ISSN 1361 - 6161

THE UNIVERSITY of MANCHESTER

Computer Science
University of Manchester

A Proposal for a LOTOS-Based Semantics for UML

Paulo Pinheiro da Silva

Department of Computer Science

University of Manchester

Technical Report Series

UMCS-01-06-1

A Proposal for a LOTOS-Based Semantics for UML*

Paulo Pinheiro da Silva

Department of Computer Science
University of Manchester
Oxford Road, Manchester, UK.
pinheirp@cs.man.ac.uk

*Refereed by: Enrico Franconi

Abstract

The Unified Modeling Language (UML) is a popular visual language for object-oriented modelling of software systems. Despite this popularity, the use of UML to support the development of software systems is challenging for many reasons such as its lack of a formal semantics. As a result, the interpretation of models required to carry out the implementation of a software system may be performed in different and contradictory ways. There have been many attempts to develop a formal semantics for UML. However, most of these formalise some group of constructors of UML responsible for modelling either structural or behavioural aspects of the software systems. Indeed, most of the previous work seem to face some difficulties specifying all the models of UML using a single formal notation. The Language Of Temporal Ordering Specification (LOTOS) is a formal specification language that provides operators for specifying structural and behavioural aspects of software systems. Thus, this paper describes a function that maps UML constructors into LOTOS specifications. Typical model checking of activity diagrams identifying deadlocks, livelocks and unreachable states is achieved using available LOTOS tools to verify the generated LOTOS specifications. Further, the identification of semantic problems related to complex relationships between class and activity diagrams, e.g., creation and destruction of inter-dependent objects, is also achieved, verifying the generated specifications.

1 Introduction

UML [4] is widely used for object-oriented modelling of software systems. Further, UML is an official standard that benefits from the endorsement of the Object Management Group (OMG) [27]. The popularity of UML has indicated that the language may be suitable for describing software systems at an unprescribed though relevant extent. Moreover, in the same way as there are many reports describing successful uses of UML for modelling systems, there are many reports identifying difficulties in the use of UML for modelling, for example, interactive systems [31] and real-time systems [36]. The lack of a formal semantics is another identified difficulty related to the use of UML [5, 7, 8, 24]. Indeed, this lack of a formal semantics indicates the possibility of ambiguous interpretations of UML models. For instance, it indicates that people using UML may have different and contradictory interpretations of a same set of models. This may lead to disputes over the interpretation of the models, and over the implementation of systems that do not fulfill the intentions of the designers. A formal semantics for UML could solve this problem of contradictory interpretations of UML models. Further, such a semantics could be useful for implementing automated verification of models to identify incorrect uses of the notation, as well automated interpretation of models to generate software code.

The question of how best to provide a formal description of the semantics of UML is still open. For example, a specification of UML in terms of a mathematical notation or a formal specification language could provide a semantics for UML. The expressiveness gap between UML and a mathematical notation may be bigger than between UML and a formal specification language. Indeed, the semantics of specification languages are often provided by their mathematical specifications. Thus, the approach in this paper is based on the use of a formal specification language. Such a specification language should be powerful enough to specify the behavioural and structural aspects that can be described by UML models. In fact, UML is composed of a set of diagrams that mainly describe behavioural (also called dynamic) characteristics of software systems, e.g., sequence and activity diagrams, and a set of diagrams that mainly describe structural (also called static) characteristics of software systems, e.g., class and deployment diagrams. Most formal specification languages provide facilities for modelling and verifying behavioural and structural aspects of software systems. Some formal specification languages, e.g., Z [37], are appropriate while describing structural aspects of software systems. However, it may be difficult to check some behavioural properties such as concurrency of their specifications since these languages do not describe any computation that explains how their specifications can be executed [10]. A combination of specification languages could be considered, e.g., Z [37], CCS [25] and CSP [14]. However, it would be most desirable to use just one specification language to provide the required formalism.

In this paper, a LOTOS [3, 16] approach for specifying the semantics of UML is presented. LOTOS is a specification language that has succeeded in the challenging task of describing structural and behavioural aspects of software systems using a single notation. Indeed, LOTOS has incorporated the specification facilities of CCS and CSP, as well the facilities for specifying the abstract data types of ACT-ONE [6]. Moreover, LOTOS is an International Standardization Organization (ISO) standard specification language developed for the formal description of the Open System Interconnection (OSI) architecture that is applicable to distributed, concurrent systems in general. Thus, through a case study it is described how LOTOS can be used to specify a semantics for representative class and activity diagram constructors. Furthermore, object flows described in this paper and used in activity diagrams provide a connection between structural models, e.g., class diagrams, and behavioural models, e.g., activity diagrams without object flows.

The reading of this paper may flow in a straightforward way for readers that have familiarity with (i) the specification of the UML metamodel as described in [27], and (ii) the LOTOS notation [16]. An introductory section on the LOTOS notation along with some explanations concerning the UML metamodel presented throughout the paper try to overcome the requirements (i) and (ii) above. Thus, this paper is structured as follows. Section 2 presents an insight about the current work on specifying a semantics for UML. Section 3 provides a brief introduction to LOTOS. This section can be skipped by readers familiar with LOTOS. Section 4 describes the modelling of a software system using UML. Section 5 introduces the Φ function that translates UML models into LOTOS specifications, building on the UML metamodel. The UML model introduced in Section 4 exemplifies the use of Φ to generate LOTOS specifications. Section 6 describes the Φ function for some structural constructors of UML. Section 7 describes the Φ function for some behavioural constructors of UML. Section 8 describes the use of a LOTOS verification tool to analyse the LOTOS specification produced from the UML models introduced in Section 4. Conclusions are presented in Section 9.

2 Related Work

The efforts to provide a formal semantics for the UML can be classified in many ways. In this paper, related work is presented emphasising the existing dichotomy between approaches formalising structural and behavioural aspects of the UML.

There are many approaches to formalising structural aspects of UML. Evans et al. [7] is an example of one of the two approaches of *the precise UML group* [38] of researchers concerned about the lack of a semantics for UML. In this approach, a semantics for UML is expected to be achieved by the formalisation of some class diagram constructors used to build the *UML metamodel*, as described in [27]. Therefore, the semantics of the other constructors of UML can follow from the previously formalised constructors, specifying in this way a semantics for the entire UML. Particularly in Evans et al. [7], Z [37] is used to formalise the class diagram. In Evans and Kent [8], the use of set theory embedded in Object Constraint Language (OCL) [40] constraints is used to provide a semantics for the generalisation and package concepts. The Action Semantics proposal originated by Mellor et al.[24] also aims to achieve a formalisation of the OCL. Further, Richters and Gogolla [33] have proposed a formalisation of the OCL in an integrated way with some constructors of class diagrams. [8, 33] are examples of the second approach of the precise UML group where a semantics for UML is expected to be achieved by the formalisation of the OCL.

There is much work on formalising behavioural aspects of UML. The first mention in this context should probably be of David Harel's work which has influenced the development of the UML [11]. Considering this, we can say that the Harel et al. [13] and Harel and Naamad's [12] descriptions of the statechart semantics are partial descriptions of the semantics of UML. There are other results on a formalisation of the behavioural aspects of UML. For instance, one of the final aims of having a formal specification is the possibility of verifying if a model is correct. Latella et al. [21] and Lilius and Paltor [22] present robust work that describes how statecharts can be verified. Both of them are based on the use of the SPIN model checker [15]. Moreover, Latella et al. [21] indicates several points where the informal specification of UML is silent in terms of a proper specification.

There is a concern about this dichotomy between the distinct formalisation of the structural and dynamical aspects of UML. Wang et al. [39] describes a formalisation of the dynamic models of the

OMT [35]. This work is considered in this paper since the OMT is one of the three major predecessors of the UML, and its formalisation is based in LOTOS. More recently, Breu et al. [5] have proposed another use of mathematical models, denoted *system models*, to describe most of the constructors of UML. It looks like a promising approach for a mathematically-grounded semantics for UML. The approach, however, is currently a long way from being complete enough to provide a verification facility for UML. This dichotomy between the formalisation strategies of the dynamic and structural parts of UML models may be a problem for a complete verification of UML models. Keeping the semantics tractable at a certain level, as described in Latella et al. [21], may be an appropriate strategy for the verification of UML models. However, dynamic and structural models are interdependent, and as such they might be verified at once.

This paper has been motivated in particular by the lack of semantics and even notation for modelling interactive systems using UML [20, 30], and the formal specifications of interactive systems using LOTOS [28, 23] has provided a strong motivation for the proposal described here.

3 A Brief Introduction to LOTOS

3.1 Basic LOTOS

A system $\mathcal{S}$ can be specified by a LOTOS process that might be composed of other LOTOS processes. A LOTOS process $P[G]$ has a set of observable gates $G = \{g_1, g_2, ..., g_n\}$. LOTOS assumes that an environment associated with the process P exists that is composed of the process P, its subprocesses, and an unspecified *observer* process that is always ready to observe anything the system $\mathcal{S}$ may do. Thus, a LOTOS *action* can be defined as the interaction between a defined process and, at least, the *observer* process. The behaviour of a process is defined by an algebraic expression composed of: actions that may be observed at the gates (*unary* operators); internal actions that cannot be observed at any gate (*nullary* operators); and of other processes that specify their own algebraic expressions (composed operators). All these operators are connected by binary operators, as presented in Table 1. These algebraic expressions are called *behaviour expressions*.

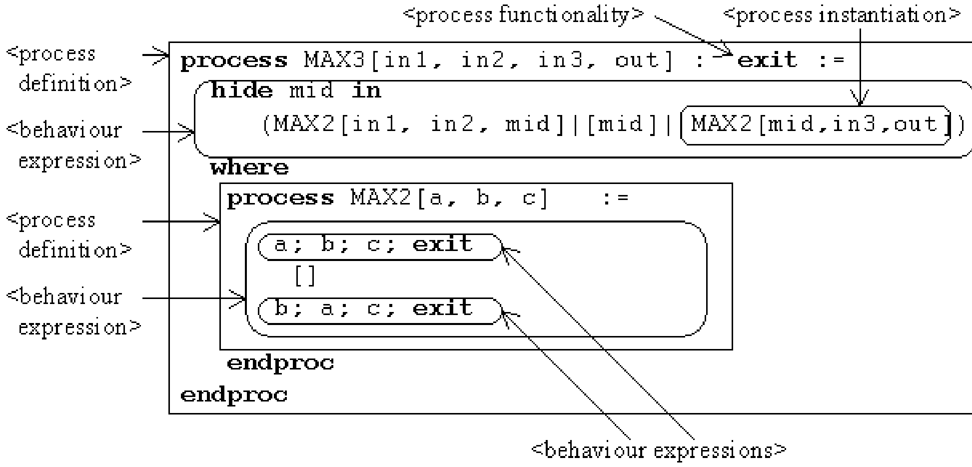


Figure 1: Definition of a LOTOS process extracted from [3].

