Proving the Refinement of Scenarios into Object-Oriented Models*

Rapport technique n° 272
Département de mathématiques et d'informatique
Université de Sherbrooke, Québec, Canada

Rapport technique CEDRIC n° 277
Institut d'Informatique d'Entreprise
Conservatoire National des Arts et Métiers, Évry, France

M. Frappier†
Département de mathématiques
et d'informatique
Université de Sherbrooke, Sherbrooke,
Québec, Canada, J1K 2R1
+1 819 821-8000x2096
Marc.Frappier@dmi.usherb.ca

R. Laleau
Laboratoire CEDRIC
Institut d'Informatique d'Entreprise
Conservatoire National des Arts et Métiers
18 allée Jean Rostand
91025 Évry Cedex France
+33 1 69 36 73 47
laleau@iie.cnam.fr

August 22, 2001

Abstract

We propose a strategy to prove that an analysis model is correct with respect to a requirements model within the context of a UML-based software development process. Both models are described using graphical notations (UML), in order to be end-user understandable, and formal notations (EB^3 and B), in order to write complete and precise specifications. The proof is conducted using the B refinement theory. We illustrate how object-oriented models may be generated from formal scenario specifications. The requirements models (scenarios) consists of entities, defined by a process algebra expression. An entity describes the sequences of events exchanged with the environment. Axioms on entity traces define the outputs of input sequences. The analysis model is expressed in terms of classes and operations modifying objects, their attributes and their associations. The correctness proof ensures that the analysis models implements the behavior described in the requirements model in terms of traces.

Keywords

Trace-based specifications, object-oriented specifications, refinement, UML, B, process algebra.

*This paper is dedicated to the memory of Philippe Facon, our friend and colleague, who contributed to this work. The research described in this paper was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

†Part of this research was conducted while Marc Frappier was visiting the Institut d'Informatique d'Entreprise, Conservatoire National des Arts et Métiers, Évry, France

Contents

1 Introduction

The UML notation is very attractive and intuitive for describing software structure and software behavior in object-oriented software development. However, UML falls short of providing means for *precise* descriptions of software behavior. UML has a *formal syntax*, which means that a document can be checked for conformance with the UML syntax rules. However, UML still lacks a *formal semantics*, which means that there is no way of verifying that an implementation satisfies a UML specification, or to prove that two UML specifications are equivalent, or that one refines the other, or that they are consistent, etc. The lack of a formal semantics may lead to serious communication problems between members of a software development team and with the end users, as each one can have its own personal understanding of the meaning of a UML diagram.

In this paper, we address some shortcomings of UML by strengthening it with a formal specification notation (one which has a formal syntax *and* a formal semantics). This extended version of UML becomes a precise, intuitive, diagrammatic representation of the formal specification notation. In turn, we may say that UML strengthens the formal notation, because it makes specifications easier to grasp and visualize. As noted in [SN01], based on interviews with formal methods practitioners, the use of informal techniques like UML facilitates the transition from informal user requirements to a formal specification. Our extension of UML allows to prove properties about UML documents.

1.1 Scope

This paper addresses the *requirements capture phase* and the *analysis phase* of the software development process.

In UML, the following documents are used for requirements capture. The *use case diagram* describes the functional requirements of the system and define the limits of the analyzed system and the scenarios of events exchanged between the system and the external actors (man or process). It is usually written in natural language so that it can be understood by non experts. Use cases should abstract from internal design issues. The *sequence diagram* describes event sequences in a temporal view in terms of interactions between objects. For data-intensive applications, a *requirements class diagram* (also called a *business model* in UML) may be defined. It captures the most important types of objects in the context of the system. We refer to these diagrams as the *requirements model*.

The purpose of the analysis phase is to abstractly define the internal structure of a software fulfilling the requirements. The structure is defined in terms of objects and operations modifying the objects in response to external events. The following documents are used. The *analysis class diagram* describes the static structure of the system with classes and associations. The *collaboration diagram* specifies how objects collaborate to perform scenarios. The *statechart diagram* describes the local behaviour of objects. These diagrams are referred to as the *analysis model*.

1.2 Extending UML with Formal Methods

We use two *orthogonal* formal specification notations to extend UML. For the requirements model, we use the EB^3 method (Entity-Based Black Box Specification [FSD99a, FSD99b]) in conjunction with the use case diagram and the requirements class diagram. An EB^3 specification replaces sequence diagrams. It allows a complete description of the system behavior in terms of *traces* of events. The input event traces are defined using a process algebra derived from CCS [Mil89], CSP [Hoa85] and LOTOS [BB87]. The relationship between input events and output events is defined using first-order formula relating an input event trace to an output event. An EB^3 specification is executable; hence a specification can be validated with end users using simulation. Model checking can also be applied to verify properties about the specification.

For the analysis model, we use a restricted version of the UML diagrams for which a rigorous definition has been specified [FLN+99, LM00b, LP01b]. The formal semantics of the UML-based language relies on the semantics of the B language [Abr96] and a set of rules that translate diagrammatic notations into a B specification. Note that denotational semantics have also been proposed for the graphical notations that describe class diagrams [LP01a], but this approach has not been retained for the description of behavioral notations because it is much more complex than the chosen approach. We use the B method because it is a complete formal development method that allows proofs of properties to be achieved and refinement to a

provably correct code. As B provides a model-based (state-based) specification language, it is well adapted for the development steps that describe the states and the transitions of the system, thus we use it during the last steps of the development process (from analysis up to implementation). The B method is supported by industrial strength case tools (Atelier B [ClearSy] and the B tool [B-core]) which provide syntax checkers, theorem provers and animators. The B method scales up to industrial size problems: it has been successfully used on several large industrial projects in France and in the UK in various domains like transportation and smart cards [B98]. It is one of the most mature formal methods. Figure 1 contrasts the traditional UML development process with our proposed formal UML development process. Furthermore, using formal notations allows us to check the correctness of the analysis model with respect to the requirements model.

1.3 Applicability

Our framework is designed for data-intensive information systems where the ordering of events is complex. In the past, the reliability requirements for information systems where not as strong, as they were used in-house only. Faults did not usually have a significant impact on an organisation. With the rapid deployment of the world wide web, organisations are developing web access to their information systems, thereby increasing the need for high quality systems. Data integrity becomes a critical issue if ordinary clients can use an organisation's information systems.

1.4 Results and Justification

We can prove the correctness of the analysis model with respect to the requirements model by proving that the analysis model is a *refinement* of the requirements model. To do so, the EB^3 specification is translated (automatically) into a B specification. Then, we prove that the B specification of the analysis model is a refinement of the B specification of the requirements model. Figure 2 illustrates the verification process.

Using two formal notations provides a more complete framework. On one hand, trace-based specifications are quite powerful for representing the behavior in an explicit manner. In addition, they are more resilient to requirements evolution than state-based specification, which are closer to the implementation. State-based specifications are more difficult to validate from a dynamic view-point: visualizing sequences of events is not easy, since transitions are defined by modification of the state variables. The EB^3 requirements model can be seen as a specification of dynamic properties (i.e., properties about sequences of events) for state-based specifications; the refinement proof is the verification of these properties. On the other hand, state-based specifications, like B specifications, are well adapted to give an abstract view of an information system. For such systems, we need a conceptual model for representing the static part of a system (i.e., the data) the numerous links between them, and the integrity constraints that they are subject to. Traditionally, they are described using entity-relationship models (or variants of) whose underlying theoretical foundations are similar to those of the B notation. In addition, the B method provides a complete framework to describe both the data and the operations that act upon them and the possibility to prove the correctness of the operations with respect to the static elements. We can use the B refinement process to produce database implementations [LM00a].

In summary, the main results of this work are as follows. First, it integrates semi-formal and formal methods to take advantage of their strengths: intuitive diagrammatic notation for the former; formality and provability for the latter. Second, it proposes a proof strategy to show the refinement of a trace-based requirements model by a state-based analysis model.

1.5 Related Work

The issue of integrating UML or other informal methods (e.g., OMT, SA) with formal methods has been studied by several groups. We are mainly interested in approaches that relate to information systems and database development. The precise UML group has investigated the formalization of UML [EK99]. So far, a precise semantics has been given to the class diagram using, among others, the Z notation. Neither the formalisation of other UML diagrams nor the correctness/consistency relationship between them has been addressed in detail so far. Some UML diagrams (class, object and statecharts) have also been formalized using the ASM (Abstract State Machine) notation [SCH01, CR01]; these diagrams can be simulated and checked

Software Development Phases	Classical UML	UML + Formal Description
Requirements Capture	*Requirements Model* Uses Cases Requirements Class Diagram Sequence Diagrams	*Formal Requirements Model* Use Case Requirements Class Diagram EB^3 Specification
Analysis	*Analysis Model* Analysis Class Diagram Statechart Diagrams Collaboration Diagrams	*Formal Analysis Model* Analysis Class Diagram Statechart Diagrams with formal annotations Collaboration Diagrams with formal annotations

Figure 1: The software development process

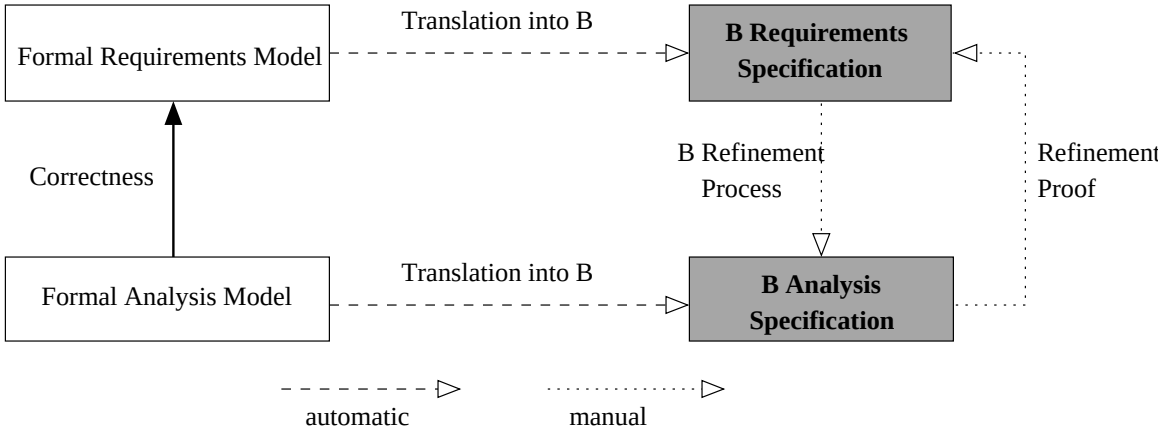

Figure 2: The formal software development process

for consistency. For early approaches, see [SFD92] and the Fusion method [CAB94]. The SAZ method integrates Z with the structured analysis notation [MP95, PWM93]. The work of Dupuy also addresses the derivation of a Z specification from OMT or UML models [DLC00]. Both approaches do not allow reasoning about the diagrammatic specification (which is separate from the Z specification). In DAIDA [GSW93], a formal notation is used to specify information systems, but with no diagrammatic notation. It allows the derivation of relational database programs. Formal refinement is carried from TDL, a formal conceptual modeling language, to B. A similar approach using Asso and B is applied to object-oriented databases in [ML99]. The CSP2B tool [But00] allows to mix CSP specification and action systems specifications using the B notation. The CSP-OZ [Fis97] is an integrated formal method which combines the state-oriented specification language Object-Z with CSP. These approaches are oriented towards event-driven systems and distributed systems.

