{-1} \fatsemi InputMessTransaction \fatsemi TrTC$$

with the associated constraint:

$$UseCaseTCClass \in UseCAse \rightarrowtail TransactionControlClass$$

Let us detail one of the mapping rule. This is described by a B operation, in Fig. 11. Bracketed numbers refer to the numbered lines of Fig. 11. Let *sd* be the state diagram associated to an event *ev*:

$$sd == \mathbf{CHOICE}(transTrigger^{-1} \fatsemi sTransition[\{ev\}])$$

The **CHOICE** operator applies on a non-empty set, and returns an arbitrary element of the set. Here the set $transTrigger^{-1} \fatsemi sTransition[\{ev\}]$ has only one element, the state diagram of *ev*, because of the constraint that all the transitions associated to one event belong to the same state diagram (see page 26).

For each event *ev*, an extended operation *newOp* is created in the relevant *EntityControlClass* instance. The precondition, line (1), of the operation *MapEvToOp* specifies that the name of the event cannot be the name of an existing operation of the class associated to *sd* because in a class, all operations must have distinct names and we create the operation *newop* with the name of the event *ev*. The body of the operation *MapEvToOp* consists of:

- creating the new operation *op*, lines (2) and (3), and setting its attributes *opName* and *opParams*, lines (4) and (5).
- setting its associations *BaOpExOp* and *BaOpClass* (see Fig. 6). The first, line (6), links an extended operation to the generated operations it calls. For an extended operation corresponding to an event *ev*, these operations are retrieved by taking the generated operation associated to the action defined on each transition triggered by the event:

  $$transTrigger^{-1} \fatsemi effect^{-1} \fatsemi ActRefOp[\{ev\}]$$

  BaOpClass links a basic operation to the class where it is defined. In this case, line (7), the class is that associated with the state diagram *sd* of *ev*, that is *StDiagToClass*(*sd*).
- linking the new operation *op* to the relevant *EntityControlClass* instance, if it exists, line (9). Otherwise we need to create this class, lines (10) to (14). To know if this class exists, we use the association *ECuseOpCl* by testing whether *StDiagToClass*(*sd*) is linked to an *EntityControlClass* instance, line (8).
- linking *op* and *ev* with the association *OpEventExtOp*, line (15).

5 Conclusions and further work

Our work on IS-UML provides a specialization of UML for IS development. The formal metamodel, and the ability to automatically translate models into

$newOp \leftarrow MapEvToOp(ev)$
PRE
$ev \in OpEvent \ \wedge$
(1) $eventName(ev) \notin StDiagToClass \fatsemi BaOpClass^{-1} \fatsemi opName[\{sd\}] \ \wedge$
$operation \subset OPERATION$
THEN
(2) **ANY** op **WHERE** $op \in OPERATION - operation$
THEN
(3) $ExtendedOp := ExtendedOp \ \cup \ \{op\} \ ||$
(4) $opName(op) := evName(ev) \ ||$
(5) $opParams(op) := evParams(ev) \ ||$
$BaOpExOp := BaOpExOp \cup$
(6) $\{op\} \times (transTrigger^{-1} \fatsemi effect^{-1} \fatsemi ActRefOp[\{ev\}]) \ ||$
(7) $BaOpClass(op) := StDiagToClass(sd) \ ||$
(8) **IF** $StDiagToClass(sd) \in ran(ECuseOpCl)$
THEN
(9) $ExOpEC(op) := StDiagToClass \fatsemi ECuseOpCl^{-1}(sd)$
ELSE
(10) **ANY** $newec$ **WHERE**
(11) $newec \in CONTROLCLASS - ControlClass$
THEN
(12) $EntityControlClass := EntityControlClass \cup \{newec\} \ ||$
(13) $name(newec) := name(StDiagToClass(sd)) \ + \ 'Control' \ ||$
(14) $ExOpEC(op) := newec$
END
END $||$
(15) $OpEventExtOp(ev) := op \ ||$
(16) $newOp := op$
END
END

Fig. 11. Mapping an Event to an Extended Operation

B gives the ability to rigorously check the conformance of models to the meta-model, the internal consistency of models, consistency across class, state and collaboration diagrams, and the ability of the modelled system to maintain data integrity.

Whilst other research has applied formal definition and translation to E/R diagrams or class models [6,7,5,14,18,27,26], there is little previous work on formal modelling of the functional aspects, particularly for IS. The adaptation of conventional UML diagrams for IS is essential, since most IS still use relational, rather than object-oriented, database technology. It is essential to model IS transactions, since these are the primary functional concept of IS,

and are responsible for establishing and maintaining data integrity. In addition, for most IS applications, transactions are important for security control (access control, authentication, etc). It is thus not sufficient to only model the low-level operations on database tables. To model the internal structure of transactions, we must model the control aspect, since there are no objects to call the methods of other objects.