Figure 1 shows an example of a LOTOS process definition. There, the `MAX3` process is defined that has `MAX2` as a subprocess. The observable gates of `MAX3` are `in1`, `in2`, `in3` and `out`, and the observable gates of `MAX2` are `a`, `b` and `c`. The behaviour of `MAX3` is defined by the behaviour of the two instances of the `MAX2` process that are synchronised on the `mid` gate, as specified by the interleaving binary operation (|[]|) connecting the two instantiations of `MAX2`. Further, an operation that is expected to terminate has

Category	Operator	Notation	Description
Binary Operators	Action prefix	;	The expression $a; C$ means that the process behaves like C after the execution of a.
	Enabling	>>	The expression $C >> D$ means that the process D is enabled if, and only if, C terminates successfully.
	Choice	[]	The expression $a; C[]b; D$ means that the process can start to behave like either C or D depending on the next actions provided by the interaction of the current process with its environment. If the environment offers an action a then the process starts to behave like C. If the environment offers an action b, then the process starts to behave like D.
	Interleaving	|||	The expression $C\|\|\|D$ means that the actions of the processes C and D do not need to synchronise (completely independent).
	Interleaving with synchronisation gates	|[]|	if a is an observable action in both C and D processes, then the expression $C\|[a]\|D$ means that C and D must synchronise in order to perform a.
	Hiding	**hide** ... **in**	The expression **hide** x **in** C means that the action x originally specified as observable by any other process interacting with the current process at the gate x now can only be observable from the process C.
	Disabling	[>	The expression $C[> D$ means that the process C can be interrupted any time before its successful termination in the case that the environment provides an unspecified interrupting action called a *disabling operator*. In this case, the process starts to behave like D.
Unary Operators	stop	**stop**	This operator, which offers no event to the environment, means that the process becomes inactive.
	exit	**exit**	This operator offers an special event to the environment notifying the successful termination of the process. After raising this special event, the process also becomes inactive as a result of the **stop** operator.

Table 1: Binary and especial unary operators of LOTOS. It is assumed that a and b are LOTOS observable actions and that C and D are LOTOS processes in the descriptions.

the **exit** functionality, as in the MAX3 process specification. Otherwise, the process can have a **noexit** functionality. As a LOTOS convention, action names are written in lowercase letters and process names are written in uppercase letters. To facilitate the reading of this paper, words in bold fonts are reserved words of LOTOS.

3.2 Full LOTOS

The LOTOS presented so far is *basic LOTOS*, since only behavioural aspects can be specified. The specification of observable actions, however, can be refined with the use of abstract data structures and values. Thus, the abstract data type specification language ACT-ONE [6] has been integrated with LOTOS, specifying *full LOTOS*. New interprocess communications can be achieved in *full LOTOS*.

- Value passing: Suppose that the processes C and D are synchronised on the gate x, e.g. $C|[x]|D$. Moreover, suppose that the process C is performing $x!TRUE$ and the process D is performing $x?b$:**Bool**. We can say that C is passing the value $TRUE$ to the process D. Moreover, this $TRUE$ value is assigned to the b variable of the process D.

- Signal matching: Once again, suppose $C|[x]|D$. This time, suppose that C is performing $x!z_1$ and D is performing $x!z_2$. This means that C and D will only be synchronised if the values of z_1 and z_2 become the same ($z_1 = z_2$).

The type of the b variable in the *value passing* example is **Bool** (that means, *Boolean*). *Full* LOTOS, or just LOTOS, provides a set of primitive types for modelling simple data structures. For instance, the LOTOS primitive **nat** denotes a type the domain of which must be a natural number. LOTOS also provides the ability to specify more complex data structures composed of primitive types and other complex data structures. For example, the PERSON type definition presented as follows describes a type that might be used by objects of a class PERSON.

```
type PERSON is String
   sorts Person
   opns   mk_Person :                      -> Person
          mk_Person2: String , String -> Person
          setname: Person , String       -> Person
          getname: Person                -> String
          setcode: Person , String       -> Person
          getcode: Person                -> String
   eqns   forall name1, name2, code1 , code2: String
          ofsort String
            getname(mk_Person2(name1, code1)) = name1;
            getcode(mk_Person2(name1, code1)) = code1;
          ofsort Person
            setname(mk_Person2(name2, code2), name1) = mk_Person2(name1, code2);
            setcode (mk_Person2(name2, code2), code1 ) = mk_Person2(name2, code1);
endtype
```

Figure 2: `Person` type specification.

The `PERSON` in Figure 2 is a type specification. The **String** after the **is** indicates that `PERSON` incorporates the specification of the **String** type. If required, other type specifications could be incorporated along with the **String** type. Further, the **eqns** operator indicates equations used to specify constraints relating intrinsic operations. Equations can be complex since they can specify complex constraints. However, only a set of simple equations required to provide meaningful type specifications as those presented in Figure 2 are considered in this paper. Table 2 provides a brief explanation of the type operators in Figure 2.

Operator	Description
sorts	Specifies *data carriers* of a type.
opns	Specifies the intrinsic operations that can be performed over variables of a type.
eqns	Begins the specification of *equations* where constraints relating intrinsic operations are specified.
forall	Declared under an **eqns** operator, it specifies free variables used in equations.
ofsort	Declared under an **eqns** operator, it specifies the outermost operation in its following equations.

Table 2: Some type operators of LOTOS.

Thus, for a given variable $aPerson$ of the type `Person`, for example, the operation `getname(aPerson)` returns a value of type **String** that is stored in `Person`. From the definition of `getname(aPerson)` it is possible to identify the declarative nature of LOTOS. The specification of the `getname()` operation specifies *what* is wanted rather than *how* the value can be retrieved from `aPerson`. Two special operations are considered in the type specifications presented in this paper. In the case of the type `Person`, the `mk_Person` operation is a default *constructor* that does not require any parameter to create a value of type `Person`. The `mk_Person2` operation is also a constructor, but one that requires the constituent values (attributes) of `Person` to create a composite value of type `Person`. Therefore, using `mk_Person2`, it is possible to see in Figure 2 that, for example, the `getname()` operations can be used to get an attribute of type **String** from `Person`, and the `setname()` operation can be used store a **String** value as an attribute in `Person`. The **String** type is presented as a reserved word in Figure 2 since it is a primitive type of LOTOS.

The LOTOS processes in this paper are built in an incremental way and using mainly the basic constructors of LOTOS. Thus, it may be expected that readers without prior experience of LOTOS can understand the LOTOS notation from the description of LOTOS presented in this section. Nevertheless, Bolognese and Brinksma [3] is a suggested introductory paper on LOTOS.

4 A Case Study

A Library case study is used to exemplify how UML models can be translated into LOTOS specifications. The design introduced in this section is just one of the many possible results of the modelling of a Library System. Further, the set of models presented in this paper is a subset of the models described in [31]. Moreover, in this paper, we are focusing on the meaning of the presented set of UML models rather than on how UML models can be built. A model of process concerns for this case study is provided in [30].

Actors in the Library System are *Librarians* and *Borrowers*. The Library System must guarantee that only registered users can log into the system. Further, the system must guarantee that borrowers can only perform services associated with borrowers, and that librarians can only perform services associated with librarians. This is achieved by the `connect` Activity in the top-level activity diagram of the Library System in Figure 3. This paper presents the part of the Library System design that specifies the services for the Librarians. Librarians use the Library System to manage the book catalogue and the loan records. Librarians need mainly to inform the Library System when books are borrowed, the `borrowBook` Activity, and returned, the `returnBook` Activity. Additionally, the due date in loan records can extend using the `renewLoan` Activity. The relationship between these identified activities is also presented by the top-level activity diagram in Figure 3. There, `borrowBook`, `renewLoan` and `returnBook` are subactivities of the `SelectFunction` Activity. Moreover, `createMainUI` and `mui.selectService()` are ActionStates responsible for the instantiation and use of the `mui:MainUI` object flow, respectively. The `mui` Object is actually the main user interface where librarians can select which services they want to perform.

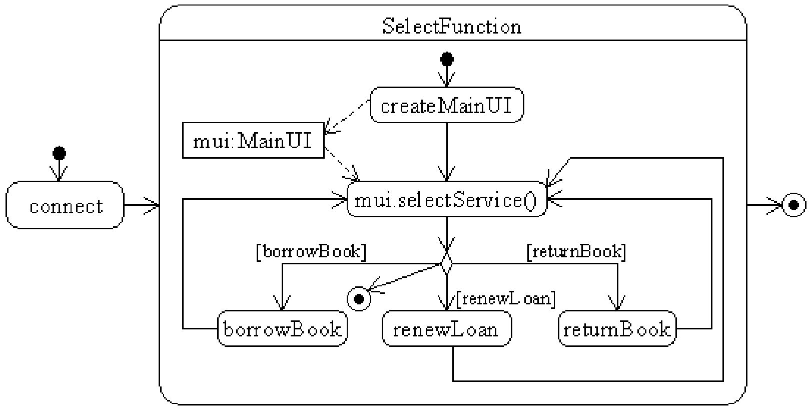


Figure 3: A top activity diagram for the Library System.

Activities can be refined into less abstract Activities and ActionStates. For instance, `connect`, `borrowBook`, `renewLoan` and `returnBook` can be refined since they are Activities. However, `createMainUI` cannot be refined since it is an ActionState. Figure 4 presents a refinement of the `renewLoan` Activity of Figure 3. The `renewLoan` Activity is composed of the `createRenewUI`, `ui.getBookCopy()` and `bc.renewLoan()` ActionStates. Additionally, the activity diagram in Figure 4 specifies the `ui:RenewUI` and `bc:BookCopy` object flows that are modelled using ObjectFlowStates, rendered as dashed arrows, and ClassifierInStates, rendered as boxes. These object flows provide an integration between the structural part of the Library

System, as described by the class diagram in Figure 5, and the behavioural part of the Library System, as described by the activity diagrams presented in Figures 3 and 4.

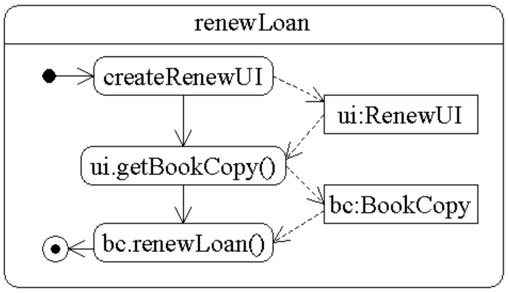


Figure 4: A refinement of the `renewLoan` activity.