1.6 Structure of the Paper

In Section 2, we introduce our formal requirements model. An overview of the EB^3 specification notation is presented. Section 3 introduces the restricted UML notation for the diagrams used in the analysis model. An

overview of the translation of these diagrams into B is presented. The two methods are illustrated by using a simple case study. In Section 4, we outline the proof strategy for showing that an analysis model is correct with respect to a requirements model. Relevant refinement proofs issued from the case study are presented. Section 5 concludes with an appraisal of this work, identifying its strengths, weaknesses and future work.

2 Formal Requirements Model

Our formal requirements model is made of use case diagrams, the requirements class diagram and an EB^3 specification. We omit the use case diagram in this section, as it consists mainly of a textual description of the user requirements.

The semantics of an EB^3 specification is given by a relation R such that $R \subseteq I^+ \times O$, where I is the set of input events that the environment can submit to the system and O is the set of output events that the system can send to the environment. Expression I^+ denotes the set all non-empty finite sequences built using elements of I (i.e., input traces). An EB^3 is constructed in four steps:

1. specification of the requirements class diagram, which contains the entity types, their associations, and their event names;
2. specification of event signatures (input and output);
3. specification of input ordering constraints;
4. specification of an output for each input trace.

Step 2 defines the sets I and O. Step 3 defines the set of valid traces, called *Valid* in the sequel. Finally step 4 defines the pairs of relation R.

To illustrate our approach, we use a simple example throughout this paper, a video club system. It manages video cassettes and customers. A customer must register with the system in order to borrow a cassette.

2.1 The Requirements Class Diagram

In UML, the requirements class diagram consists of the entities of the system with their associations. It is represented by a simple class diagram without any attribute or method. In EB^3, we add to this diagram the list of external input events for each entity and association. These are the events submitted by the environment (e.g., the user or another system). In the requirements class diagram, we use the terms *entity type* and *entity* instead of class and object. Figure 3 illustrates the requirements class diagram for the video club system.

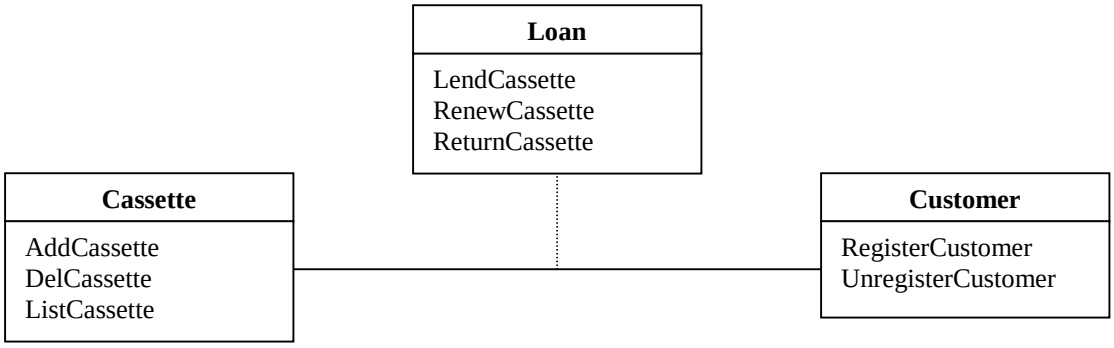

Figure 3: The requirements class diagram of the video club

2.2 The Events

The event signatures are declared using a list of pairs of the form *input_event : output_event*. An input event declaration is of the form $event\_label(inParam_1 : TYPE_1, \ldots, inParam_n : TYPE_n)$, where $inParam_i$ denotes a parameter name and $TYPE_1$ denotes a type name (a type is a set). An output event declaration is of the form $(outParam_1 : TYPE_1, \ldots, outParam_n : TYPE_n)$, where $outParam_i$ is an output parameter name. We assume that after receiving an input, the system produces the associated output *before* processing the next input. This assumption still allows the implementation of the specification using parallel transactions in a database management with proper concurrency control techniques (e.g., two-phase locking or transaction serialization). When the system does not produce a visible output for an input event (for, e.g., an update input event), the special value "void" is used.

$M_1 \triangleq$ RegisterCustomer($customer : CUSTOMER, name : STRING$)
: void
$M_2 \triangleq$ UnregisterCustomer($customer : CUSTOMER$)
: void
$M_3 \triangleq$ AddCassette($cassette : CASSETTE, title : STRING$)
: void
$M_4 \triangleq$ LendCassette($customer : CUSTOMER, cassette : CASSETTE$)
: void
$M_5 \triangleq$ RenewCassette($cassette : CASSETTE$)
: void
$M_6 \triangleq$ ReturnCassette($cassette : CASSETTE$)
: void
$M_7 \triangleq$ ListCassette($cassette : CASSETTE$)
: ($title : STRING, loanCnt : NATURAL$)
$M_8 \triangleq$ DelCassette($cassette : CASSETTE$)
: void

2.3 The Input Ordering Constraints

Input ordering constraints define the valid system input traces. The constraints are defined by entity type, allowing a step-wise construction of the requirements model.

An entity in EB^3 is represented by a trace consisting of the inputs related to this entity. For instance, the cassette ca_1 of a video club can be represented by the following entity trace.

$$\mathsf{AddCassette}(ca_1, t) \,.\, \mathsf{LendCassette}(cu_1, ca_1) \,.\, \mathsf{ReturnCassette}(ca_1) \,.\, \mathsf{LendCassette}(cu_2, ca_1) \qquad (1)$$

This entity trace contains four inputs related to the cassette ca_1. Inputs are concatenated using operator "." to form a trace. From this trace, we know that the cassette ca_1 was borrowed by customer cu_1, returned, and borrowed by customer cu_2. The cassette is still on loan, as it has not been returned yet. To describe all possible traces for a cassette entity, we use a process expression whose syntax is very similar to regular expressions, CSP [Hoa85], CCS [Mil89], LOTOS [BB87] and Jackson's entity structure diagrams [Jac83]. A process expression is a specification of input ordering constraints. It can also be seen as a scenario specification for input events.

Entity Type **cassette**($ca : CASSETTE$) $\triangleq$

AddCassette($ca, \_$) .
(
  loan($\_, ca$)*
|||
  ListCassette(ca)*
) .
DelCassette(ca)

The body of this process definition provides that the first input of a cassette is an AddCassette. Symbol "_" denotes an arbitrary choice of a value from the type of the parameter. This AddCassette is followed by inputs from the interleave (operator "$|||$") of two process expressions. This interleave represents all possible ways of merging the input traces of each expression. For instance, $a \,|||\, (b \,.\, c) = (a \,.\, b \,.\, c) \mid (b \,.\, a \,.\, c) \mid (b \,.\, c \,.\, a)$, where operator "$\mid$" represents a choice between two expressions. The first expression of the interleave in process **cassette** provides that a cassette may be loaned zero or more times (operator "*"); the second one provides that zero or more inputs ListCassette can be submitted for a cassette. Finally these inputs are followed by a DeleteCassette. Process **loan** is defined below.

Auxiliary process $\mathbf{loan}(cu : \mathit{CUSTOMER}, ca : \mathit{CASSETTE}) \triangleq$

$\mathsf{LendCassette}(cu, ca)$.
$\mathsf{RenewCassette}(ca)^*$.
$\mathsf{ReturnCassette}(ca)$

A loan is a sequence of inputs starting with a LendCassette followed by zero or more RenewCassette and followed by a ReturnCassette. The precedence of the process operators is the following, from highest to lowest: *, ., $\mid$, $|||$.

A trace of a process expression is any sequence of events that the process may execute. A trace need not be complete (i.e., it does not need to include all the inputs up to the end of the process expression). For instance, the entity trace (1) is a trace of the process call $\mathbf{cassette}(ca_1)$.

It may be interesting at this point to relate the notion of entity trace to the notion of system state. When the video club system is implemented, some appropriate representation of the system state must be chosen. For instance, if a relational database is used, a cassette could be represented by a tuple in a cassette table. If the history of loans of a cassette must be remembered, it could be represented by a tuple in a loan table. If only the last loan needs to be remembered, it could be represented as attributes in the cassette table (e.g., with attributes borrower and loan date). Several other representations are possible. An entity trace is a very abstract representation of the system state for a particular entity. For instance, entity trace (1) represents the tuple of cassette ca_1 in the cassette table and associated loan tuples in the loan table (assuming the history of loans is required). Traces have several advantages over more concrete representations like tuples of relational DB or objects of OODB. First, they are very resilient to requirements changes. Whether it is decided to keep the history of loans or simply the last loan, it makes no change to the definition of the entity trace of a cassette. It contains all the inputs for a cassette; hence both the last loan or the history of loans can be determined from a cassette trace.

The other entity type of the system is a customer. Its definition is provided below.

Entity Type $\mathbf{customer}(cu : \mathit{CUSTOMER}) \triangleq$

$\mathsf{RegisterCustomer}(cu, \_)$.
$(\,|||\, x : x \in \mathit{CASSETTE} : \mathbf{loan}(cu, x)^*)$.
$\mathsf{UnregisterCustomer}(cu)$

Process **customer** also calls auxiliary process **loan**, because a customer participates to loans. It also contains a quantified operator of the form "$|||\, x : x \in X : p(x)$", which represents the interleave of a number of $\mathbf{loan}^*$ processes. For each element x of *CASSETTE*, a process $\mathbf{loan}(cu, x)^*$ is launched. This expression represents in a very compact manner the fact that a customer may borrow several cassettes concurrently.

There are important differences in the structure of the **cassette** entity type and the **customer** entity type with respect to loan scenarios. The first distinction is that a cassette is borrowed by one customer at a time, and that it can be borrowed several times during its life. The loan scenarios for a cassette are represented by expression $\mathbf{loan}(\_, \mathit{cassette})^*$. The * implies that expression **loan** is executed as many times as necessary (0 or more). A customer may borrow several cassettes concurrently, and he may borrow the same cassette as often he wants. The expression $\mathbf{loan}(cu, \_)^*$ alone is not appropriate for a customer trace, because it implies that a customer borrows only one cassette at a time. Hence, we had to use a quantified interleave. Note that the interleave of all elements of *CASSETTE* does not imply that a customer must borrow all the cassettes of the video club; the process expression allows it, but it also allows that some

cassettes are not borrowed, thanks to the * applied to the **loan** process call. The following trace is an example of an entity trace of customer cu_1.

$$\mathsf{RegisterCustomer}(cu_1, n) \,.\, \mathsf{LendCassette}(cu_1, ca_1) \,.\, \mathsf{LendCassette}(cu_1, ca_2) \,.\, \mathsf{ReturnCassette}(ca_1) \tag{2}$$

The customer cu_1 has borrowed two cassettes so far. He has returned the cassette ca_1, but he still has the cassette ca_2.

2.4 The System Input Trace

The system input trace records the input events in their arrival order. It represents a global view of the system, whereas an entity trace represent a local view for an entity. In this section, we describe how the system input trace can be constructed from the entity traces.