We use B in our work on IS-UML, but recognize that the same approach can be used with any model-based formal language. The main advantage of B are the relatively-advanced tool support for proof. Most commercial work with B is for critical systems. For IS, the necessary consistency proofs are comparatively straightforward, and are automatically discharged by tools such as Atelier B [4]. Furthermore, the proofs follow a general form which allows reuse of proof tactics [15]. Most aspects of the B model can thus be generated and proved automatically from a set of UML diagrams that conforms to the IS-UML metamodel. For this reason, we believe that our approach can be incorporated into practical IS modelling and development.

In our approach, there is a direct mapping between the concepts in the B models and the structure of the IS-UML diagrams. This assists in traceability (see [12]). Other research projects combining UML and B find similar advantage in the use of B to add rigour to UML. However, none of the other work is dedicated to IS. Our focus on IS simplifies the metamodel and formal modelling, but, as a result, the semantics given by translation rules is necessarily different from that of, for example, UML-B [26], which is developed for more general systems modelling.

Other rigorous approaches to UML only apply consistency checking to the formal models (by using support tools), not to diagrams. In IS-UML, we have a formally-based metamodel that allows us to apply a range of consistency checks to the diagrammatic models before translation to B.

For the translation to B, in addition to conformance to the IS-UML metamodel, it is currently necessary to express logical expressions associated with model constructs (conditions, guards etc.) as B predicates. However, if the IS-UML approach were extended to permit OCL annotations, and to automatically translate these to B predicates, then the developers could construct models entirely in the familiar diagrammatic medium, before automatic translation to B for formal analysis. A consequence of this extension would be that IS-UML could be completely aligned with current initiatives in object-oriented development (model driven development, model management, etc [20]), whilst continuing to provide rigorous analysis and proof of consistency.

Our work includes a prototype tool that takes IS-UML diagrams and generates the B models needed for rigorous analysis [17]. Up to now, the diagrams

are produced using the Rose environment [24]; annotations that cannot be expressed graphically are added in the documentation field of the relevant element. Rose is no longer the best UML tool available; we need to generalize tool support for modern UML tools, exploiting the serialized output formats provided by commercial tools (eg. in XMI). Ideally, we need to be able to customize the modelling tool to impose the necessary features of IS-UML on the models, rather than having to apply checks of conformance to the metamodel after diagram construction. Again, alignment with current practice in metamodelling tools would allow us to exploit model management techniques (model comparison, model merging, model transformation) that are under development (see EU Modelware project publications [19]).

Although most of our work has been on the formal IS-UML metamodel, we are also looking at other ways of analysing the metamodel. For example, with Paige, we are developing an executable metamodel using Eiffel that can be analysed by testing the metamodel program, following the method of [23].

Immediate plans for IS-UML include at least the following.

- Complete the behavioural aspects of IS-UML — for example, we need to add behaviours in relation to inheritance and composition, to the limited extent that these apply to IS.
- Further facilitate proofs of model consistency by exploring an intermediate level of consistency checking, to verify that the predicates used in the different diagrams are satisfiable (ie. that they are not always false). This is not essential, but if such checks are not applied, then the B proof tool might generate a proof obligation that is always false and thus cannot be automatically discharged; the cause of such a problem is not easily traceable from the output of the B proof tools.

References

[1] J-R. Abrial, *The B-Book: Assigning programs to meanings*, Cambridge University Press, Cambridge, 1996.

[2] J-R. Abrial and D. Cansell, Click'n Prove: Interactive Proofs within Set Theory, In TPHOLs, Roma, Italy, Volume 2758 of LNCS, Springer-Verlag, 2003.

[3] F. Badeau and A. Amelot, Using B as a High Level Programming Language in an Industrial Project: Roissy VAL, In ZB 2005: Formal Specification and Development in Z and B: 4th International Conference of B and Z Users, Guildford, UK, Volume 3455 of LNCS, Springer-Verlag, 2005.

[4] Clearsy, Atelier B, `http://www.atelierb.societe.com`.

[5] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud, An Overview of RoZ: A Tool for Integrating UML and Z Specifications, In 12th International Conference Advanced Information Systems Engineering (CAiSE'00), Volume 1789 of LNCS, Springer-Verlag, 2000.

[6] A. Hall, Using Z as a Specification Calculus for Object-Oriented Systems, In VDM'90, 3rd International Conference, Kiel, Germany, Volume 428 of LNCS. Springer-Verlag, 1990.

[7] S.K. Kim and D. Carrington, Formalizing the UML class diagram using OBJECT-Z, In UML99, Volume 1723 of LNCS, Springer-Verlag, 1999.

[8] R. Laleau, On the interest of combining UML with the B formal method for the specification of database applications, In ICEIS2000: 2nd International Conference on Enterprise Information Systems, Stafford, UK, July 2000.

[9] 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.

[10] R. Laleau and F. Polack, 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.

[11] R. Laleau and F. Polack, Specification of integrity-preserving operations in information systems by using a formal UML- based language, Information and Software Technology, 43:693–704, 2001.

[12] R. Laleau and F. Polack, Coming and going from UML to B: a proposal to support traceability in rigorous IS development, In ZB2002: Formal Specification and Development in B and Z, 2nd International Conference of B and Z Users, Grenoble, France, Volume 2272 of LNCS, Springer-Verlag, January 2002.