The structural specification of the Library System is described by the class diagram shown in Figure 5. This class diagram is composed of the classes `Person`, `Librarian`, `Borrower`, `Book`, `Loan`, `BookCopy`, `Reservation`, `ConnectUI`, `MainUI`, `RenewUI` and `ReturnedCopy`. The first three classes correspond to the `LibrarySystemUser`, `Librarian` and `Borrower` Actors, respectively. A `Person` has a `name`, an identification `code` in the system and a `password` used to connect to the system. The `Librarian` Class is a specialisation of `Person` which has the `salary` Attribute in addition to the Attributes inherited from `Person`. The `Borrower` Class is another specialisation of `Person` which can borrow books from the library. In this case, instances of `Borrower` should be associated to instances of `Loan`.

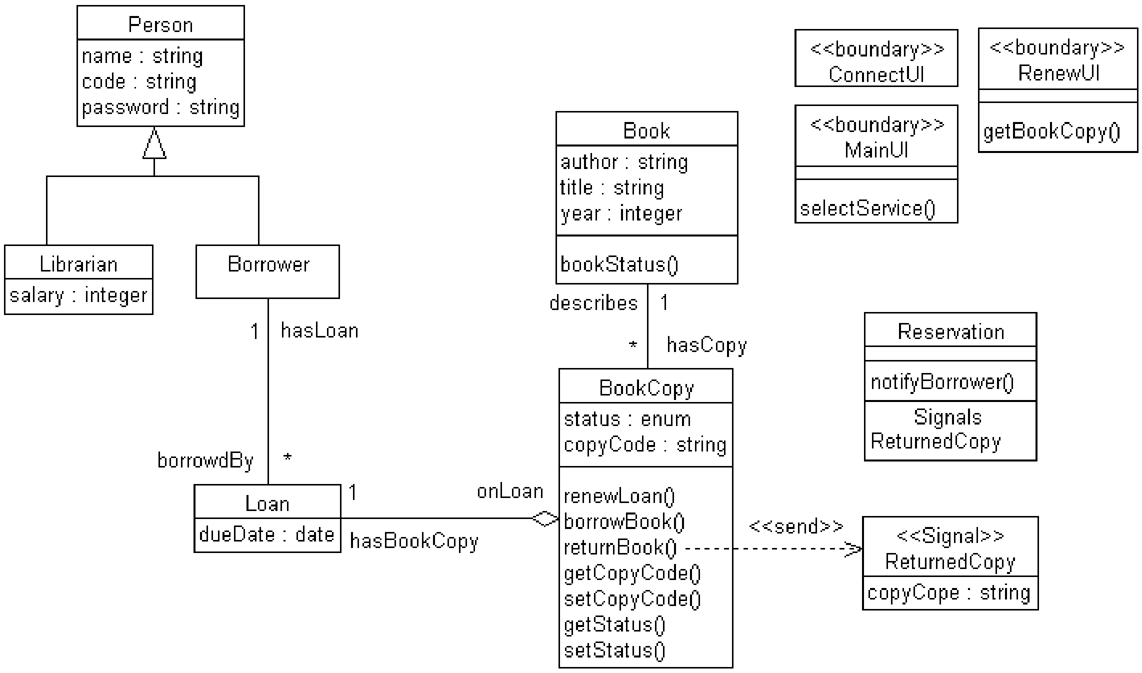


Figure 5: A class diagram for the Library System.

The existence of an instance of `Book` means that the book has an entry in the library catalogue. To manage its stock, the Library System has a `BookCopy` class that represents copies of the books the library has. An instance of `Loan` is created in the `borrowBook` Activity and destroyed in the `returnBook`

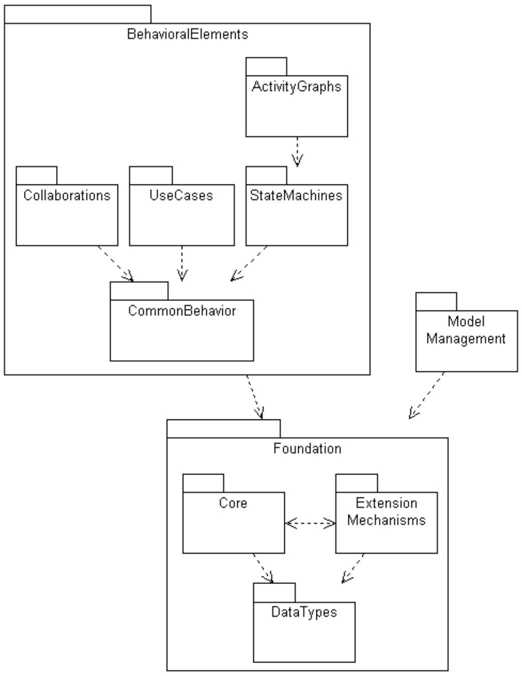


Figure 6: Packages of the UML metamodel. The dashed arrows are dependencies indicating, for example, that at least one class of the `Core` Package depends on a specification of a class of the `DataTypes` Package.

Activity. A `Loan` Object records the date the book should be returned. `ConnectUI`, `MainUI` and `RenewUI` are Classes that describe the structure of the user interface of the Library System. These user interface Classes have the ≪*boundary*≫ stereotype, as proposed by Jacobson et al. [17]. The `ReturnedCopy` Signals are raised by the invocation of the `returnBook` method of `BookCopy`. These `ReturnedCopy` Signals are received by an instance of the `Reservation` Class that notifies `Borrowers` with book reservations.

5 From UML Models to LOTOS Specifications

A semantics for UML models can be provided by LOTOS specifications generated from these UML models. Starting in this section, we explain how UML models can be translated into LOTOS specifications.

5.1 UML Metamodel

UML has been revised several times since the elaboration of its first proposal submitted to the OMG in 1997 [19]. A description of the structural aspects of the UML diagrams is an outcome of these revisions. This description is called the *UML metamodel*, as it is partially composed of UML class diagrams. The complete specification of the UML metamodel is described in the "UML Semantics" chapter in [27], which is organised by the Packages that compose the UML metamodel. Figure 6 presents the UML Packages which are partially organised by both the participation of the metaclasses in the UML diagrams and the dependencies among the metaclasses. Thus, from Figure 6 we can observe that most of the metaclasses of the class diagram are specified in the `Core` Package within the `Foundation` Package, where the structural constructors of UML are specified. Metaclasses of dynamic (or behavioural) diagrams are specified in Packages of the `BehavioralElements`. Therefore, the metaclasses of the collaboration and sequence diagrams are specified in the `Collaborations` Package. The metaclasses of the use case diagram are specified in the `UseCases` Packages. The metaclasses of the state diagram are specified in the

StateMachines Package. The metaclasses of the activity diagram are specified in the ActivityGraphs Package.

For each Package, the documentation provides three informal and complementary descriptions of the metamodel: the *abstract syntax*, *well-formedness rules* and *modelling element semantics*. The class diagrams, that can properly be called the UML metamodel, are part of the abstract syntax. An example of part of one of these class diagrams of the abstract syntax is presented in Figure 7. There, the State constructor specified in the StateMachines Package is a State if used in a statechart, or is an Activity if used in an activity diagram. In both diagrams a State can be connected to another State or to a PseudoState by a Transition. PseudoState and Transition are also specified in the StateMachines Package. The abstract syntax also provides a description in prose of each element that composes the UML metamodel. The well-formedness rules are written in OCL. These rules provide additional constraints concerning the Classes of the abstract syntax that cannot be expressed using just the class diagram constructors. These well-formedness rules are also supported by textual descriptions of their meaning. The semantics of a modelling element is a description, once again in prose, about the meaning of the Package itself.

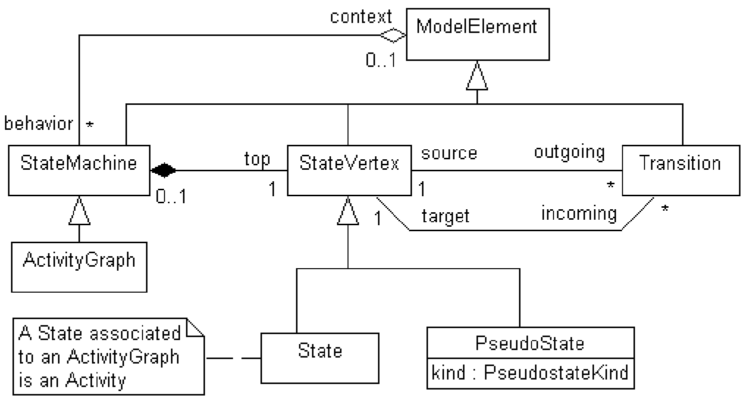


Figure 7: Partial representation of the State Machines and Activity Graphs packages of the UML metamodel.

A UML Constructor is a Class *in a class diagram of the abstract syntax of the UML specification.* In this document, the names of UML constructors are exactly as specified in [27]. Additionally, the UML constructor names are printed using a sans serif font in order to facilitate their identification for readers not familiar with the UML metamodel terminology. For example, the ModelElement, StateVertex, Transition, State and PseudoStates Classes presented in Figure 7 are UML constructors.

The UML metamodel is important since it plays a key role in the development of many UML tools. In fact, the UML metamodel has facilitated the implementation of, e.g., Rational Rose [32] and ARGO/UML [34], for modelling, handling and sharing UML models. However, the UML metamodel aims to be a framework for describing a UML semantics, not only a facility for implementing tools. However, the metamodel basically relies on the use of the English language to describe the meaning of its constructors.

5.2 Mapping Strategy

The strategy of this LOTOS-based proposal for a UML semantics follows the *core meta-modelling* strategy described in [8]. Basically, the idea is to provide an initial semantics for some UML constructors considered essential for modelling most UML diagrams. The semantics specified for these constructors can then be used as a framework for specifying a semantics for the other constructors. For example, a formalisation for Class might be required in order to formalise Package. The following definitions are required to explain how a semantics for such UML constructors can be specified.

LOTOS has a context-free grammar. According to [1], a context-free grammar has a set of *terminal symbols*; a set of *non-terminal symbols*; a set of *productions* where each production is composed of a non-terminal symbol, an arrow, and a sequence of terminals and/or non-terminals; and a designation of one of the non-terminals as the *start symbol*. Thus, a non-terminal symbol of LOTOS is a LOTOS constructor. Assuming that $\mathcal{U}$ is a UML constructor and that $\mathcal{L}$ is a LOTOS constructor, a semantics for UML can be provided by the LOTOS semantics contained in the specification generated through the use of $\Phi(\mathcal{U}) = \mathcal{L}$. The Φ function is specified through the definition of *UML constructor definitions (UCDs)*, which are definitions of UML constructors in terms of sets of at least one LOTOS constructor.

From this UCD we can see that our approach conforms with the official UML approach. In fact, we are respecting the UML specification in the sense that we are neither *modifying* nor *removing* any element of UML. Moreover, we are using the UML specification as a foundation for the proposed semantics for UML.