From entity traces (1) and (2), we know that there are at least two other entities in the system: the cassette ca_2, because it is borrowed by customer cu_1, and the customer cu_2, because he has borrowed cassette ca_1. Provided below are their respective entity traces. For the sake of the example, we assume there is no other entity.

$$\mathsf{AddCassette}(ca_2, t') \,.\, \mathsf{LendCassette}(cu_1, ca_2) \tag{3}$$

$$\mathsf{RegisterCustomer}(cu_2, n') \,.\, \mathsf{LendCassette}(cu_2, ca_1) \tag{4}$$

How should these four entity traces be composed to form a system input trace? Some inputs occur in two traces (e.g., $\mathsf{LendCassette}(cu_1, ca_1)$ occurs in (1) and in (2)), because both process expressions **cassette** and **customer** call auxiliary process **loan**. These inputs must occur only once in the system trace. Moreover, the order of inputs prescribe by one entity should be respected in the system trace. The following is a valid system input trace; there are several other possibilities.

$$\begin{array}{l}\mathsf{AddCassette}(ca_1, t) \,.\, \mathsf{RegisterCustomer}(cu_1, n) \,.\, \mathsf{LendCassette}(cu_1, ca_1) \,. \\ \mathsf{ReturnCassette}(ca_1) \,.\, \mathsf{RegisterCustomer}(cu_2, n') \,.\, \mathsf{LendCassette}(cu_2, ca_1) \,. \\ \mathsf{AddCassette}(ca_2, t') \,.\, \mathsf{LendCassette}(cu_1, ca_2) \,.\, \mathsf{ReturnCassette}(ca_1)\end{array} \tag{5}$$

The next trace is not a valid trace system input trace. It is built out of the concatenation of traces (1) to (3) with duplicate inputs removed.

$$\begin{array}{l}\mathsf{AddCassette}(ca_1, t) \,.\, \mathsf{LendCassette}(cu_1, ca_1) \,.\, \mathsf{ReturnCassette}(ca_1) \,. \\ \mathsf{LendCassette}(cu_2, ca_1) \,.\, \mathsf{RegisterCustomer}(cu_1, n) \,.\, \mathsf{LendCassette}(cu_1, ca_2) \,. \\ \mathsf{ReturnCassette}(ca_1) \,.\, \mathsf{RegisterCustomer}(cu_2, n') \,.\, \mathsf{AddCassette}(ca_2, t')\end{array} \tag{6}$$

The problem is that input $\mathsf{LendCassette}(cu_1, ca_1)$ occurs before the input $\mathsf{RegisterCustomer}(cu_1, n)$, which means that the cassette was loaned before the customer was registered. Similar problems occur for input $\mathsf{LendCassette}(cu_2, ca_1)$ and input $\mathsf{LendCassette}(cu_1, ca_2)$. Of course, it should be possible to submit any input at any point in time during the execution of the system. If the customer is not registered yet, or if the cassette has not been added yet, the system should respond by issuing an appropriate error[1] message. Hence, the final system should accept input trace (6). However, it is much simpler to deal with error messages in a subsequent phase of the specification process. In our approach, we restrict the specification of scenarios to valid system trace, i.e., traces that satisfy the entity type process expressions; all that is missing in our specification is which specific error message to issue when an invalid input is received.

The key to solve the problem of valid system input trace generation is to use the parallel composition operator "$||$" of CSP. This operator acts like a conjunction operator for traces. An expression $E_1 \,||\, E_2$ has the following meaning. If E_1 can execute an input i that does not occur in E_2, then E_1 can execute it alone. Similarly, if E_2 can execute an input i that does not occur in E_1, then E_2 can execute it alone. If E_1 can execute an input i that occurs in E_2, or vice-versa, then E_1 and E_2 must execute it simultaneously, and the input occurs only once in the trace. We say that E_1 and E_2 synchronize on shared inputs. Operator $||$ has

[1]The word "error"is used in its customary sense, which means that an informative message is sent to the user and the transaction processing does not modify the internal system state. It is possible to characterise this notion (i.e., the transaction did not induce any state modification) in terms of traces. For the sake of simplicity, we omit its definition here.

the same precedence as $|||$. For instance, consider the expression $(a \,.\, f \,.\, b) \,||\, (c \,.\, f \,.\, d)$. Input f occurs in (or is shared by) both operands of $||$. Therefore, it must be executed simultaneously by both operands. Inputs a, b, c, d are not shared. Hence, they can be executed alone. This expression has the same traces as the expression $(a \,|||\, c) \,.\, f \,.\, (b \,|||\, d)$, which are also the same as the traces of expression $(a \,.\, c \mid c \,.\, a) \,.\, f \,.\, (b \,.\, d \mid d \,.\, b)$.

The valid system traces obtained by the composition of entity traces (1), (2), (4), (3) are defined by the following process expression, where we use the trace number rather than the actual trace, for the sake of concision.

$$(\ (1) \,|||\, (3)\ )\ \ ||\ \ (\ (2) \,|||\, (4)\ ) \tag{7}$$

Traces (1) and (3) are interleaved, because there is no ordering constraint between two cassette entity traces. Similarly for customer entity traces (2) and (4). These interleave expressions are composed with a $||$, hence they synchronize on **loan** inputs. In that way, **loan** inputs occur only once in the system trace, and the input ordering of each entity type is respected. As an example, trace (5) is a trace of expression (7). We can generalise the expression (7) to represent all valid system traces for a system with two cassettes and two customers as follows.

$$(\mathbf{cassette}(ca_1) \,|||\, \mathbf{cassette}(ca_2))\ \ ||\ \ (\mathbf{customer}(cu_1) \,|||\, \mathbf{customer}(cu_2)) \tag{8}$$

A complete scenario specification must include an arbitrary number of entities for each entity type. Hence, we can generalise the expression (8) as follows.

$$(\ |||\, x : x \in \mathit{CASSETTE} : \mathbf{cassette}(x))\ \ ||\ \ (\ |||\, x : x \in \mathit{CUSTOMER} : \mathbf{customer}(x)) \tag{9}$$

2.5 The Specification of Outputs

The final step of our requirements specification method is to associate an output to each valid system input trace. For instance, consider the following one element system trace: $\mathsf{AddCassette}(ca_1, t)$. According to the list of messages, the output produced for an input AddCassette is void, which means that no visible output is sent to the environment. We use the special symbol τ to denote this output. Hence, we have that $(\mathsf{AddCassette}(ca_1, t), \tau) \in R$. Now consider the following trace.

$$\mathsf{AddCassette}(ca_1, t) \,.\, \mathsf{ListCassette}(ca_1) \tag{10}$$

The list of events provides that the output produced for an input ListCassette is the title and the number of loans. For input trace (10), the output is tuple $\langle t, 0\rangle$ (we use a tuple $\langle v_1, \ldots, v_n\rangle$ to denote an output event). For input trace (5) $.\, \mathsf{ListCassette}(ca_1)$, (i.e., input trace (5) right-appended with input $\mathsf{ListCassette}(ca_1)$), the output should be $\langle t, 2\rangle$, because cassette ca_1 has been loaned twice in input trace (5). To precisely define what is the output associated with a valid system input trace, we use recursive functions. For input ListCassette, we need the recursive function $\mathit{nbLoans}(s, ca)$, which computes the number of loans for a system input trace s and a customer ca. It calls traditional sequence functions *last* (returns the last element of a sequence) and *front* (returns all but the last element of a sequence).

$\mathit{nbLoans}(s, ca) \triangleq$
  case
    $\mathit{last}(s) = \mathsf{AddCassette}(ca, \_)$: **return** 0;
    $\mathit{last}(s) \in \{\mathsf{LendCassette}(\_, ca), \mathsf{RenewCassette}(ca)\}$: **return** $1 + \mathit{nbLoans}(\mathit{front}(s), ca)$;
    otherwise : **return** $\mathit{nbLoans}(\mathit{front}(s), ca)$
  endCase

A function that computes the title of a cassette, say $\mathit{cTitle}(s, ca)$, can be defined in similar manner. We are now ready to define the rule that specifies the output for an input ListCassette.

rule R1
  var $ca : \mathit{CASSETTE}$
  input ListCassette(ca)
  output $\mathit{title} := \mathit{cTitle}(s, ca)$, $\mathit{loanCnt} := \mathit{nbLoans}(s, ca)$
endrule

Variable s represents by convention the system trace. It is possible to refer to message output parameters (i.e., *title* and *loanCnt*) in the output clause. Rules have a formal semantics defined in first-order logic. For instance, the rule R1 denotes the following predicate.

$$\begin{array}{l}\forall\, ca, s : \\ \quad ca \in CASSETTE \wedge s \in Valid \wedge last(s) = \mathsf{ListCassette}(ca) \\ \quad \Rightarrow \\ \quad (s, \langle cTitle(s, ca), nbLoans(s, ca)\rangle) \in R\end{array}$$

Set *Valid* denotes by convention the set of valid input traces, which is defined as the traces of (9).

If we compare trace-based specifications and state-based specifications, we find that the function *nbLoans* is very similar to an attribute nbLoans of a cassette table in a relational DB. In general, it is possible to define a recursive function for each attribute of an entity type. Note how encapsulated is the definition of *nbLoans* compared with a state-based implementation (or state-based specification). In a state-based implementation, attribute nbLoans would be modified in two places : in the method that handles an AddCassette transaction (to be initialised at zero) and in the method that handles a LendCassette transaction (to be incremented by one).

In order to write concise specifications of attributes, we use a number of predicates and operators on traces. Predicate $entity\_traces(s, T)$ allows the decomposition of a valid system input trace s into entity traces. Symbol T denotes the decomposition of s. It is a function taking as argument an entity type name (e.g., **cassette**) and the entity type parameters (e.g., a cassette ca_1), and returning the associated entity trace. For example, when $entity\_traces((5), T)$ holds, then T has the following value.

$$\begin{array}{rcl@{\qquad\qquad}rcl} T(\mathbf{cassette})(ca_1) & \triangleq & (1) & T(\mathbf{cassette})(ca_2) & \triangleq & (3) \\ T(\mathbf{customer})(cu_1) & \triangleq & (2) & T(\mathbf{customer})(cu_2) & \triangleq & (4) \end{array}$$

The following operators are also used. Expression $t{\restriction}\{event\_label\}$ is the restriction of trace t to events of label *event_label*. Expression $\#t$ denotes the length of trace t. Finally, expression $param_i(e)$ denotes the value of parameter $param_i$ in event e. Using these operators, we may rewrite function *nbLoans* as follows. Assume that $entity\_traces(s, T)$ holds.

$$nbLoans(T, ca) \;=\; \#\,(T(\mathbf{cassette})(ca){\restriction}\{\mathsf{LendCassette}\})$$

Similarly, we can define a function *cTitle* which returns the title of a cassette.

$$cTitle(T, ca) \;=\; title(last(T(\mathbf{cassette})(ca){\restriction}\{\mathsf{AddCassette}\}))$$

That completes the presentation of the EB^3 specification method. For more details, the reader is referred to [FSD99b]. The entire specification of the video club system consists of the list of messages, process expressions **cassette**, **customer**, and **loan**, attributes *cTitle* and *nbLoans*, and output rule R1. Other formula where introduced to either define the semantics of the specification language or to illustrate the approach.

3 Formal Analysis Model

The requirements capture phase describes the scenarios of external events exchange between the system and its environment. It allows the developers and the customers to agree on the external behavior of the system. The aim of the analysis phase is to provide a specification of the requirements described using the language of the developers. Thus it provides an abstract description of the software that will implement the functionalities satisfying the requirements. In information systems, it describes the data of the system, but without specifying how they will be implemented, and the effects of the external events on these data.

The language used to described the formal analysis model is based on UML for its graphical syntax and B for its formal semantics. In this section, we illustrate some features of the language using diagrams corresponding to the case study and their translation into a B specification.

The first step in the preparation of the analysis model is the derivation of the analysis class diagram, which defines all the classes and associations with their multiplicities. Then, each external event is described by using specific UML constructs.