[13] R. Laleau, Conception et développement formels d'applications bases de données, Habilitation Thesis, CEDRIC Laboratory, France, 2002, Available at www.univ-paris12.fr/lacl/laleau/.

[14] Ledang, H. and J. Souquieres, Modeling Class Operations in B: Application to UML Behavioral Diagrams, In The Sixteenth IEEE International Conference on Automated Software Engineering (ASE'01), IEEE Computer Society, 2001.

[15] A. Mammar and R. Laleau, Design of an Automatic Prover Dedicated to the Refinement of Database Applications, In FM03: the 12th International FME Symposium, Pisa, Italy, Volume 2805 of LNCS, Springer-Verlag, September 8-14, 2003.

[16] A. Mammar and R. Laleau, From a B formal specification to an executable code: application to the relational database domain, Information and Software Technology, Volume 48:4, April 2006.

[17] A. Mammar and R. Laleau, UB2SQL: A Tool for Building Database Applications Using UML and B Formal Method, Journal of Database Management, 2006 (to be published).

[18] R. Marcano and N. Levy, Transformation rules of OCL Constraints into B Formal Expressions, In J. Jurjens, M. V. Cengarle, E. B. Fernandez, B. Rumpe, and R. Sandner (eds.): Critical Systems Development with UML, Proceedings of the UML'02 workshop. pp. 155-162, Technische Universitat Munchen, Institut fur Informatik, 2002.

[19] Modelware: http://www.modelware-ist.org/index.php

[20] MOF Core Specification 2.0 http://www.omg.org/cgi-bin/apps/doc?formal/06-01-01.pdf, January 2006.

[21] H. P. Nguyen, Dérivation de spécifications formelles B à partir de spécifications semi–formelles, PhD thesis, CEDRIC Laboratory, CNAM, Dec 1998, Available at www.univ-paris12.fr/lacl/laleau/.

[22] Object Management Group, Action Semantics for the UML, doc ad/2001-08-04, Available at http://www.kabira.com/as/.

[23] R.F. Paige, P.J. Brooke, and J.S. Ostroff, Agile Development of a Metamodel in Eiffel, In Fifteenth IEEE International Symposium on Software Reliability Engineering, St-Malo, France, November 2004.

[24] Rational Rose, http://www.rational.com, 2003.

[25] J. Rumbaugh, I. Jacobson, and G. Booch, *The UML Reference Manual*, Addison-Wesley, 1999.

[26] C. Snook, and M. Butler, UML-B: Formal modelling and design aided by UML, ACM Transactions on Software Engineering and Methodology, 2006 (to be published).

[27] H. Treharne, Supplementing a UML development process with B, In FME02: Formal Methods Europe, Copenhagen, Denmark, Volume 2391 of LNCS, Springer-Verlag, July 2002.

[28] J. Warmer, and A. Kleppe. *The Object Constraint Language (Second Edition) - Getting your models ready for MDA*. Addison-Wesley, 2003.

Appendix A: B notations

A B **MACHINE** comprises a number of (optional) clauses. Those used in this paper are as follows. The details are taken from the B language and method definition [1].

MACHINE
introduces the name of the machine
SETS
introduces the name of given sets — we use deferred sets, in which details are omitted; such sets are implicitly finite and non-empty
VARIABLES
introduces the machine components, the identifiers of its variables
INVARIANT
Conjoined predicates that constrain the machine variables — there must be a predicate to define the type of each variable in the **VARIABLES** *clause, in the form,* variable ∈ SET *; there may be further constraints.*
DEFINITIONS
Macro statements for use in the other machine clauses
OPERATIONS
Introduces all the operations of this machine
OperationName(parameter_list)
PRE
The precondition of the operation — conjoined predicates, including (at least) a predicate to define the type of each parameter
THEN
Specification of the operation — a parallel series of substitutions of the form
variable := expression || ...
END
To terminate an operation
END
To terminate the machine specification

In B, there are separate notational conventions for specifications and refinements; all the notations used here are for specifications. The notations used for operation substitutions in this paper are as follows.

OperationName(parameter_list)
PRE
predicates
THEN
SELECT P **THEN** S
WHEN Q **THEN** T
END
Bounded choice: P and Q are predicates, and S and T are the substitutions that occur when the predicates are true.

OperationName(parameter_list)
PRE
predicates

THEN
ANY z **WHERE** P
THEN S
END
Unbounded choice, allowing an arbitrary instance to be selected: z is any instance of a variable (that does not occur in the PRE clause); P is a predicate constraining the instance of z, and S is the substitution that occurs on the selected variable.

In B, large machines are constructed using smaller machines through various access links. A machine *Ma* that uses another machine *Mb* (link **USES**) can read the variables of *Mb* but not modify them. A machine *Ma* may include another machine *Mb* (link **INCLUDES**): the variables of *Mb* can be read in *Ma* but modified only by using the operations of *Mb*.

Appendix B: Translation of associations into B

The following table is a summary of the translation of UML associations into binary relations available in B.

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

Table 13
Example Mappings Modelled by Relation and Functions