A basic understanding of the UML metamodel may be required to understand some Φ mappings in the following sections. Conducting a detailed reading of the "UML Semantics" in [27] is heavy going, but, to the best of our knowledge, it is the unique description of the entire UML metamodel.

5.3 Basic Mappings from UML to LOTOS

From this point in the paper we start to introduce a set of 28 UCDs which composes the foundation of a strategy to map UML models into LOTOS specifications. Although limited, this set of mappings indicates that their underlying strategy can be extended to incorporate the other elements of UML not described in this paper.

A system $\mathcal{S}$ can be modelled from a top-level activity diagram, for example, as shown in Figure 3. The Activities in this top-level activity diagram can be recursively decomposed into less abstract Activities and ActionStates connected by Transitions and PseudoStates. The decomposition of the top-level activity diagram is considered complete when the Activities are entirely described in terms of ActionStates, which are the "leaves" of the tree of Activities. Thus, the UML Activity constructor can be specified as follows.

UCD 1 *An* Activity *that has subactivities $Sub_1..Sub_X$ is defined by a LOTOS process definition specified as follows.*

```
process ACTIVITY_ACT[abort] (...) : exit :=
   <final_activity_behaviour>
  where
   process SUB_1_ACT : exit (...) :=
      ...
   endproc
      ...
   process SUB_X_ACT : exit (...) :=
      ...
   endproc
endproc
```

The <final_activity_behaviour> *in the ACTIVITY_ACT process above is a text defined by a BNF grammar specified as follows.*

```
<final_activity_behaviour> ::=
      <activity_behaviour> ``[> abort; exit''
<activity_behaviour> ::= <activity> |
      <activity_behaviour> <operation> <activity>
<activity> ::= <action_state_imp> | <activity_imp>
<action_state_imp> ::= <CALL_SEND_ACTIONSTATE> |
                        <CREATE_ACTIONSTATE>
<activity_imp> ::= <LOTOS PROCESS>
<operation> ::= ``[]'' | ``|||'' | ``>>''
```

In the grammar above: <CALL_SEND_ACTIONSTATE> is a non-terminal specified in UCD 28; <CREATE_ACTIONSTATE> is a non-terminal specified in UCD 24; and <LOTOS PROCESS> is the specification of a LOTOS process instantiation.

Hereafter, underlined words in UCDs represent placeholders which vary according to the instance of the UML constructor represented by the UCD. For example, for an Activity named `SelectFunction` the ACTIVITY_ACT in UCD 1 is replaced to SELECTFUNCTION_ACT.

From UCD 1 we can see that the processes of the subactivities, which can be ActionStates, are defined as subprocesses of the `ACTIVITY` process. Nevertheless, reuse [18] can be achieved in LOTOS specifications of UML models by defining Activities and ActionStates in common higher-level Activities of the Activities that share the same functionality. Still in UCD 1, ACTIVITY_ACT has a standard `abort` gate which is responsible for finishing the process. The `abort` gate allows any process synchronised to it to finish the ACTIVITY_ACT process at any time. Thus, such a gate may be useful for handling abnormal situations such as a premature destruction of an Object or an error message from the operating system.

To conform with the LOTOS specification, an additional UCD should be specified for top-level activities.

UCD 2 *For a top-level* Activity*, the* **process** *and* **endproc** *terminators and the first appearance of the := terminator in UCD 1 are replaced by the* **specification***,* **endspec** *and* **behaviour** *terminators respectively.*

ActionStates and Classes should also be translated into LOTOS specifications in order to describe $\mathcal{S}$ as a LOTOS process. In fact, ActionStates specify the Objects where Actions are performed, and Objects are specified by their Classes. Classes can be specified in term of LOTOS processes, as explained in Section 6. Actions can be specified in terms of LOTOS process as explained in Section 7. However, there are two mapping techniques between $\mathcal{U}$ and $\mathcal{L}$, which we are calling *foundation mappings*, that are required to explain the mappings described in Sections 6 and 7. The first foundation mapping technique explains how the connections provided by Transitions and PseudoStates in activity and statechart diagrams can be translated into binary operators of LOTOS. In fact, the binary operators of LOTOS are used to compose the behaviour expressions of processes. The second foundation mapping technique explains how types, implicitly specified by Classes in UML, can be specified by LOTOS *primitive types* and *type specifications*.

Concerning the behaviour of $\mathcal{S}$, a translation of interaction diagrams, viz. *sequence* and *collaboration* diagrams, into LOTOS specifications is not described in this paper. Indeed, despite the fact that interaction diagrams may be more useful for developers using UML than activity and statechart diagrams, we can observe that interaction diagrams are partial representations of activity and statechart diagrams [29]. Nevertheless, the semantics provided in this paper for constructors used in both activity and statechart diagrams can be used for specifying a semantics for constructors used to build interaction diagrams.

Transitions and Pseudo-States

Activity and statechart diagrams are composed of instances of StateVertex connected by instances of Transition. Recalling Figure 7, PseudoState and State are subclasses of StateVertex. Further, Branch, Fork, Join, InitialState and FinalState are categories of PseudoState. Thus, a specification of Transition, Branch, Fork, Join, InitialState and FinalState in terms of LOTOS operators provides the foundation required to map UML models describing behavioral aspects of software systems into LOTOS specifications.

Let A, B, C and D be Activities and a and b be States. Table 3 presents the mapping of Transition, Fork, Join, Branch, InitialState and FinalState into LOTOS behavioural expressions. There, a State that can only be reached once within its immediate superstate is defined as as *non-visited* State. Otherwise it is defined as a *visited* State. The mapping of a visited State into LOTOS expressions requires a recursive invocation of the generated LOTOS process, as described by the UCD 6 in Table 3.

The LOTOS specification in Figure 8 for the top-level activity diagram in Figure 3 is produced using the UCDs already presented. The activity diagram itself is mapped as the LIBRARY_ACT specification

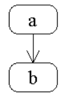
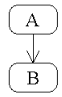
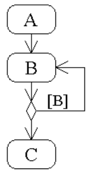
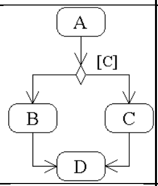
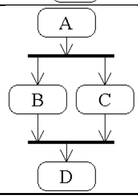
UCD	$\mathcal{U}$	$\Phi(\mathcal{U})$	a generic example	
			UML	LOTOS
3	Transition between two non-PseudoStates to a non-visited State in a statechart diagram	action prefix		a;b
4	Transition between two non-PseudoStates to a non-visited State in an activity diagram	enabling		$A >> B$
5	Transition to or from a PseudoState going to or coming from a non-visited State	considered as part of the PseudoState	—	—
6	Transition from a Branch to a visited State	recursive process		A >> REC_ACT **where** **process** REC_ACT[abort]: **exit** := B >> (C [] [B]REC_ACT) **endproc**
7	Branch	choice		$A >> (B[][C]C) >> D$
8	Fork and Join	interleave with parenthesis		$A >> (B\|\|\|C) >> D$
9	InitialState	not mapped	•	—
10	FinalState	not mapped	⦿	—

Table 3: UCDs related to Transition, Branch, Fork, Join, InitialState and FinalState constructors.

(UCD 2). Then, a navigation through the activity diagram should be performed. The navigation starts at the InitialState, which is not mapped into the behaviour expression of LIBRARY_ACT (UCD 9). Following the Transition leaving the InitialState, which is also not mapped in LIBRARY_ACT (UCD 5), the `connect` Activity is reached, which is mapped as the CONNECT_ACT process (UCD 1). The Transition leaning the `connect` Activity, mapped as >> (UCD 4), reaches the `SelectFunction` Activity, mapped as SELECTFUNCTION_ACT (UCD 1). Finally, following the Transition leaving the `SelectFunction` Activity, which is not mapped in LIBRARY_ACT (UCD 5), a FinalState is reached, which is also not mapped in LIBRARY_ACT (UCD 10).

To facilitate the identification of the roles that LOTOS processes are playing in the specification of UML constructors, process names are suffixed by "_ACT" if the processes are modelling Activities, by "_AS" if they are modelling ActionStates, and by "_CLS" if they are modelling Classifiers or their subclasses, i.e., Classes and Interfaces.

The mapping process should go into the activities. For instance, Figure 9 presents a refinement for the SELECTFUNCTION_ACT process introduced in Figure 8. The behaviour expression of SELECTFUNCTION_ACT process was created by traversing the `SelectFunction` Activity, and using the UCD's mapping in the same way as described for LIBRARY_ACT.

```
specification LIBRARY_ACT[abort] : exit
  behaviour
    CONNECT_ACT[abort] >> SELECTFUNCTION_ACT[abort]
    [> abort; exit
  where
    process CONNECT_ACT[abort] : exit :=
      (* CONNECT_ACT specification *)
    endproc
    process SELECTFUNCTION_ACT[abort] := exit :=
      (* SELECTFUNCTION_ACT specification *)
    endproc
endspec
```

Figure 8: The LOTOS specification of the `Library` Activity.

```
process SELECTFUNCTION_ACT[abort] : exit
    CREATEMAINUI_ACT[abort] >> DO_SELECTION_ACT[abort]
    [> abort; exit
  where
    process DO_SELECTION_ACT[abort] : exit :=
      SELECTSERVICE_AS[abort] >>
      ( ( [borrowbook]BORROWBOOK_ACT[abort] >> DO_SELECTION_ACT[abort] )
            []
        ( [renewloan]RENEWLOAN_ACT[abort] >> DO_SELECTION_ACT[abort] )
            []
        ( [returnbook]RETURNBOOK_ACT[abort] >> DO_SELECTION_ACT[abort] ) )
      [> abort; exit
    endproc
    (* CREATEMAINUI_ACT, SELECTSERVICE_AS, BORROWBOOK_ACT,
       RENEWLOAN_ACT and RETURNBOOK_ACT specifications *)
endproc
```

Figure 9: A refinement of the SELECTFUNCTION_ACT process introduced in Figure 8.

Type Mappings

Both UML and LOTOS provide a set of *primitive types* and allow the specification of *complex types* from these primitive types. Primitive types in UML are specified by the constructors in the UML DataType package. Primitive types in LOTOS are provided along with the specification of LOTOS. Complex types in UML are specified by the specification of Classes, where their Attributes can be primitive types, defined in terms of elements of DataType, or other Classes. In the same way, complex types in LOTOS can be specified by a type specification, as introduced in Section 3. Further, these type specifications can be composed of primitive types or other complex types.

There is almost a complete match between the primitive types of UML and LOTOS. For example, Boolean matches with **Bool**, and Integer matches with **nat**. Few primitive types of UML do not match with any primitive type of LOTOS. In this case, we can use a LOTOS specification type to define these non-matching types. For example, Enumerate can map to a LOTOS type definition as in Figure 10. There, a `element` can be retrieved from the enumeration using the `getnext` operation. In fact, `element` is a parameterised **sorts** specified by the **formalsorts** operator, and `getnext` is a parameterised **opns** specified by the **formalopns** operator.