3.1 The Analysis Class Diagram

The analysis class diagram is of course derived from the requirements class diagram since it includes all the entity types and associations. Generic basic operations, which are application-independent, can be automatically generated from the diagram. For each class, the following basic update operations are generated:

- insert an object in a class : **B_AddClassName**, or a link in an association : **B_AddAssociationName**
- delete an object from a class : **B_DelClassName**, or a link from an association : **B_DelAssociationName**
- Modify the value of an attribute : **B_ModifyAttributeName**

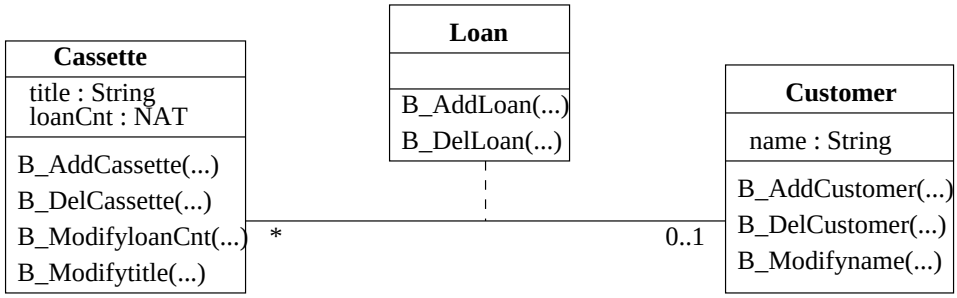

Figure 4: The analysis class diagram of the video club

The Figure 4 provides the analysis class diagram of the video club system. An instance of **Cassette** is characterised by its title. The attribute *loanCnt* gives the number of loans of a cassette. It is added in order to process the event ListCassette. A customer is characterised by its name. Multiplicity 0..1 means that a cassette can be borrowed by at most a customer and multiplicity * means that a customer can borrow 0 or more cassettes.

There are important differences between the requirements class diagram and the analysis class diagram, illustrating that an analysis model is closer to an implementation than a requirements model. First, the requirements class diagram contains the external input events for each entity type and association whereas the analysis class diagram describes the attributes and basic operations for each class and association. Secondly, the multiplicities of the associations are specified in the analysis class diagram. In the example, the multiplicity 0..1 for the association **Loan** means that only the current loan is stored; it is not necessary to store the whole history of loans of a cassette. This fact can be deduced by analyzing the behavior of events using the association. Finally some entity types of the requirements class diagram may be merged or split when the analysis model is prepared, due to design choices.

3.2 Translation of the Analysis Class Diagram into B

A class diagram can be automatically translated into a B specification. A B specification is a set of abstract machines. Each machine encapsulates state variables, an invariant constraining the state variables, and operations on the state variables. Operations are specified in the generalised substitution language which is an extension of Dijkstra's guarded command notation. A substitution is like an assignment statement, but a clever assignment statement. It allows to identify which variables are modified during an operation, while avoiding to mention which ones are not. Proof of internal consistency of an abstract machine requires that each machine operation on the state variables maintains the invariant. Large machines are constructed using smaller machines through various access relationships (e.g., *uses*, *includes*, etc). An access relationship defines what the accessing machine can do with the components of the accessed machine.

The translation of the class diagram almost entirely provides the static part of the B specification (i.e., clauses SETS, VARIABLES, INVARIANT). One abstract machine is generated for each class. If an association contains specific attributes and/or operations, an abstract machine is generated, otherwise the

association is included in the machine of one of the classes involved in the association. The B specification corresponding to the class *Customer* of the running example is given below. Substitution operator *asyncpar* in the B notation denotes simultaneous substitutions. Relational expression $A \lhd U$ denotes the restriction of the domain of relation U to the complement of set A; hence $\lhd$ is called the antirestriction operator.

MACHINE *Basic_Customer*

SETS
$CUSTOMER$; (11)

VARIABLES
$customer_{sv}$, (12)
$name_{sv}$, (13)

INVARIANT
$customer_{sv} \subseteq CUSTOMER \ \wedge$ (14)
$name_{sv} \in customer_{sv} \rightarrow STRING \ \wedge$ (15)

INITIALISATION
$customer_{sv} := \{\} \ ||$
$name_{sv} := \{\}$

OPERATIONS
B_AddCustomer$(customer_p, name_p) \triangleq$

PRE (16)
$customer_p \in CUSTOMER - customer_{sv} \ \wedge$
$name_p \in STRING$
THEN
$customer_{sv} := customer_{sv} \cup \{customer_p\} \ ||$
$name_{sv}(customer_p) := name_p$
END;

B_DelCustomer$(customer_p) \triangleq$

PRE
$customer_p \in customer_{sv}$
THEN
$customer_{sv} := \{customer_p\} \lhd customer_{sv} \ ||$
$name_{sv} := \{customer_p\} \lhd name_{sv}$
END;

B_ModifyName$(customer_p, name_p) \triangleq$

PRE
$customer_p \in customer_{sv} \ \wedge$
$name_p \in STRING$
THEN
$name_{sv}(customer_p) := name_p$
END;

The B machine generated for the class *Cassette* has the same structure as the B machine for *Customer*. The set of instances of a class is modeled by an abstract set of all possible instances and a variable representing the set of existing instances. For example, the set of instances of the class *Customer* is translated into the lines (11), (12) and (14) in machine *Basic_Customer*. Each attribute is modeled by a relation between the

set of existing instances and the attribute type. Thus attribute *name* is translated into the lines (13) and (15). As a convention, we use the subscripts $_{sv}$ for state variables and $_{p}$ for operation parameters.

Each association is modeled by a relation between two sets of existing instances. Depending on the multiplicity of each role, the relation may become a total ($\rightarrow$) or partial ($\nrightarrow$) function, an injection, etc. An association class which is not involved in other relationships (inheritance links or other associations) is modeled as a relation. Thus association *loan* is translated into the lines (17) and (18) in the machine *Basic_Loan* provided below.

MACHINE *Basic_Loan*

USES
Basic_Customer, *Basic_Cassette* /* USES clause allows variables to be shared */

VARIABLES
$loan_{sv}$ (17)

INVARIANT
$loan_{sv} \in cassette_{sv} \nrightarrow customer_{sv}$ (18)

INITIALISATION
$loan_{sv} := \{\}$

OPERATIONS
B_AddLoan$(customer_p, cassette_p) \triangleq$

PRE
$$\begin{array}{rcl} customer_p & \in & customer_{sv} \ \wedge \\ cassette_p & \in & cassette_{sv} \ \wedge \\ cassette_p & \notin & dom(loan_{sv}) \end{array} \quad (19)$$
THEN
$loan_{sv}(cassette_p) := customer_p$
END;

B_DelLoan$(cassette_p) \triangleq$

PRE
$cassette_p \in cassette_{sv}$
THEN
$loan_{sv} := \{cassette_p\} \lhd cassette_{sv}$
END;

Each basic operation has a precondition, see for example (16), which includes parameter typing and conditions that can be derived from the class diagram, such as those that take into account the multiplicities of an association or the ordering of inputs events. For instance, line (19) ensures that it is not possible to accept a LendCassette for a cassette which is already on loan.

3.3 Description of External Events

Events of the requirements model can be described by UML diagrams completed with formal notations. A state diagram can be elaborated for a class : it specifies operations with more complex preconditions, dependent of the environment, than those of the basic update operations. A collaboration diagram can be elaborated if several classes are involved by the event. It specifies the called operations of each class. For example the event LendCassette triggers operation **B_AddLoan** of association class **Loan** and operation **B_ModifyloanCnt** of class **Cassette**. The corresponding collaboration diagram is described in the Figure 5. If graphical notations are not suitable to specify an event (because diagrams become unreadable or

there is no well-adapted graphical construct) then the designer can describe it in a textual manner, using a language whose syntax is close to the B syntax [LP01b]. It is the case for the event ListCassette (20).

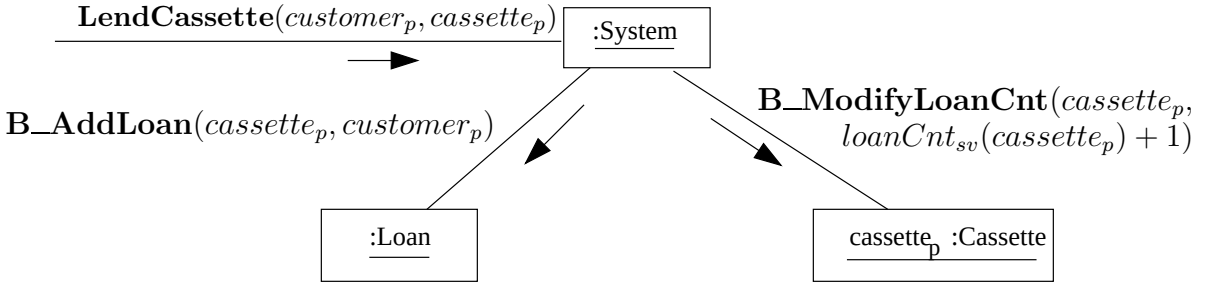

Figure 5: The collaboration diagram of the event LendCassette

3.4 Translation of the Events Specification into B

The translation of the events specification completes the architecture of the B specification obtained from the class diagram. It defines a top-level machine which includes basic machines, has no variable and contains one operation for each event of the requirements model. These operations are defined using basic operations from the basic machines. The top level machine forms the interface of the global B specification. In our case, it is called $VideoClubAnalysisModel$.

MACHINE $VideoClubAnalysisModel$

INCLUDES /* Allows access to variables and */
$Basic\_Customer, Basic\_Cassette, Basic\_Loan$ /* operations of the basic machines. */

OPERATIONS

RegisterCustomer$(customer_p, name_p)$ /* Event RegisterCustomer corresponds exactly to */
PRE /* the basic update operation **B_AddCustomer**. */
$customer_p \in CUSTOMER - customer_{sv} \wedge$
$name_p \in STRING$
THEN
$B\_AddCustomer(customer_p, name_p)$

LendCassette$(customer_p, cassette_p) \triangleq$
PRE
$customer_p \in customer_{sv} \wedge$
$cassette_p \in cassette_{sv} \wedge$
$cassette_p \notin dom(loan_{sv}) \wedge$
$loanCnt_{sv}(cassette_p) + 1 \in NAT$
THEN
$B\_AddLoan(cassette_p, customer_p) \;||$
$B\_Modifyloan Cnt(cassette_p, loanCnt_{sv}(cassette_p) + 1)$
END;

$(tl, lc) \leftarrow$ **ListCassette**$(cassette_p) \triangleq$ (20)

PRE

$cassette_p \in cassette_{sv}$

THEN

$tl := title_{sv}(cassette_p) \;||$

$lc := loanCnt_{sv}(cassette_p)$

END;

Once the entire specification is derived, internal machine consistency proofs can be carried out. They establish that each operation, called under its precondition, preserves the invariant of the specification. More details on the method can be found in [FLN+99].

4 The Refinement Proof Strategy

In this section, we present our strategy for proving the correctness between a requirements model and an analysis model. We use the B notation as an intermediate to conduct the proof because the analysis model is already expressed with B and the refinement process is well formalized in the B method. The requirements model is translated into a B machine (see Figure 2). This transformation can be done automatically. We say that the analysis model *is correct* with respect to the requirements model iff the B specification of the analysis model *refines* the B specification of the requirements model.

In the sequel, we first present the notion of refinement in B and the principles of our refinement process. We then introduce in turn the requirements B specification. The analysis B specification has already been presented in Section 3. Finally, we present the different steps of our refinement process and discuss some relevant examples of the refinement proofs.