Further, the `Enumerate` type can be used as the Enumerate to specify different enumerations. For instance, the ENUMERATE_BOOKCOPY type definition in Figure 11 can be specified from the BOOKCOPY type definition. The **actualizedby** and **using** operators define that BOOKCOPY should provide the parameters specified in Figure 10. Then, the **sortnames** and **for** operators specify that `BookCopy` is the value for the `element` sort parameter. The **opnames** and **for** operators specify that the `nextBookCopy` is the value for the `getnext` operation parameter.

Doing these type mappings as presented here, a complete translation of the types specified by Classes into LOTOS type specifications can be achieved.

```
type ENUMERATE is
   formalsorts element
   formalopns getnext: -> element
   sorts Enumerate
   opns hasnext: -> Bool
endtype
```

Figure 10: `Enumerate` type specification.

```
type ENUMERATE_BOOKCOPY is
   Enumerate actualizedby BOOKCOPY using
      sortnames BookCopy for element
      opnames nextBookCopy for getnext
endtype
```

Figure 11: Specification of `EnumerateBookCopy` from the `Enumerate` type.

6 A Semantics for Some Structural Aspects of UML

6.1 The Classifier Constructor

An informal definition of Classifier may be appropriate for readers not familiar with the UML metamodel terminology. Class is a common term when modelling and implementing object-oriented software systems. In terms of the UML metamodel, however, it is common to use Classifier rather than Class in certain circumstances. In fact, Classifier is the constructor that has StructuralFeatures such as Attributes, and BehavioralFeatures such as Operations. Class is a specialisation of Classifier that specifies that it can be instantiated into an Object. The main reason for the distinction between Class and Classifier is that there are other constructors that are Classifiers other than Class such as the Interface constructor.

Considering that Class is the major constructor for specifying structural aspects of software systems in UML, and that the Classifier is a generalisation of Class, a UML constructor definition for Classifier can provide a semantics for many structural aspects of UML. In this section, the features of the `BookCopy` Class in Figure 5 are gradually translated into LOTOS in order to introduce a generic LOTOS specification for Class, Classifier, and their related constructors.

6.2 A First Specification of Classifier

Classifier is frequently used as a type specification in UML since it plays the `type` role several times in the UML metamodel. In fact, a Classifier is an implicit definition of type in the UML context. In LOTOS, types are explicitly declared. The `BookCopy` class in Figure 5 has the attributes `status` and `copyCode`. Thus, a `BookCopy` type can be specified as in Figure 12. This means that `BookCopy` is the type of the `BookCopy` class.

```
type BOOKCOPY is enum, String
   sorts BookCopy
   opns  mk_BookCopy:                     -> BookCopy
         mk_BookCopy2: enum, String      -> BookCopy
         setstatus : BookCopy, enum       -> BookCopy
         getstatus: BookCopy              -> enum
         setcopycode: BookCopy, String   -> BookCopy
         getcopycode: BookCopy            -> String
   (* eqns specification *)
endtype
```

Figure 12: `BookCopy` type specification.

A Classifier can specify behaviour in addition to the type specification, as indicated by the BOOKCOPY_CLS process in Figure 13. There, the copyCode attribute of BookCopy is defined as a pair of low-level operations defined in the type definition, viz., setcopycode and getcopycode, that are used by a pair of observable actions, viz., cci and cco, to update and retrieve the current value of the attribute. Moreover, the BOOKCOPY_CLS process specifies that the attributes of the BookCopy's instance are ready to be updated, e.g., *cci?new_cc* : **String**, or retrieved, e.g., *cco!getcopycod(bc)*, while the process is active. The same strategy is used to specify the status attribute of BookCopy.

```
process BOOKCOPY_CLS[si, so, cci, cco, destroy_bookcopy](bc:BookCopy) :
  exit :=
  ( si?new_status:enum;
        BOOKCOPY_CLS[si, so, cci, cco, destroy_bookcopy]
                    (setstatus(bc,new_status))        []
    so!getstatus(bc);
        BOOKCOPY_CLS[si, so, cci, cco, destroy_bookcopy]
                    (bc)                              []
    cci?new_cc:String;
        BOOKCOPY_CLS[si, so, cci, cco, destroy_bookcopy]
                    (setcopycode(bc,new_cc))          []
    cco!getcopycode(bc);
        BOOKCOPY_CLS[si, so, cci, cco, destroy_bookcopy]
                    (bc)   )
  [> destroy_bookcopy; exit
endproc
```

Figure 13: Specification of Attributes in the BookCopy process.

A generic Classifier type and process can be defined from the BOOKCOPY type and BOOKCOPY_CLS process examples.

UCD 11 *A* Classifier *is defined by the specification of a LOTOS type definition and a LOTOS process definition. The type definition is specified as follows.*

```
type CLASS is
    sorts Class
    opns  mk_class  -> Class
endtype
```

The process definition is specified as follows.

```
process CLASS_CLS[destroy_class] (c: Class) : exit :=
    i; CLASS_CLS
       (* this internal action i represents
          a class with an underspecified behaviour *)
    [> destroy_class; exit
endproc
```

The destroy_class gate in UCD 11 corresponds to the abort gate of ACTIVITY_ACT in the context of Classifiers. One distinct role is that the destroy_class action associated to the destroy_class gate is the LOTOS constructor used to specifyDestroyActions as discussed in Section 7.4.

Hereafter, the CLASS and CLASS_CLS names denote the generic type and process definitions, respectively, of a Classifier. Complete specifications of CLASS and CLASS_CLS are eventually presented in Section 6.7. Considering CLASS and CLASS_CLS, it is possible to define the mappings for Object, Class and Attribute.

UCD 12 *An* Object *is a LOTOS process variable of the CLASS type and is used by the CLASS_CLS process.*

UCD 13 *A* Class *is a* Classifier *that can be used by a* CreateAction ActionState *(UCD 24) to specify* Objects *in LOTOS processes.*

UCD 14 *An* Attribute `Attr` *of type* `AttrType` *is specified by a pair of type operations in CLASS, e.g.,* `setAttr` *and* `getAttr`*, and a pair of observable actions, e.g.,* `attri` *and* `attro`*, in CLASS_CLS. The CLASS type with an* `Attr` *attribute is specified as follows:*

```
type CLASS is AttrType
    sorts Class
    opns  mk_Class:                        -> Class
          mk_Class2: AttrType              -> Class
          setAttr: Class, AttrType -> Class
          getAttr: Class                  -> AttrType
    (* eqns specs *)
endtype
```

The CLASS_CLS process with an `Attr` *attribute is specified as follows:*

```
process CLASS_CLS[attri, attro, destroy_class]
        (c: Class) : exit :=
  (attri?new_attr:AttrType;
     CLASS_CLS[attri, attro, destroy_class]
                        (setAttr(c,new_attr)) []
   attro!getAttr();
     CLASS_CLS[attri, attro, destroy_class](c)  )   []
  ... ) [> destroy_class; exit
endproc
```

The pair consisting of the BOOKCOPY type and the BOOKCOPY_CLS processes provide an incomplete specification of the BookCopy Class in Figure 5. For instance, the Operations of BookCopy are not specified in Figure 13.

6.3 Operation Specification

In addition to the attributes, the BookCopy class has seven Operations, as described in Figure 5. A LOTOS specification for the `renewLoan()` and `getCopyCode()` Operations is presented in this section. The `renewLoan()` Operation implements a functionality of the Library System, as described in Section 4. Rather than implementing Library System functionalities, the `getCopyCode()` Operation is responsible for encapsulating the `copyCode` Attribute.

An explanation of how the `renewLoan()` and `getCopyCode()` Operations can be specified in LOTOS indicates how Classifier's Operations can be specified in LOTOS. Each Operation is associated with a `CallAction`, which is composed of a pair of `Messages`. The Message and CallAction and Operation constructors are defined in terms of LOTOS operators as follows:

UCD 15 *A* Message *is the specification of an observable action of CLASS_CLS in the behaviour expression of CLASS_CLS or any of its subprocesses.*

UCD 16 *A* CallAction *is a pair of* Messages*, e.g., < callinvoker , callresponse >. In the behaviour expression of the* Classifier *process of an* Object *invoking the* Operation*, callinvokers must come before callresponses. In the* Classifier *process of an* Object *where the* Operation *is invoked, the* CallAction Messages *are used as specified in UCD 17.*

The **accept ... in** constructor of LOTOS is required to introduce the Operation's UCD. The **accept ... in** is used to assign the results of a process preceding an *enabling* (>>) constructor to a set of LOTOS variables defined after the **accept** keyword. Thus, the Operation constructor can be defined in terms of LOTOS operators as follows.

UCD 17 *An* Operation *is the specification of a subprocess in CLASS_CLS, e.g., OPERATION, defined as follows:*

```
process CLASS_CLS [callinvoker, callresponse, destroy_class]
          (c: Class) : exit :=
  ( (
    ...
    []
    ( callinvoker;
      OPERATION[callresponse](c) >>
      accept upd_c: Class in CLASS_CLS[callinvoker,
                     callresponse, destroy_class](upd_c) )
    []
    ... ) [> destroy_class; exit
  where
    process OPERATION[callresponse](c) :
             exit (Class) :=
      i ; callresponse; exit (any Class)
    endproc
endproc
```

According to UCD 16, a pair of *CallInvoker* and *CallResponse* observable actions is specified for each Operation. Further, the Operation process, called *operation process*, preserves the state of its Classifier, receiving and returning the state of their Classifier's process, as in UCD 17. For example, in Figure 14, the RENEWLOAN process receives the bc object of type BookCopy, returning it on the **exit**(**any** BookCopy). The **any** operator of LOTOS specifies that any value in the domain of the specified type, e.g., BookCopy, can be returned. Moreover, the value of c declared in UCD 11 may be unaffected. For instance, if the Operation in the UML model is specified with the isQuery Attribute set to TRUE then the **any** Class in **exit** must be replaced by the parameter c of CLASS_CLS. An Operation's parameters are added to the Operation process's parameter list. Return values are added as a LOTOS event to the CallResponse action. In the case of the GETCOPYCODE process, the string returned by the Operation, e.g., getccode(bc), is added to getcc_res, the GETCOPYCODE's CallResponse action.

```
process BOOKCOPY_CLS[getcc, getcc_res, renew, renew_res,
                destroy_bookcopy] (bc: BookCopy) : exit :=
  ( getcc; GETCOPYCODE[getcc_res](bc) >>
    accept upd_bc:BookCopy in
      BOOKCOPY_CLS[getcc, getcc_res, renew, renew_res,
                   destroy_bookcopy](upd_bc)
  []
    renew; RENEWLOAN[renew_res](bc) >>
    accept upd_bc:BookCopy in
      BOOKCOPY_CLS[getcc, getcc_res, renew, renew_res,
                   destroy_bookcopy](upd_bc) )
  [> destroy_bookcopy; exit
 where
  process GETCOPYCODE[getcc_res](bc:BookCopy) : exit(BookCopy) :=
    i; getcc_res!getccode(bc); exit(any BookCopy)
  endproc
  process RENEWLOAN[renew_res](bc:BookCopy) : exit(BookCopy) :=
    i; renew_res; exit(any BookCopy)
  endproc
endproc
```

Figure 14: Specification of Operations in the BookCopy process.

Finally, behavioural expressions of Operation processes are specified by an i unobservable action of LOTOS prefixing a CallResponse action, prefixing an **exit** operation. The i action specifies that some action should happen during the execution of the method, but nothing can be said about this action. Broadly speaking, LOTOS can be used for the specification of implemented software systems. UML, however, is intended to be used during the design phase of the development process of a software system. Therefore, some *under-specifications* are expected in LOTOS specifications generated from UML models [5]. Most under-specifications are implicitly represented in the LOTOS specification. For example, Activities that are not refined into ActionStates, such as the `connect` Activity in Figure 8, are under-specifications. In the case of Operations, however, the i actions in their behaviour expressions are explicitly under-specifications. These i actions are used in LOTOS, in this case, to represent an internal event that may not be influenced by any process (non-determinism). In fact, it is assumed that the specification of methods is an implementation concern rather than a design concern. Nevertheless, LOTOS could be used to specify the implementation of methods.

The BOOKCOPY_CLS version with the encapsulated attributes does not specify attributes. In fact, this is a simplification in order to keep the translation of operations concise. However, a proper specification of the BookCopy class should preserve the attribute specification that may be hidden from other processes using the **hide** operator of LOTOS.

6.4 Association and ClassifierRole Specifications

Let $\mathcal{C}$ be the finite set of Classifiers of a UML design of $\mathcal{S}$, and $\mathcal{A} = \mathcal{C} \times \mathcal{C}$. Thus, an Association can be defined as follows.

Definition 1 *An Association (α) is a binary relationship between* Classifiers *($\alpha \in \mathcal{A}$).*

An Association, as in Definition 1, is not a UCD. Indeed, in this paper, the Association's UCD is defined by the ClassifierRoles related to an Association. Thus, let $\mathcal{R} = \mathcal{A} \times \mathcal{C}$. From $\mathcal{R}$ and Definition 1 we can see that $\forall \alpha \in \mathcal{A} \Rightarrow \exists \rho_1, \rho_2 \in \mathcal{R} \bullet \rho_1 \neq \rho_2 \wedge \Pi_{\mathcal{A}}(\rho_1) = \Pi_{\mathcal{A}}(\rho_2) = \alpha$, where $\Pi_{\mathcal{A}}(\rho_z)$ is a projection of the value in the domain of $\mathcal{A}$ in ρ_z. Then a ClassifierRole can be defined as follows.

UCD 18 *A* ClassifierRole *(ρ) is a binary relationship between a* Classifier *and an* Association *($\rho \in \mathcal{R}$) specified as a* CallAction*, where its CallResponse action returns an* Enumeration *of related instances of the associated* Classifier[1] *for the current instance. The* Enumeration *is generated by a GEN_ENUM process which precedes the CallResponse. For an associated Class2* Classifier*, the GEN_ENUM is specified as follows.*

process GEN_ENUM[ ] : **exit** (Enumeration_Class2) :=
  exit (**any** Enumeration_Class2)
endproc

Association's UCD can be specified from UCD 18. In fact, ClassifierRoles are synchronisations between processes that are equivalent to Associations mathematically defined as in Definition 1 [2]. Further, the UML specification says that there are two instances of AssociationEnd for each instance of Association. Moreover, the AssociationEnd has an attribute `aggregation` which can have the values `none`, `aggregate` or `composite`. Therefore, Table 4 introduces a set of UCDs based on UCD 18 and the possible combination of types of AssociationEnds in an Association. There, α_1 is an Association between a $P1$ Classifier playing a ρ_1 ClassifierRole and a $P2$ Classifier playing a ρ_2 ClassifierRole. Furthermore, the execution of the `CREATE_CLASS_AS[]` instantiates Class, as described in Section 7.1.

The `BookCopy` Class is associated to the `Loan` and `Book` Classes, as presented in Figure 5. Thus, the example in Figure 15 presents the LOTOS specification of the `onLoan` ClassifierRole in BOOKCOPY_CLS. There, the `Eloan` returned by the `onloan_res` is an enumeration of type `Enumeration_Loan`. The invocation of the `hasBookCopy` ClassifierRole may be specified in the specification of the Methods of the `BookCopy` Operations later in the implementation phase of the Library system.

6.5 Signal, Sender and Receiver Specifications

The interaction between objects, as presented so far, is represented by Operations performed in a synchronous manner. For instance, methods performing CallActions need to wait for the conclusion of the

[1] As described in Section 5.3, there is an enumeration type definition for every Classifier type definition translated from a UML model.

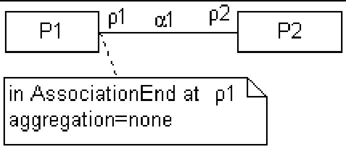
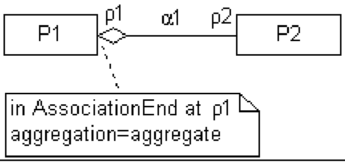
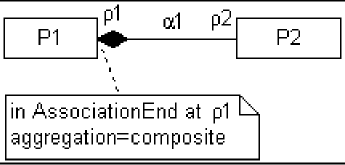
UCD	U	$\Phi(U)$
19		$(\exists\Phi(\rho_1) \vee \Phi(\rho_2))$ where $\Phi(\rho_1)$ and $\Phi(\rho_2)$ are defined as in UCD 18.
20		$(\exists\Phi(\rho_1) \vee \Phi(\rho_2))$ $\wedge$ (`P1_CLS` has parameter of `P2` type)
21		$(\exists\Phi(\rho_1) \vee \Phi(\rho_2))$ $\wedge$ (`P1_CLS` has parameter of `P2` type) $\wedge$ (execution of `CREATE_P1_AS[]` $\Rightarrow$ execution of `CREATE_P2_AS[]`) $\wedge$ (execution of `destroy_p1` $\Rightarrow$ execution of `destroy_p2`)

Table 4: Association mapping.

```
process BOOKCOPY_CLS[ setcc , ..., onloan , onloan_res , hasbookcopy ,
                hasbookcopy_res , destroy_bookcopy ] ( bc: BookCopy ) : exit :=
   ( (* previously defined attributes and operations *)
     []
     onloan ; ONLOAN[ onloan_res ]( bc) >>
     accept upd_bc : BookCopy in
       BOOKCOPY_CLS[ setcc , ..., onloan , onloan_res , hasbookcopy ,
                     hasbookcopy_res , destroy_bookcopy ]( upd_bc )
   [> destroy_bookcopy ; exit
 where
   (* previously defined operation processes *)
   process ONLOAN[ onloan_res ]( bc : BookCopy ) : exit ( BookCopy ) :=
     i ; GEN_ENUM >> accept Eloan : Enumeration_Loan in
        onloan_res ! Eloan ; exit ( any BookCopy)
     where
       process GEN_ENUM[ ] : exit ( Enumeration_Loan ) :=
         exit ( any Enumeration_Loan )
       endproc
   endproc
endproc
```

Figure 15: Specification of Associations in the BookCopy process.

triggered Operation. However, there may be actions that must to be performed in an asynchronous way. In the running case study, for instance, a Signal can be raised every time a `BookCopy` object is returned. In fact, `returnBook()` is the Sender Operation related to the `ReturnedCopy` Signal in Figure 5. Moreover, the invocation of the `returnBook()` Operation that raises the `ReturnedCopy` Signal is a SendAction for the Signal. Nevertheless, the Signal constructor is responsible for providing such a facility modelled in Figure 5.

UCD 22 *A* Signal *is a* Message *specified immediately after the* **in** *keyword in the invocation of the* Signal*'s associated* Operation *process (UCD 17).*

UCD 23 *A* SendAction *is the specification of a* Signal *in the behavioural expression of an* ActionState *(UCD 28).*

Thus, UCD 22 can be used to implement the required asynchronous action presented above. For instance, `returnedcopy_sig` in Figure 16 is raised every time the RETURNBOOK operation of BOOKCOPY_CLS is performed.

```
process BOOKCOPY_CLS[..., returnbook, returnbook_res,
          returnedcopy_sig, destroy_bookcopy] (bc: BookCopy) : exit :=
   ( (* prev. defined attributes, operations and associations *)
     []
     ( returnbook?new_bc: Bookcopy; RETURNBOOK[returnbook_res](bc, new_bc) >>
         accept upd_bc:BookCopy in returnedcopy_sig!upd_bc;
         BOOKCOPY_CLS[..., returnbook, returnbook_res,
                     returnedcopy_sig, destroy_bookcopy](upd_bc) ) )
   [> destroy_bookcopy; exit
 where
   (* other operation and association processes *)
endproc
```

Figure 16: Specification of Signals in the BookCopy process.

The use of Signals has more impact on the structure of Classes acting as Receivers than in Operations acting as Senders. Instances of the `Reservation` Class must receive the `returnedcopy_sig` Signal raised by BOOKCOPY_CLS in an asynchronous way. In Figure 17, the *interleave* operator (|||) specifies that the `returnbook_sig` is not synchronised with any other action that may be performed within RESERVATION_CLS.

```
process RESERVATION_CLS[..., returnedcopy_sig, destroy_reservation]
            (res:Reservation) : exit :=
   (
     (* behaviour expression for attributes,
             operations and associations *)
   )
   |||
   ( returnedcopy_sig?bc:BookCopy;
     notify!bc; notify_res;
     RESERVATION_CLS[..., returnedcopy_sig, destroy_reservation](res))
   [> destroy_reservation; exit
 where
     (* definitions of operation and association processes *)
endproc
```

Figure 17: Specification of the `Reservation` process as a Receiver of the `ReturnBook` Signal.

The RESERVATION_CLS is acting as a handler of the `returnedcopy_sig` since the Signal is invoking a `Notify` Operation also defined in the RESERVATION_CLS. However, it may be the case that RESERVATION_CLS could rely on other Classifiers that could act as Handlers as well.

6.6 Specification of Generalisation

As presented so far, a Classifier is defined by its type and process. The `BookCopy` classifier is defined by the BOOKCOPY type and the BOOKCOPY_CLS process. Thus, a Classifier can be generalised as long it can inherit the type and process actions provided by its superclass.