4.1 The Refinement Process

4.1.1 Refinement in B

A machine M is refined by a machine M' (denoted $M \sqsubseteq M'$) iff their operation's signatures are the same and the operations of M' preserve the *observable behaviour* of the operations of M. The preservation of observable behavior for operations can be defined as follows. Let s be an arbitrary sequence of operations calls starting from the initialisation of the machine. If machine M can successfully executed this sequence of operation calls, then M' must also be able to successfully execute it. Moreover, let o an output produced by an operation call in s. If a call in M' delivers output o, then output o must be allowed by machine M. In other words, machine M can be replaced by machine M', and the observable behavior of M' is preserved. Note that this definition allows M' to accept sequence of operation calls that M does not accept. It also allows M' to reduce the nondeterminacy of M.

The proof obligations associated to the refinement of M by M' are as follows. Consider the B specifications of M and M" in Figure 6. First we have to prove that the initialisation is correctly refined, which is translated by the following proof obligation.

$$[Init_{M'}]\neg[Init_M]\neg Inv_{M'} \tag{21}$$

For each operation $\mathbf{op}_i$, we have to prove the following.

$$\begin{array}{l} Inv_M \wedge Inv_{M'} \wedge P_M \\ \Rightarrow \\ P_{M'} \wedge [S_{M'}]\neg[S_M]\neg Inv_{M'} \end{array} \tag{22}$$

The refinement proof obligations are expressed in terms of weakest-precondition. For substitution S and postcondition P, the formula $[S]P$ characterizes those initial states from which S is guaranteed to terminate in a state satisfying P. When substitutions are deterministic, the proof obligations take a simpler form, because the two negations cancel each other. This will be the case for the refinement proofs presented in the sequel. Their simplified forms are the following.

$$[Init_{M'}][Init_M]Inv_{M'} \tag{23}$$

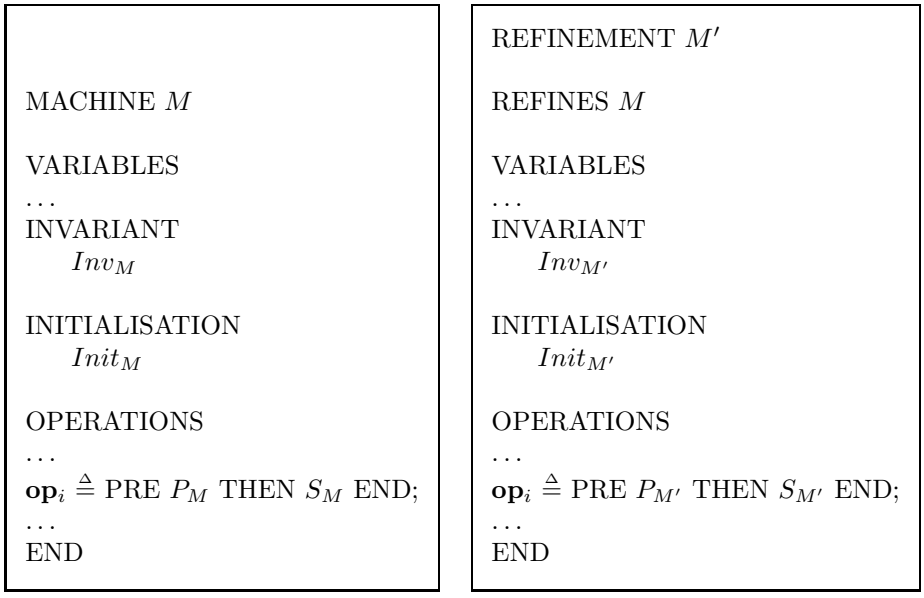

Figure 6: A machine M and its refinement M'

$$\begin{array}{l} Inv_M \wedge Inv_{M'} \wedge P_M \\ \Rightarrow \\ P_{M'} \wedge [S_{M'}][S_M] Inv_{M'} \end{array} \tag{24}$$

A refinement machine must include in its invariant a formula, called the *gluing invariant*, relating the abstract state of the refined machine and the concrete state of the refining machine. The gluing invariant is a key in the refinement proof.

4.1.2 Main Principles of our Refinement Process

In our case, the aim of the refinement process is to transform a trace-based representation into a state-based representation where state variables model the entities and associations of the system and where operations translate the effects of the requirements model events on these state variables. In order to simplify this transformation and the refinement proof, we use an intermediate refinement machine between the requirements B specification and the analysis B specification. This intermediate refinement machine is manually generated by the software designer; computer support could be provided to generate a skeleton for the non creative parts.

$$RequirementsModel \sqsubseteq IntermediateModel \sqsubseteq AnalysisModel$$

Refinement is transitive. The first refinement consists of splitting the system trace into entity traces. The second refinement consists of explicitly defining the variables needed to store information which is required to process the events. These variables correspond to the elements of the analysis class diagram. Figure 7 illustrates the refinement process with the intermediate refinement machine.

In fact, the second refinement is not a "classical" B refinement since the analysis B specification is already defined. In order to establish that *it is* a refinement of the intermediate refinement machine, we add the gluing invariant. This invariant relates objects, attributes and associations to entity traces whereas the gluing invariant in the intermediate machine relates entity traces to the system trace. Figure 8 illustrates the connection between the state spaces of the three machines involved in refinement.

4.2 The B Machine for the Requirements Model

The B Machine for the requirements model is provided below. This machine can be generated automatically from the EB^3 spec.

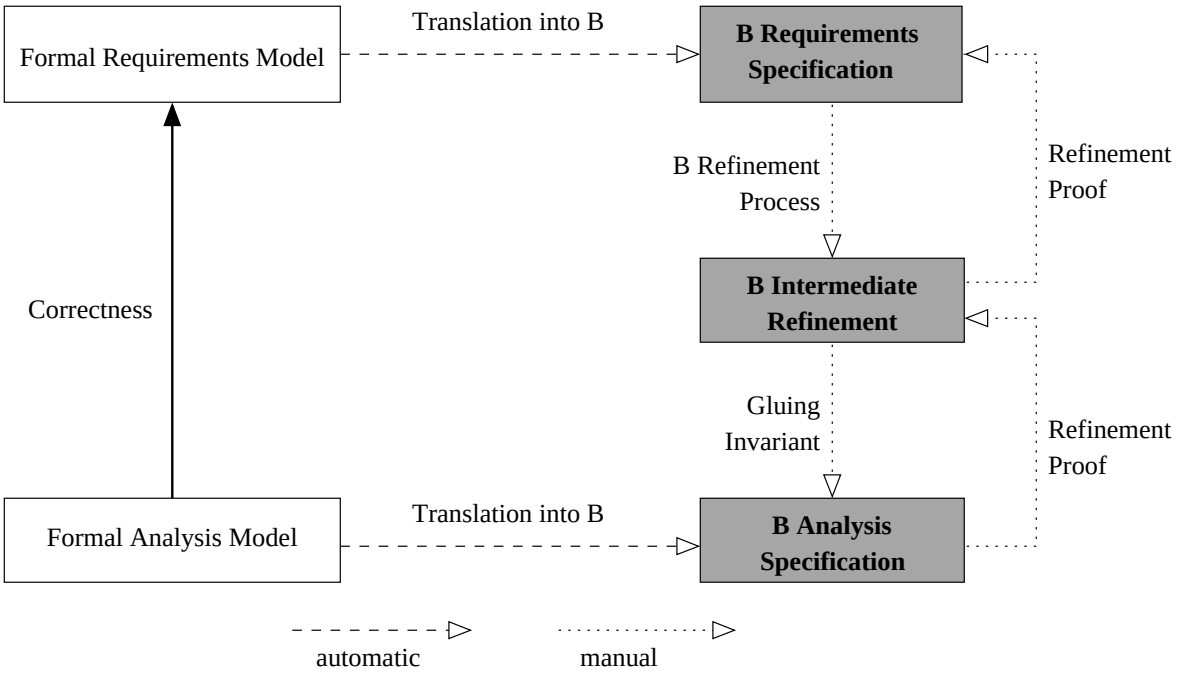

Figure 7: The complete refinement process

Requirements Machine	Intermediate Machine	Analysis Machine
system trace	entity traces	objects
AddCassette(ca, t) . RegisterCustomer(cu, n) . LendCassette(cu, ca)	**cassette** ca : AddCassette(ca, t) . LendCassette(cu, ca) **customer** cu : RegisterCustomer(cu, n) . LendCassette(cu, ca)	$cassette_{sv} = \{ca\}$ $title_{sv} = \{ca \mapsto t\}$ $loanCnt_{sv} = \{ca \mapsto 1\}$ $customer_{sv} = \{cu\}$ $name_{sv} = \{cu \mapsto n\}$ $loan_{sv} = \{ca \mapsto cu\}$

Figure 8: Example of state values for each machine

MACHINE $VideoClubRequirements$

SEES /* Allows properties to be shared. */
$ProcessAlgebraDef$, $VideoClub^{\text{EB}^3}Spec$

SETS
$CUSTOMER, CASSETTE$

VARIABLES
$systemTrace$

INVARIANT
$systemTrace \in Valid$ /* *Valid* is the set of valid traces. */
/* A trace is expressed using a sequence in B. */

INITIALISATION

$systemTrace := [\,]$

OPERATIONS

RegisterCustomer$(customer_p, name_p) \triangleq$

LET $traceAfter$ BE
$traceAfter = systemTrace \leftarrow \langle \mathsf{RegisterCustomer}, customer_p, name_p \rangle$
IN
PRE
$customer_p \in CUSTOMER \wedge$
$name_p \in STRING \wedge$
$traceAfter \in Valid$
THEN
$systemTrace := traceAfter$
END
END;

LendCassette$(customer_p, cassette_p) \triangleq$

LET $traceAfter$ BE
$traceAfter = systemTrace \leftarrow \langle \mathsf{LendCassette}, customer_p, cassette_p \rangle$
IN
PRE
$customer_p \in CUSTOMER \wedge$
$cassette_p \in CASSETTE \wedge$
$traceAfter \in Valid$
THEN
$systemTrace := traceAfter$
END
END;

$o_p \longleftarrow$ **ListCassette**$(cassette_p) \triangleq$

LET $traceAfter$ BE
$traceAfter = systemTrace \leftarrow \langle \mathsf{ListCassette}, cassette_p \rangle$
IN
PRE
$cassette_p \in CASSETTE \wedge$
$traceAfter \in Valid$
THEN
ANY o WHERE $(traceAfter, o) \in R$ THEN
$systemTrace := traceAfter \;||$
$o_p := o$
END
END
END;
...

Machine *VideoClubRequirements* has only one state variable, $systemTrace$, which contains the current value of the valid system trace. The invariant states that variable $systemTrace$ is always valid, that is, it is a trace of the process expression (9). The initial state of the machine is the empty trace. The machine contains one operation for each input event declared in the input-output space. Each operation has the same structure: the precondition checks that the system trace appended (using sequence operator $\leftarrow$) with

the input event corresponding to the operation call constitutes a valid trace; the postcondition appends the input event to the system trace and produces any output allowed by the EB^3 specification. The machine $VideoClubRequirements$ sees two machines: the machine $ProcessAlgebraDef$, which contains the definitions related to the mathematical foundations of EB^3 specifications and the machine $VideoClub$EB^3Spec, which contains the EB^3 specification of relation R (the input-output relation)and set *Valid* for the video club. For the sake of concision, we omit their definitions here. Moreover, for illustration purposes it suffices to provide the definition of three operations; the others are omitted.

4.3 The First Refinement Step

We first describe the intermediate refinement machine and then discuss the refinement proof.

4.3.1 The Intermediate Refinement Machine

Refinement of machines in B consists in defining a more concrete state space and redefining machine operations based on this concrete state space. To simplify the refinement proofs, it is preferable to proceed in small steps for the refinement. For this first refinement step, we have chosen to structure the state space using the entity traces. Hence, the machine has only one state variable, T (it has already been described in Section 2.5). This choice is general and applicable to any specification constructed using our approach. The gluing invariant states that T is a decomposition of $systemTrace$ into entity traces, i.e., that $entity\_traces(systemTrace, T)$ holds.

The intermediate refinement machine $VideoClubIntermediate$ is provided below. An inquiry input like ListCassette must compute its output from the entity traces. Hence, we made use here of the functions $cTitle$ and $nbLoans$ defined in Section 3. It must also add the operation call to the cassette entity trace, because the entity traces of T must be a decomposition of the system trace, which includes ListCassette. One can see that the main difference between the abstract operation and the refined operation for an input update event is in the precondition expression. Indeed, the state update is still quite primitive, as it simply consists of adding the input corresponding to the operation call to each entity trace involved with that input. For an operation related to an entity type in the requirements class diagram, there is usually (but not necessarily) only one entity trace to update. For instance, only the customer entity trace is updated in operation **RegisterCustomer**. For an operation related to an association in the requirements class diagram, like **LendCassette**, the entity trace of each entity associated must be updated. Hence, in **LendCassette**, both the customer entity trace and the cassette entity trace must be updated. The precondition can be found by analysing the EB^3 expression of each entity. Intuitively a precondition is a translation of the ordering constraints specified in the EB^3 specification. Note that other preconditions can be defined, for example to enforce integrity constraints of the system. Let us consider the **LendCassette** operation. The EB^3 specification provides that a LendCassette event must always be preceded by a RegisterCustomer and an AddCassette, that there are no UnregisterCustomer and DeleteCassette before, and finally that the cassette to borrow has not already been borrowed. This information is manually gathered by inspecting all entity type process expressions where input LendCassette occurs.

In the following machine, the set E represents the set of all the entity type names of the EB^3 specification. The set K represents the union of all entity type parameters. For the video club system, we have $\mathsf{K} \triangleq CASSETTE \cup CUSTOMER$.

REFINEMENT $VideoClubIntermediate$

REFINES $VideoClubRequirements$

SEES
$ProcessAlgebraDef$
$VideoClub$EB^3Spec

DEFINITIONS
/* these definitions are used to simplify precondition expressions*/

$activeCustomer(T, c) \triangleq c \in dom(T(\mathbf{customer})) \ \wedge$ /* a customer is *active* if he has been */
$label(last(T(\mathbf{customer})(c))) \neq \mathsf{UnregisterCustomer}$ /* registered but not unregistered */

$activeCassette(T, c) \triangleq c \in dom(T(\mathbf{cassette})) \ \wedge$ /* a cassette is *active* if it has */
$label(last(T(\mathbf{cassette})(c))) \neq \mathsf{DelCassette}$ /* been created but not deleted */

$onLoan(T, c) \triangleq label(last(T(\mathbf{cassette})(c)$ /* a cassette is *onLoan* if the */
$\lceil\lceil\{\mathsf{LendCassette}, \mathsf{ReturnCassette}\})) = \mathsf{LendCassette}$ /* last lend or return is a lend */

$borrower(T, c) \triangleq$ /* the borrower of a cassette is the */
$customer(last(T(\mathbf{cassette})(c)\lceil\lceil\{\mathsf{LendCassette}\}))$ /* customer of the last lend */

VARIABLES /* T is the decomposition of */
T /* system trace into entity traces */

INVARIANT
$T \in \mathsf{E} \nrightarrow (\mathsf{K} \nrightarrow seq(I)) \ \wedge$
$entity\_traces(systemTrace, T)$ /* gluing invariant */

INITIALISATION
$T{:=}\{\}$

OPERATIONS

RegisterCustomer$(customer_p, name_p) \triangleq$

PRE
$customer_p \in CUSTOMER \ \wedge$
$name_p \in STRING \ \wedge$
$customer_p \notin dom(T(\mathbf{customer}))$
THEN
$T(\mathbf{customer})(customer_p) := \langle \mathsf{RegisterCustomer}, customer_p, name_p \rangle$
END

LendCassette$(customer_p, cassette_p) \triangleq$

PRE
$customer_p \in CUSTOMER \ \wedge$
$cassette_p \in CASSETTE \ \wedge$
$activeCustomer(T, customer_p) \ \wedge$
$activeCassette(T, cassette_p) \ \wedge$
$\neg onLoan(T, cassette_p)$
THEN
$T(\mathbf{customer})(customer_p) := T(\mathbf{customer})(customer_p) \leftarrow \langle \mathsf{LendCassette}, customer_p, cassette_p \rangle \ ||$
$T(\mathbf{cassette})(cassette_p) := T(\mathbf{cassette})(cassette_p) \leftarrow \langle \mathsf{LendCassette}, customer_p, cassette_p \rangle$
END;

$(tl, lc) \leftarrow$ **ListCassette**$(cassette_p) \triangleq$

PRE

$cassette_p \in CASSETTE \wedge$
$activeCassette(T, cassette_p)$
THEN
$tl := cTitle(T, cassette_p) \;||$ /* Functions $cTitle$ and $nbLoans$
$lc := nbLoans(T, cassette_p)$ are defined in Section 2.5 */
END;

...
END

4.3.2 Proving the Refinement of VideoClubRequirements by VideoClubIntermediate

For the sake of illustration and concision, we content ourselves with a proof of the initialisation clause and one operation (**RegisterCustomer**). It is sufficient to provide a flavor of the refinement proof complexity.

4.3.2.1 Initialisation Let $Init_{RM}$ and $Init_{Inter}$ be the initialisation substitutions of machine VideoClubRequirements and machine VideoClubIntermediate, respectively. Let Inv_{Inter} be the invariant of machine VideoClubIntermediate. We have to prove the following statement:

$$[Init_{Inter}][Init_{RM}]Inv_{Inter}$$

This proof is quite simple. Because the formula involved here spans on several lines, and for space limitations, we content ourselves with a sketch of the proof. Applying substitutions $Init_{RM}$ and $Init_{Inter}$, we obtain the following formula.

$$\{\} \in \mathsf{E} \pfun (\mathsf{K} \pfun seq(I)) \;\wedge$$
$$entity\_traces([], \{\})$$

These two conjuncts trivially hold.

4.3.2.2 Operation RegisterCustomer We use the following abbreviations:

- Pre_{RM} and Pre_{Inter} : precondition predicates of operation **RegisterCustomer** in machine VideoClubRequirements and in machine VideoClubIntermediate, respectively;
- S_{RM} and S_{Inter} : THEN substitutions of operation **RegisterCustomer** in machine VideoClubRequirements and in machine VideoClubIntermediate, respectively;
- Inv_{Inter} : invariant of machine VideoClubIntermediate.

We have to prove the following statement:

$$\begin{array}{l} Inv_{RM} \wedge Inv_{Inter} \wedge Pre_{RM} \\ \Rightarrow \\ Pre_{Inter} \wedge [S_{Inter}][S_{RM}]Inv_{Inter} \end{array} \tag{25}$$

We assume the antecedent of (25) and prove each conjunct of the consequent of (25) separately. We start with the proof of Pre_{Inter}. To work with more compact formula, let us use some abbreviations. Let e_1, e_2 be **customer** and **cassette**, respectively; let rc be $\langle \mathsf{RegisterCustomer}, customer_p, name_p \rangle$; let s, c_p be $systemTrace$, $customer_p$; let T' be $T \lhd\!\!+ \{(e_1, T(e_1) \lhd\!\!+ \{(c_p, rc)\})\}$, where $\lhd\!\!+$ is the relational override operator (i.e., $U \lhd\!\!+ V \triangleq (dom(V) \lhd\!\!- U) \cup V$. From the definitions of the antecedent, we have the following hypotheses:

$$entity\_traces(s, T) \;\wedge \tag{26}$$
$$c_p \in CUSTOMER \;\wedge \tag{27}$$
$$name_p \in STRING \;\wedge \tag{28}$$
$$s \leftarrow rc \in Valid \tag{29}$$

By definition of Pre_{Inter}, we must prove

$$c_p \in CUSTOMER \ \wedge \tag{30}$$
$$name_p \in STRING \ \wedge \tag{31}$$
$$c_p \notin dom(T(e_1)) \tag{32}$$

Since (30) and (31) are hypotheses (27) and (28), it remains to prove (32), which we do by contradiction. Assume that $c_p \in dom(T(e_1))$. Then, by $entity\_traces(s, T)$, we may conclude that s contains an input RegisterCustomer for c_p. This means that $s \leftarrow rc$ contains two inputs RegisterCustomer for c_p, which contradicts entity type **customer** and hypothesis (29).

We may now prove $[S_{Inter}][S_{RM}]Inv_{Inter}$. Applying the substitutions and expanding the definitions, we have to prove the following formula.

$$T' \ \in \ \mathsf{E} \nrightarrow (\mathsf{K} \nrightarrow seq(I)) \tag{33}$$
$$\wedge$$
$$entity\_traces(s \leftarrow rc, \ T') \tag{34}$$

The conjunct (33) is a typing theorem which is easy to show; its proof is omitted. To prove (34), we need the definition of predicate *entity_traces* used in hypothesis (26). Let $|||\, f$ denote the interleaving of the images of function f, i.e., $|||\, f \triangleq |||\, x : x \in dom(f) : f(x)$, let αe_i denote the set of event labels used in entity type e_i, let $\Delta = \alpha e_1 \cap \alpha e_2$, let $P \overset{s}{\rightsquigarrow} P'$ denote that process expression P can execute trace s and transform into process expression P'. We simply use $P \overset{s}{\rightsquigarrow}$ to denote that P can execute s. Finally, let process expression $\boxdot$ denote execution completion; $\boxdot$ cannot execute any input. In the sequel, for the sake of simplicity, we consider that a trace (or a sequence of inputs) is also a process expression.

$$entity\_traces(s, T) \ \Leftrightarrow$$
$$\forall\, x \in dom(T(e_1)) : e_1(x) \overset{T(e_1)(x)}{\rightsquigarrow} \ \wedge \tag{35}$$
$$\forall\, x \in dom(T(e_2)) : e_2(x) \overset{T(e_2)(x)}{\rightsquigarrow} \ \wedge \tag{36}$$
$$|||\, T(e_1) \ |[\Delta]| \ |||\, T(e_2) \overset{s}{\rightsquigarrow} \boxdot \tag{37}$$

The first two conjuncts state that each entity trace can be executed by its entity. The last conjunct states that the combination of the entity traces can execute s (i.e., s is a combination of the entity traces). Note that we use operator $|[\Delta]|$ from Lotos, which is parallel composition with synchronization over elements of Δ. We refer the reader to [FSD99b] for more details.