In terms of type there is no difficulty in implementing this. For instance, supposing that the `Person` classifier is already specified in LOTOS, the type of the `Borrower` classifier can be specified in the way presented in Figure 18. For instance, the `getname(castPerson(aBorrower))` type operation can return the `name` Attribute defined in the `Person` type specification.

Moreover, Classifier features, viz. Attributes, Operations, Associations and Signals are specified as observable actions. Therefore, a full synchronisation between PERSON_CLS and BORROWER_CLS (e.g., $PERSON\_CLS \parallel BORROWER\_CLS$) makes `Borrower` inherit the features of `Person`. In this case, `Borrower` must only to implement the role actions of its association with the `Loan` class, which is what makes it different from `Person`.

```
type BORROWER is PERSON with
   sorts Borrower
   opns mk_borrower :              -> Borrower
        isPerson : Person          -> Borrower
        castPerson : Borrower      -> Person
endtype
```

Figure 18: Specification of `Borrower` as a specialisation of `Person`.

6.7 A Generic Type and Process Definition for Classifiers

The presented UCDs have demonstrated how to generate a LOTOS specification for a Classifier and its Attributes, Operations, Associations and Signals. Thus, assume that a Classifier can be represented by a tuple $\mathcal{T} = < \Gamma, \Omega, \Sigma_s, \Sigma_r >$, where:

- Γ is a finite set of Attributes;
- Ω is a finite set of Operations;
- Σ_s is a finite set of Signals where the Classifier is a Sender;
- Σ_r is a finite set of Signals where the Classifier is a Receiver

For each tuple $\mathcal{T}$ there exists one LOTOS specification of a CLASS type and a CLASS_CLS process. For instance, suppose that for a specific Classifier, the cardinality of Γ is w ($\#\Gamma = w$), which means, $\Gamma = \{ a_1, a_2, ..., a_w \}$. Further, suppose that each attribute of Γ has a corresponding type a_1type, a_2type, ..., a_wtype. Figure 19 presents the generic CLASS type.

```
type CLASS is  a1type , ... , awtype
   sorts Class
   opns  mk_Class :                        -> Class
         mk_Class2 :  a1type , ... , awtype -> Class
         set a1 : Class, a1type            -> Class
         get a1 : Class                    -> a1type
         set a2 : Class, a2type            -> Class
         get a2 : Class                    -> a2type
         ...
         set aw : Class, awtype            -> Class
         get aw : Class                    -> awtype
   (* eqns specifications *)
endtype
```

Figure 19: A type for a generic Classifier.

Moreover, suppose that $\#\Omega = x$ ($\Omega = \{ o_1, o_2, ..., o_x \}$), $\#\Sigma_s = y$ ($\Sigma_s = \{ s_1, s_2, ..., s_y \}$); and $\#\Sigma_r = z$ ($\Sigma_r = \{ t_1, t_2, ..., t_z \}$). Then, a set of parameters and their types is defined which is provided by each Signal of Σ_r, $\{t_ip_1$: t_ip_1type, t_ip_2: t_ip_2type, ... $\}$. So, for each Operation, e.g., o_i, is defined:

- a *CallInvoke* action, o_i, and a *CallResponse* action, o_ires;
- a set of parameters along with their respective types, $\{o_ip_1$: o_ip_1type, o_ip_2: o_ip_2type, ... $\}$; and
- a set of results along with their respective types, $\{o_ir_1$: o_ir_1type, o_ir_2: o_ir_2type, ... $\}$.

Finally, to simplify the representation of CLASS_CLS, let G be the ordered set of gates: $\{a_1inp$, a_1out, a_2inp, a_2out, ..., a_winp, a_wout, o_1, o_1res, o_2, o_2res, ..., o_x, o_xres, s_1, s_2, ..., s_q, ..., s_y, t_1, t_2, ..., t_z, destroy_class$\}$. Thus, Figure 20 presents the generic CLASS_CLS process.

```
process CLASS_CLS [G](c: Class) : exit :=
 (
   (  a1inp?p1:a1type;   CLASS_CLS[G](seta1(c,p1))
   [] a1out!a1;           CLASS_CLS[G](c)
      ...
   [] awinp?pw:awtype;   CLASS_CLS[G](setaw(c,pw))
   [] awout!aw;           CLASS_CLS[G](c)
   [] o1?o1p1:o1p1type?o1p2:o1p2type...;  OPER1[o1res](c, o1p1, o1p2, ...) >>
          accept upd_c:Class in CLASS_CLS[G](upd_c)
      ...
   [] ox?oxp1:oxp1type?oxp2:oxp2type...;  OPERx[oxres](c, oxp1, oxp2, ...) >>
          accept upd_c:Class in CLASS_CLS[G](upd_c)
   ||| (t1?t1p1:t1p1type?t1p2:t1p2type?...; ...;  CLASS_CLS[G](...))
   ...
   ||| (tz?tzp1:tzp1type?tzp2:tzp2type?...; ...;  CLASS_CLS[G](...))
 ) [> destroy_class; exit
 where
   process OPER1[o1res]
         (c: Class, o1p1: o1p1type, o1p2: o1p2type, ...): exit(Class) :=
      i; o1res!o1r1!o1r2!...; exit(any Class)
   endproc
   ...
   process OPERx[oxres]
         (c: Class, oxp1: oxp1type, oxp2: oxp2type, ...): exit(Class) :=
      i; oxres!oxr1!oxr2!...; exit(any Class)
   endproc
endproc
```

Figure 20: A process for a generic Classifier.

7 A Semantics for Some Behavioral Aspects of UML

The presented specification of the Classifier constructor can describe important aspects of any software system. Attributes can describe the state of systems. Operations and Signals can describe the communication mechanisms of objects. However, there are two major aspects of software systems that still need to be specified. The first aspect is how actions, e.g., CallActions and SendActions, can be performed to provide a system's functionalities. The second aspect is how objects are interconnected to share an Operation's messages and Signals, thereby supporting actions. The specification of a top-level Activity (UCD 2) composed of ActionStates and subactivities, that are also Activities (UCD 1), explains how actions can be performed. Thus, a UCD for each category of ActionState in the UML specification may be needed to generate LOTOS specifications for Activities. The specification of ObjectFlowStates associated with ActionStates explains how objects, in this case, ClassifierInStates, can be interconnected. Thus, UCDs for ObjectFlowState and ClassifierInState may also be needed to generate such specifications for Activities. In fact, using these required UCDs it is possible, for instance, to produce the LOTOS specification in Figure 21 for the RenewLoan Activity in Figure 4. Therefore, the missing UCDs needed to generate the RENEWLOAN_ACT in Figure 21 are introduced in this section.

7.1 CreateAction ActionState Specification

CreateActions are performed to instantiate Classes. The specification of when actions creating objects can take place must be specified to identify when objects are available. For instance, in Figure 4, `createRenewUI` is an ActionState that creates `ui`, which is an instance of the `RenewUI` Class. In Figure 21, CREATE_RENEWUI_AS is the process responsible for the instantiation of `ui`. Indeed, `ui` is created by the `mk_renewui` type operation (UCD 11).

UCD 24 *A* CreationAction ActionState *for a Class* Classifier *is specified as follows:*

```
process CREATE_CLASS_AS[abort] : exit(Class) :=
    ( i; exit(mk_class) )
    [> abort; exit
endproc
```

```
process RENEWLOAN_ACT [ abort ] : exit :=
    hide setcc , setcc_res , getcc , getcc_res , renew , renew_res ,
        returnbook , returnbook_res , destroy_bookcopy ,
        getbookcopy , getbookcopy_res , destroy_renewui in
    ( ( CREATE_RENEWUI_AS[ abort ] >>
        accept ui : RenewUI in
        (   RENEWUI_CLS[ getbookcopy , getbookcopy_res , destroy_renewui ]( ui )
             |[ getbookcopy , getbookcopy_res ]|
           ( GET_BOOK_COPY_AS[ getbookcopy , getbookcopy_res , abort ] >>
             accept bc : BookCopy in
             (   BOOKCOPY_CLS[ setcc , setcc_res , getcc , getcc_res ,
                              renew , renew_res , returnbook , returnbook_res ,
                              destroy_bookcopy ]( bc )
                  |[ renew , renew_res ]|
               ( RENEW_LOAN_AS[ renew , renew_res , abort ]>>
                 destroy_bookcopy ; destroy_renewui ; exit
                ) ) ) ) )
      [> abort ; exit )
  where
    process CREATE_RENEWUI_AS[ abort ] : exit (RenewUI) :=
       ( i ; exit (mk_RenewUI) )
       [> abort ; exit
    endproc
    process GET_BOOK_COPY_AS[ getbookcopy , getbookcopy_res , abort ] :
            exit (BookCopy) :=
       ( getbookcopy ; i ; getbookcopy_res ?bc : BookCopy ; exit ( bc ) )
       [> abort ; exit
    endproc
    process RENEW_LOAN_AS[ renew , renew_res , abort ] : exit :=
       ( renew ; i ; renew_res ; exit )
       [> abort ; exit
    endproc
endproc
```

Figure 21: Specification of the ActionStates of the RenewLoan Activity.

The CreateAction ActionState *is invoked in its* Activity*'s behaviour expression (UCD 1) by the <CREATE_ACTIONSTATE> non-terminal specified as follows.*

```
<CREATE_ACTIONSTATE> ::=
    ''CREATE_CLASS_AS[ abort ] >> accept c : Class in (''
    <activity_behaviour > '')''
```

The definition of the types and processes of the modelled Classifiers are specified in the top-level Activity. Thus, the description of the structural part of the software system can be used throughout the specification of the behavioural aspects of the software system. For instance, CreateActions can be specified within any Activity.

7.2 ObjectFlowState and ClassifierInState Specifications

The term *object flow* is not clearly presented in the UML specification [27]. Even though, UML provides the ObjectFlowState and ClassifierInState constructors to specify object flows. Figure 22 indicates the UML constructors for building object flows. In Figure 21, the `ui` ClassifierInState from Figure 4 is produced as a result of the `CreateRenewUI` ActionState. ObjectFlowStates specify the incoming and outgoing of objects with respect to the scope of an ActionState. Furthermore, ObjectFlowStates provide the Object where the associated ActionState takes place, and optional Objects which can be used as parameters to the Action performed in an ActionState.

UCD 25 *A* ClassifierInState *is an* Object*, as specified by UCD 12. This UCD is introduced since* ClassifierInStates *and* Objects *are distinct constructors in the UML metamodel.*

UCD 26 *An incoming* ObjectFlowState *is a synchronisation of a* Signal *or the* Messages *of an* Operation *between an* ActionState *process and a* Classifier *process.*

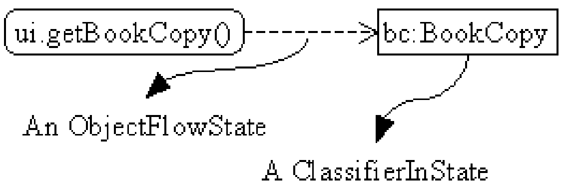


Figure 22: An *object flow* and its ObjectFlowState and ClassifierInState.