We may now prove (34). If we substitute *entity_traces* by its definition in (34), we obtain the following formula to prove.

$$\forall\, x \in dom(T'(e_1)) : e_1(x) \overset{T'(e_1)(x)}{\rightsquigarrow} \ \wedge \tag{38}$$
$$\forall\, x \in dom(T'(e_2)) : e_2(x) \overset{T'(e_2)(x)}{\rightsquigarrow} \ \wedge \tag{39}$$
$$|||\, T'(e_1) \ |[\Delta]| \ |||\, T'(e_2) \overset{s \leftarrow rc}{\rightsquigarrow} \boxdot \tag{40}$$

The proof of (38) is as follows. First, note that entity $e_1(c_p)$ can execute rc. Hence we have $e_1(c_p) \overset{rc}{\rightsquigarrow}$.

$$\begin{array}{ll}
& (35) \ \wedge \ e_1(c_p) \overset{rc}{\rightsquigarrow} \\
\Leftrightarrow & \quad \langle \text{ replace (35) } \rangle \\
& (\forall\, x \in dom(T(e_1)) : e_1(x) \overset{T(e_1)(x)}{\rightsquigarrow}) \ \wedge \ e_1(c_p) \overset{rc}{\rightsquigarrow} \\
\Leftrightarrow & \quad \langle \text{ by definition of } T' : \ rc = T'(e_1)(c_p) \ \rangle \\
& (\forall\, x \in dom(T(e_1)) : e_1(x) \overset{T(e_1)(x)}{\rightsquigarrow}) \ \wedge \ e_1(c_p) \overset{T'(e_1)(c_p)}{\rightsquigarrow} \\
\Leftrightarrow & \quad \langle \text{ by definition of } T' \text{ and (32)} : T(e_1)(x) = T'(e_1)(x) \text{ for } x \in dom(T) \ \rangle \\
& (\forall\, x \in dom(T(e_1)) : e_1(x) \overset{T'(e_1)(x)}{\rightsquigarrow}) \ \wedge \ e_1(c_p) \overset{T'(e_1)(c_p)}{\rightsquigarrow} \\
\Leftrightarrow & \quad \langle \text{ predicate calculus : merge two conjuncts } \rangle \\
& \forall\, x \in dom(T(e_1) \cup \{c_p\}) : e_1(x) \overset{T'(e_1)(x)}{\rightsquigarrow} \\
\Leftrightarrow & \quad \langle \text{ by definition of } T' : \ dom(T'(e_1)) = dom(T(e_1)) \cup \{c_p\} \ \rangle \\
& \forall\, x \in dom(T'(e_1)) : e_1(x) \overset{T'(e_1)(x)}{\rightsquigarrow}
\end{array}$$

□

The proof (39) is straightforward from hypothesis (36) since $T'(e_2) = T(e_2)$ by definition of T'. The proof of (40) is as follows.

$$
\begin{array}{ll}
 & |||\, T(e_1) \;\; |[\Delta]| \;\; |||\, T(e_2) \overset{s}{\rightsquigarrow} \boxdot \\
\Rightarrow & \qquad \langle \text{ process algebra laws : add } rc \;\; \rangle \\
 & (|||\, T(e_1) \;\; |[\Delta]| \;\; |||\, T(e_2)) \centerdot rc \overset{s \leftarrow rc}{\rightsquigarrow} \boxdot \\
\Rightarrow & \qquad \langle \text{ process algebra laws : push } rc \text{ on the left hand side of } |[\Delta]| \text{ since } label(rc) \notin \Delta \;\; \rangle \\
 & (|||\, T(e_1)) \centerdot rc \;\; |[\Delta]| \;\; |||\, T(e_2) \overset{s \leftarrow rc}{\rightsquigarrow} \boxdot \\
\Rightarrow & \qquad \langle \text{ process algebra laws : replace } \centerdot \text{ by } ||| \;\; \rangle \\
 & ((|||\, T(e_1)) |||\, rc) \;\; |[\Delta]| \;\; |||\, T(e_2) \overset{s \leftarrow rc}{\rightsquigarrow} \boxdot \\
\Leftrightarrow & \qquad \langle \; c_p \notin dom(T(e_1)) \;\; \rangle \\
 & |||\, (T(e_1) \lhd \{(c_p, rc)\}) \;\; |[\Delta]| \;\; |||\, T(e_2) \overset{s \leftarrow rc}{\rightsquigarrow} \boxdot \\
\Leftrightarrow & \qquad \langle \text{ by definition of } T',\; T'(e_1) = T(e_1) \lhd \{(c_p, rc)\} \text{ and } T'(e_2) = T(e_2) \;\; \rangle \\
 & |||\, (T'(e_1)) \;\; |[\Delta]| \;\; |||\, T'(e_2) \overset{s \leftarrow rc}{\rightsquigarrow} \boxdot
\end{array}
$$

□

4.4 The Second Refinement Step

In this step we refine entity traces by the state variables that correspond to the analysis class diagram.

4.4.1 From VideoClubIntermediate to VideoClubAnalysisModel

The gluing invariant, added to the invariant of the machine $VideoClubAnalysisModel$ and described below, explicites the data refinement from entity traces to more concrete state variables.

$$
\begin{array}{rcl}
customer_{sv} & = & \{cu \mid \; activeCustomer(T, cu)\} \;\; \wedge \\
name_{sv} & = & \lambda\, cu \cdot (activeCustomer(T, cu) \mid cName(T, cu)) \;\; \wedge \\
cassette_{sv} & = & \{ca \mid \; activeCassette(T, ca)\} \;\; \wedge \\
title_{sv} & = & \lambda\, ca \cdot (activeCassette(T, ca) \mid cTitle(T, ca)) \;\; \wedge \\
loanCnt_{sv} & = & \lambda\, ca \cdot (activeCassette(T, ca) \mid nbLoans(T, ca)) \;\; \wedge \\
loan_{sv} & = & \lambda\, ca \cdot (activeCassette(T, ca) \;\; \wedge \;\; onLoan(T, ca) \mid borrower(T, ca))
\end{array}
$$

Each variable corresponding to a class of the analysis class diagram (e.g., $customer_{sv}$ and $cassette_{sv}$) is related to the trace of the corresponding entity of the requirements class diagram. Intuitively, this relationship is determined as the set of objects which have been created and not yet deleted. A variable related to an association (e.g., $loan_{sv}$) is more complex to determine because we need to define the multiplicity. No general rule can be stated; one has to study the different events that act on the association. Attributes of a class (e.g., $name_{sv}$, $title_{sv}$, $loanCnt_{sv}$) are defined only if they are needed to compute the output events or to define a precondition of an operation. To determine attributes, we consider the behavior of these events in the EB^3 specification.

The gluing invariant can also guide the designer in validating operation preconditions and postconditions in the analysis model. The preconditions in machine $VideoClubAnalysisModel$ are just a rewriting of those of in machine $VideoClubIntermediate$, taking into account the relationship identified in the gluing invariant. For instance, consider the precondition of operation **LendCassette**. The conjunct $customer_p \in customer_{sv}$ corresponds to $activeCustomer(T, customer_p)$, and this relationship is defined in the gluing invariant. It is the same reasoning for the substitutions of operations corresponding to output events. For input events, the substitutions has to be discovered: an append to a trace must be reflected by proper updates of state variables. The refinement proof ensures that the substitutions are valid. In fact, the structure of the gluing invariant is the basis of the derivation of an analysis model from a requirements model. We believe this derivation could not be entirely automated, but a set of precise rules could be elaborated, the correctness of which could only be obtain by the refinement proofs.

4.4.2 Proving the Refinement of VideoClubIntermediate by VideoClubAnalysisModel

4.4.2.1 Initialisation Let $Init_{AM}$ be the initialisation substitutions of machine VideoClubAnalysis-Model. Let Inv_{AM} be the invariant of machine VideoClubAnalysisModel. We have to prove the following

statement: $[Init_{AM}][Init_{Inter}]Inv_{AM}$. To show the principles of the proof, we consider only the variables $customer_{sv}$ and $name_{sv}$ and the related invariant. Applying the substitutions, we obtain:

$$\begin{array}{l}\{\} \subseteq CUSTOMER \ \wedge \\ \{\} \in \{\} \rightarrow STRING \ \wedge \\ \{\} = \{cu \mid activeCustomer(\{\}, cu)\} \ \wedge \\ \{\} = \lambda\, cu \cdot (activeCustomer(\{\}, cu) \mid\ cName(\{\}, cu))\end{array}$$

This formula trivially holds, since $activeCustomer(T, cu)$ is false for $T = \{\}$.

4.4.2.2 Operation RegisterCustomer

We use the following abbreviations:

- Pre_{AM} : precondition predicates of operation **RegisterCustomer** in machine VideoClubAnalysis-Model;
- S_{AM} : THEN substitutions of operation **RegisterCustomer** in machine VideoClubAnalysisModel;
- Inv_{AM} : invariant of machine VideoClubAnalysisModel.

We have to prove the following statement:

$$\begin{array}{l}Inv_{Inter} \wedge Inv_{AM} \wedge Pre_{Inter} \\ \Rightarrow \\ Pre_{AM} \wedge [S_{AM}][S_{Inter}]Inv_{AM}\end{array} \tag{41}$$

Assuming the antecedent of (41), we obtain the following hypotheses.

$$customer_{sv} \subseteq CUSTOMER \ \wedge \tag{42}$$
$$customer_{sv} = \{c \mid activeCustomer(T, c)\} \ \wedge \tag{43}$$
$$customer_p \in CUSTOMER \ \wedge \tag{44}$$
$$name_p \in STRING \ \wedge \tag{45}$$
$$customer_p \notin dom(T(\mathbf{customer})) \tag{46}$$

Let us first prove Pre_{AM}. Replacing Pre_{AM} by its definition, we obtain the following statements to prove.

$$customer_p \in CUSTOMER - customer_{sv} \ \wedge \tag{47}$$
$$name_p \in STRING \tag{48}$$

Proof of (47) : by (46), we can deduce that $activeCustomer(T, customer_p)$ is false. Hence, $customer_p \notin customer_{sv}$. From this and (44) we deduce (47).
Proof of (48) : (48) is exactly (45).

We now prove $[S_{AM}][S_{Inter}]Inv_{AM}$. For the sake of concision, we use the same abbreviations as the ones defined in Section 4.3.2.2. In addition, let n_p be $name_p$. Applying the substitutions, we obtain the following statements to prove.

$$customer_{sv} \cup \{c_p\} \subseteq CUSTOMER \ \wedge \tag{49}$$
$$name_{sv} \cup \{(c_p, n_p)\} \ \in \ customer_{sv} \cup \{c_p\} \rightarrow STRING \ \wedge \tag{50}$$
$$customer_{sv} \cup \{c_p\} = \{c \mid activeCustomer(T', c)\} \ \wedge \tag{51}$$
$$name_{sv} \cup \{(c_p, n_p)\} = \lambda\, c \cdot (activeCustomer(T', c) \mid\ cName(T', c)) \tag{52}$$
$$\ldots$$

Other conjuncts of Inv_{AM} are omitted because their proofs are trivial since they are not affected by operation **RegisterCustomer**.

Proof of (49) : It follows from (42) and (44).
Proof of (50) : We have to show that $name_{sv} \cup \{(c_p, n_p)\}$ is still a function. From (46), we can deduce that $c_p \notin customer_{sv}$, thus $c_p \notin dom(name_{sv})$.
Proof of (51). We need two lemmas, (53) and (54), which we now introduce and prove. Let Ψ denote $T(e_1) \lhd\!\!+ \{(c_p, rc)\}$.

$$\{c \mid c = c_p \ \wedge \ c \in dom(\Psi) \ \wedge \ label(last(\Psi(c))) \neq \mathsf{UnregisterCustomer}\} = \{c_p\} \tag{53}$$

$$\{c \mid c \neq c_p \ \wedge \ c \in dom(\Psi) \ \wedge \ label(last(\Psi(c))) \neq \mathsf{UnregisterCustomer}\} = customer_{sv} \tag{54}$$

Proof of (53)

$$\begin{array}{ll} & \{c \mid c = c_p \ \wedge \ c \in dom(\Psi) \ \wedge \ label(last(\Psi(c))) \neq \mathsf{UnregisterCustomer}\} \\ = & \quad \langle \text{ replace } c \text{ by } c_p \ \rangle \\ & \{c \mid c = c_p \ \wedge \ c_p \in dom(\Psi) \ \wedge \ label(last(\Psi(c_p))) \neq \mathsf{UnregisterCustomer}\} \\ = & \quad \langle \text{ remove } 2^{nd} \text{ conjunct since } dom(\Psi) = dom(T(e_1)) \cup \{c_p\} \ \rangle \\ & \{c \mid c = c_p \ \wedge \ label(last(\Psi(c_p))) \neq \mathsf{UnregisterCustomer}\} \\ = & \quad \langle \ \Psi(c_p) = rc \ \rangle \\ & \{c \mid c = c_p \ \wedge \ label(last(rc)) \neq \mathsf{UnregisterCustomer}\} \\ = & \quad \langle \text{ remove } 2^{nd} \text{ conjunct since } label(last(rc)) = \mathsf{RegisterCustomer} \ \rangle \\ & \{c \mid c = c_p\} \\ = & \\ & \{c_p\} \end{array}$$

□

Proof of (54)

$$\begin{array}{ll} & \{c \mid c \neq c_p \ \wedge \ c \in dom(\Psi) \ \wedge \ label(last(\Psi(c))) \neq \mathsf{UnregisterCustomer}\} \\ = & \quad \langle \ c \neq c_p \text{ implies that } \Psi(c) = T(e_1)(c) \ \rangle \\ & \{c \mid c \neq c_p \ \wedge \ c \in dom(\Psi) \ \wedge \ label(last(T(e_1)(c))) \neq \mathsf{UnregisterCustomer}\} \\ = & \quad \langle \text{ by (46) : } c \neq c_p \ \wedge \ c \in dom(\Psi) \ \Leftrightarrow \ c \in dom(T(e_1)) \ \rangle \\ & \{c \mid c \in dom(T(e_1)) \ \wedge \ label(last(T(e_1)(c))) \neq \mathsf{UnregisterCustomer}\} \\ = & \quad \langle \ (43) \text{ and definition of } activeCustomer \ \rangle \\ & customer_{sv} \end{array}$$

□

Finally, here is the proof of (51).

$$\begin{array}{ll} & \{c \mid activeCustomer(T', c)\} \\ = & \quad \langle \text{ definition of } activeCustomer \text{ and } T' \ \rangle \\ & \{c \mid c \in dom(T \lhd\!\!+ \{(e_1, \Psi)\}(e_1)) \ \wedge \ label(last(T \lhd\!\!+ \{(e_1, \Psi)\}(e_1)(c))) \neq \mathsf{UnregisterCustomer}\} \\ = & \quad \langle \ T \lhd\!\!+ \{(e_1, \Psi)\}(e_1) = \Psi \ \rangle \\ & \{c \mid c \in dom(\Psi) \ \wedge \ label(last(\Psi(c))) \neq \mathsf{UnregisterCustomer}\} \\ = & \quad \langle \text{ case analysis using } c = c_p \vee c \neq c_p \ \rangle \\ & \{c \mid c = c_p \ \wedge \ c \in dom(\Psi) \ \wedge \ label(last(\Psi(c))) \neq \mathsf{UnregisterCustomer}\} \\ & \cup \\ & \{c \mid c \neq c_p \ \wedge \ c \in dom(\Psi) \ \wedge \ label(last(\Psi(c))) \neq \mathsf{UnregisterCustomer}\} \\ = & \quad \langle \text{ lemmas (53), (54) } \rangle \\ & \{c_p\} \cup customer_{sv} \end{array}$$

□

The proof of (52) is similar to the proof of (51).

5 Conclusion

In this paper we have proposed a strategy to prove that an analysis model is correct with respect to a requirements model within the context of a UML-based software development process. Both models are described using graphical notations (UML) in order to be end-user understandable and formal notations (EB^3 and B) in order to write complete and precise specifications. The proof is conducted using the B refinement theory.

The requirements model is orthogonal to the analysis model; that is a powerful methodological principle of software validation. While the EB^3 specification represents the scenario view, the B specification represents the state-transition view. Defining the expected behavior of a system in two completely different views fosters the identification of errors and omissions: what is easily expressed in one view might have been missed in the other. The proof of correctness makes sure that the two views are the same; hence, it provides a systematic

way of finding inconsistencies. The proof of correctness also provides insight on the connection between traces and state transitions through the gluing invariant. Traditionally, formal approaches to information systems specification mostly paid attention to the analysis model; the requirements model was always stated in plain natural language; hence, no formal validation of the analysis model was possible with respect to the requirements model.

This work shows that process algebras are not only useful for distributed or concurrent systems, but also for information systems. We believe that our approach is best suited for systems involving complex, inter-related data structures and complex scenarios of events. The EB^3 specification makes the scenarios and the behavior more explicit, a feature which should facilitate validation with the users. Scenarios can also be diagrammatically represented using JSD (Jackson System Development) structure diagrams [Jac83] (which are essentially syntax trees of the entity process expressions). The B notation is well-suited for describing information systems. It allows a conceptual representation of complex data structures together with a formal description of the operations that act upon them. Evolution in the user requirements should not affect the requirements model as much as the analysis model, because EB^3 scenarios are independent, whereas the structure of the classes in the analysis model may be modified.

To improve our approach and make it easier to apply, we envisage the development of refinement heuristics that would generate a skeleton of the analysis model from the requirements model. These heuristics would be inspired from the derivation of the refinement proof strategy and the gluing invariant. The idea is to automatically handle mundane analysis steps and let the software engineer concentrate on more creative steps. For instance, the classes, attributes and some operations of the analysis model could be automatically generated, while the more complex operation preconditions and postconditions would be left to the software engineer. These heuristics could stem from patterns. Software development is a repeat business with some customization. The EB^3 notation seem to be a good candidate to abstractly represent patterns. Its compositionality seems adequate to represent similarities in behavior without making internal design choices.

We also consider the development of tools to support proof development. For industrial-size examples, tools support is mandatory. The challenge is to merge the theory of process algebra with the theory of model-based specification like B.

Finally, the use of a diagrammatic notation like UML is necessary and inescapable in modern software engineering. Most engineering disciplines use diagrams. Textbooks in mathematics and physics use diagrams. The issue is that software engineering cannot get away with diagrams only, as much as it cannot get away with mathematics only.

References

[Abr96] Abrial, J.-R.: *The B-Book*. Cambridge University Press, Cambridge, UK, 1996.

[B-core] B-Core Limited: Oxford, United Kingdom, `http://www.b-core.com`

[B98] Bert, D. (Ed.): B'98: *Recent Advances in the Development and Use of the B Method*, Second International B Conference, Montpellier, France, Springer-Verlag, LNCS 1393, April 1998.

[EK99] Evans, A., Kent, S.: Core Meta-Modelling Semantics of UML: The pUML Approach. In 2nd International Conference on the Unified Modeling Language. Editors: B.Rumpe and R.B. France, Colorado, LNCS 1723, 1999.

[BB87] Bolognesi, T. and Brinksma, E.: Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, **14**(1):25–59, 1987.

[But00] Butler, M.: csp2B: A Practical Approach to Combining CSP and B. *Formal Aspects of Computing*, **12**(4):182–198, 2000.

[CR01] Cavarra, A., Riccobene, E.: Simulating UML Statecharts. In *Formal Methods and Tools for Computer Science, Eurocast 2001*, Universidad de Las Palmas de Gran Canaria, February 2001, isbn 84-699-3971-8.

[ClearSy] CLEARSY System Engineering: Aix-en-Provence, France, `http://www.clearsy.com/`

[CAB94] Coleman, D., Arnold, P., Bodok, S., Dollin, C., Gilchrist, H., Hayes, F., Jeremaes, P. : *Object-Oriented Development: The Fusion Method*. Prentice Hall, Englewood Cliffs, NJ, Object-Oriented Series edition, 1994.

[DLC00] Dupuy, S., Ledru, Y., Chabre-Peccoud, M.: Overview of Roz: A tool for integrating UML and Z specifications. In *CAiSE00: 12th International Conference on Advanced Information Systems Engineering*, Stockholm, Sweden, Springer-Verlag, LNCS 1789, June 2000.

[FLN+99] Facon, P., Laleau, R., Nguyen, H.P., Mammar, A.: Combining UML with the B Formal Method for the Specification of Databases Applications. Technical Report, Laboratoire Cedric, Paris, France, September 1999.

[Fis97] Fischer, C.: CSP-OZ: A combination of Object-Z and CSP. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, 423–438, Chapman & Hall, 1997.

[FSD99a] Frappier, M., St-Denis, R.: Combining JSD and Cleanroom for Object-Oriented Scenario Specification. In *Object-Oriented Behavioral Specifications*, H. Kilov, B. Rumpe, I. Simmonds, eds., Kluwer Academic Publishers, 1999.

[FSD99b] Frappier, M., St-Denis, R.: Specifying Information Systems through Structured Input-Output Traces, Technical Report, Département de mathéematiques et d'informatique, Université de Sherbrooke, Sherbrooke (Québec), Canada J1K 2R1, 1999.

[GSW93] Günther, T., Schewe, K.D., Wetzel, I.: On the derivation of executable database programs from formal specifications. In *FME93: First International Symposium of Formal Methods Europe*, Odense, Denmark, Springer-Verlag, LNCS 670, April 1993.

[Hoa85] Hoare, C. A. R.: *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, 1985.

[Jac83] Jackson, M.: *System Development*. Prentice Hall, Englewood Cliffs, 1983.

[LM00a] Laleau, R. Mammar, A.: A Generic Process to Refine a B Specification into a Relational Database Implementation. In *ZB 2000: Formal Specification and Development in Z and B*, First International Conference of B and Z Users, York, UK, Springer-Verlag, LNCS 1393, September 2000, 22–41.

[LM00b] Laleau, R. Mammar, A.: 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.

[LP01a] Laleau, R., Polack, F.A.C: A Rigorous Metamodel for UML Static Conceptual Modelling of Information Systems. *CAISE: 13th International Conference on Advanced Information Systems Engineering*, Interlaken, Suisse, Springer-Verlag, LNCS 2068, June 2001.

[LP01b] Laleau, R. Polack, F.A.C.: Specification of integrity-preserving operations in information systems by using a formal UML-based language. *Information and Software Technology*, to appear.

[MP95] Mander, K. C., Polack F.A.C.: Rigorous Specification Using Structured Systems Analysis and Z. *Information and Software Technology*, **37**(5–6):285–291, 1995.

[ML99] Matthews, B., Locuratolo, E.: Formal development of databases in ASSO and B. In *FM99: World Congress on Formal Methods*, Toulouse, France, Springer-Verlag, LNCS 1709, September 1999, 388–410.

[Mil89] Milner, R.: *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, 1989.

[PWM93] Polack, F.A.C., Whiston, M., Mander, K.: The SAZ project: Integrating SSADM and Z. In *FME93: First International Symposium of Formal Methods Europe*, Odense, Denmark, Springer-Verlag, LNCS 670, April 1993.

[SFD92] Semmens, L.T., France, R.B., Docker, T.W.G.: Integrated structured analysis and formal specification techniques. *The Computer Journal*, **35**(6):600–610, 1992.

[SCH01] Shen, W., Compton, K., Huggins, J.: A Validation Method for a UML Model Based on Abstract State Machine. In *Formal Methods and Tools for Computer Science, Eurocast 2001*, Universidad de Las Palmas de Gran Canaria, February 2001, isbn 84-699-3971-8.

[SN01] Snook, C., Harrison, R.: Practitioners' views on the use of formal methods: an industrial survey by structured interview. *Information and Software Technology* **43**:275–283, 2001.