UCD 27 *an outgoing* ObjectFlowState *is the passing of a* ClassifierInState *to an* ActionState*.*

7.3 CallAction and SendAction ActionState Specifications

The `ui.getBookCopy()` is a CallAction ActionState where the GET_BOOK_COPY operation of a `RenewUI` Object instantiated in `createRenewUI` is invoked. This GET_BOOK_COPY operation is defined in the RENEWUI_CLS. No parameter is passed to this operation, i.e., the `getBookCopy` action is not followed by any variable. The result of the operation is the `bc` Object of the `BookCopy` Class. In fact, this `bc` should be an Object either previously instantiated in the current session or produced as the result of a query submitted to an associated database system. Once again, the exactly specification of how the Method of an implementation could be implemented is under-specified during the design.

A SendAction ActionState has a LOTOS specification quite similar to a CallAction ActionState. The main differences between these two categories of ActionStates are those differences between SendActions and CallActions. For instance, if the associated Action is a SendAction, the specification of the ActionState does not includes the *CallInvoker* Message. With respect to their ActionStates, ObjectFlowStates and `ClassifierInStates` are defined in the same way.

UCD 28 *An* ActionState *performing a* CallAction *or a* SendAction *that has incoming object flows OF_IN_1 ... OF_IN_M and outgoing object flows OF_OUT_1 ... OF_OUT_N of types $OF\_OUT_1\_TYPE$ ... $OF\_OUT_N\_TYPE$ is defined by a LOTOS process definition specified as follows:*

```
process ACTIONSTATE_AS [action, action_res] :
                exit (type_result) :=
   action; action_res? result : type_result;
   exit (result)
endproc
```

The ActionState *is invoked in its* Activity*'s behaviour expression (UCD 1) by the <CALL_SEND_ACTIONSTATE> non-terminal specified as follows.*

```
<CALL_SEND_ACTIONSTATE> ::=
     <object_flow_in> <synch>
     `` ACTIONSTATE_AS[action, action_res] accept ''
     <object_flow_out_list> `` in (''
     <activity_behaviour> ``)''
<object_flow_in> ::=
     <object_flow_in_name>
     ``['' <object_flow_gates> ``]''
<synch> ::= `` |[action, action_res]| ''
<object_flow_in_name> ::= `` OF_IN_1 '' - `` OF_IN_M ''
<object_flow_out_list> ::= <> |
     <object_flow_out_list> <object_flow_out>
<object_flow_out> ::= <object_flow_out_name> ``:''
     <object_flow_out_type>
<object_flow_out_name> ::=
```

```
    ‘‘ OF_OUT_1 ’’ − ‘‘ OF_OUT_N ’’
<object_flow_out_type > ::=
    ‘‘ OF_OUT_1_TYPE ’’ − ‘‘ OF_OUT_N_TYPE ’’
```

7.4 DestroyAction ActionState Specification

ActionStates where DestroyActions are performed could be specified in a similar way to SendAction ActionStates. However, DestroyActions are not specified in activity diagrams of UML. This means that no explicit notation exists in activity diagrams to specify the destruction of an Object. We assume this lack of specification is due to two aspects of the UML specification:

- lack of a formal framework;
- lack of a feasible description of how users in interactive systems can communicate cancelling actions to Objects and Activities.

We suggest the LOTOS semantics for UML as a feasible formal framework for discussing such DestroyAction semantics. However, a minimal specification is required to avoid the creation of deadlocks when compiling the generated LOTOS specification. Therefore, the problem here is the identification of a semantics for the DestroyAction that does not requires any special notation. The presented approach is to force a DestroyAction for any Object created in an Activity when leaving the activity. This is implemented in the LOTOS mapping as an invocation of the `destroy_class` actions of every class used within the Activity that is not used in other Activity. In fact, `destroy_class` actions are the expected LOTOS constructors to be used in a DestroyAction ActionState's UCD.

8 LOTOS-Based UML Model Checking

There are many tools that can perform verification of LOTOS specifications making use of the formal properties of LOTOS. Therefore, suppose that the Φ function introduced in this paper can be defined for the other UML constructors not considered in this paper. This means that using Φ it may be possible to verify any UML model. Furthermore, problems that may be identified in LOTOS specifications generated by the Φ function may be interpreted as a *UML semantic problem.*

8.1 Verification of the Library System specification

The LOTOS specifications of the UML models of the Library System presented in this paper were checked using a LOTOS verification tool. Thus, a LOTOS specification for the class diagram in Figure 5 and the activity diagrams in Figures 3 and 4 was implemented applying the UCDs presented in this paper. Indeed, most of the LOTOS examples in this papers are fragments of this LOTOS specification, which has 351 lines of code.

CADP [9] was selected to verify the LOTOS specification. CADP is a set of integrated tools used to verify LOTOS specifications. EUCALYPTUS is the graphical environment of CADP responsible for invoking the CADP tools that analyse the LOTOS specification. Figure 23 presents a snapshot of EUCALYPTUS when verifying the LOTOS specifications of the Library System. The backgroud frame in Figure 23 is the main user interface of EUCALYPTUS.

The CAESAR and CAESAR.ADT tools in CADP are responsible for the compilation and verification of the LOTOS specification. For instance, using CAESAR it was possible to verify if the LOTOS specifications were syntacticly correct. Further, it was possible to generate a *Binary Coded Graph* (BCG), which is a computer representation for a *Labelled Transition System* [26] from the LOTOS specifications. In a BCG, a state is of the entire application rather than of part of the application such as an object. In the case of the Library System specifications, we can see in the `BCG monitor` frame in Figure 23 that its BCG is composed of 243 states and 940 transitions. Moreover, from the BCG generated, it was possible to verify if the LOTOS specifications did not have deadlocks and livelocks. and unreachable states. Still

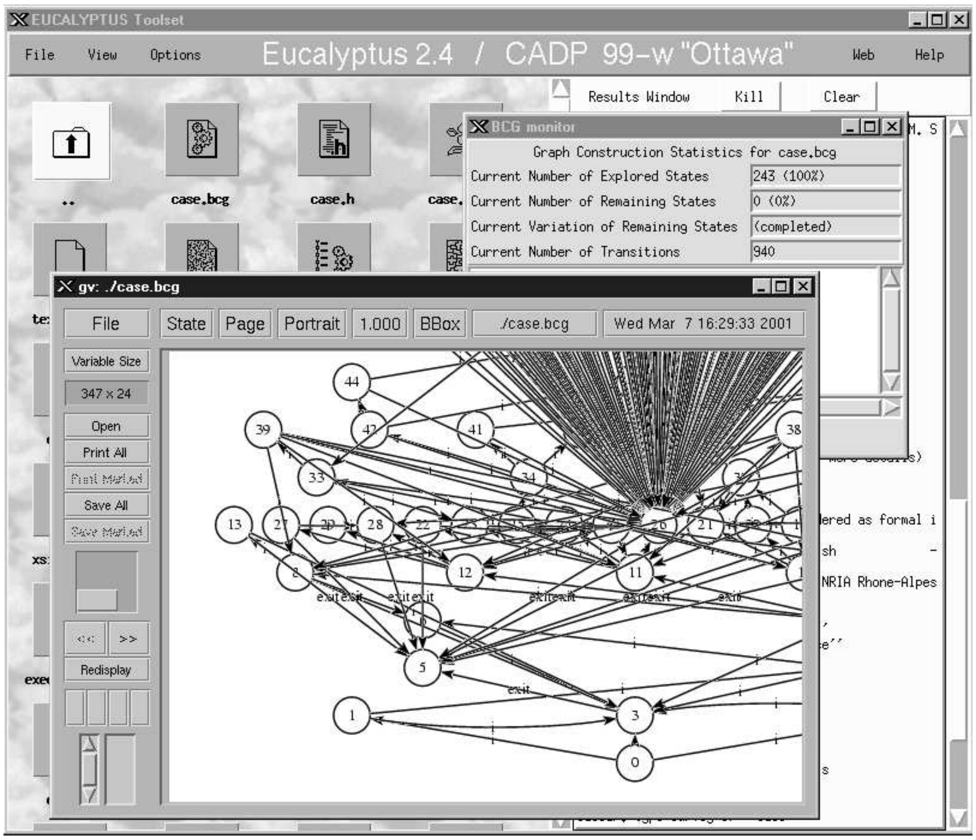


Figure 23: Snapshot of CADP when analysing the Library System specification.

in Figure 23, we can have a partial view of the generated BCG. The arcs in this partial view are the transitions and they are labelled with their triggering actions. As expected, most actions in the BCG are internal actions, e.g., *i*, since Operations are under-specified in the LOTOS specifications generated from UML models.

8.2 Complex UML Model Checking

The identification of potential problems such as deadlocks and livelocks were expected to be achieved at the beginning of this work. However, it was not expected that many of these deadlocks could be interpreted as modelling problems related to the proper use of object flows. Furthermore, it was not expected that many other of these deadlocks could be interpreted as modelling problems related to improper specifications of the complex relationship between activity diagrams and class diagrams. In fact, some identified deadlocks were related to the control-flow properties of the activity diagrams. For instance, it was possible to create LOTOS specifications with deadlocks from activity diagrams that do not specify object flows. On the other hand, there were deadlocks that were related to the data-flow properties of the activity diagrams. For instance, there were deadlocks caused by the improper use of object flows. Nevertheless, the UML models used to generate the LOTOS specifications that have both categories of deadlocks, however, conform with the UML metamodel. This means that these are valid UML models since they can be modelled in any UML tool. However, these UML models are incorrect in the sense they can specify systems that can have defects such as deadlocks, livelocks and unreachable states. Considering this, we are referring to these defects as *semantic problems* of UML.

Two modelling scenarios derived from the UML models of the Library System are used to exemplify the potential benefit to use the LOTOS-based semantics to verify UML models.

Scenario 1 presents a specification where is not guaranteed that the `bc.BookCopy` will be instantiated before the `bc.renewLoan()` ActionState is reached. This kind of problem may be easy to identify in a simple diagram such as that in Figure 24. However, this is a kind of problem that tends to be difficult to identify in medium and large-scale models. The identification of this category of problem using CADP is not also a trivial task. In fact, the problem was identified as a deadlock that could be fixed in the generated LOTOS specifications. However, amendment performed in the LOTOS specification might not be able to be mapped back into the original UML model. Nevertheless, CADP could identify the point in the LOTOS specification were the problem were, and from the LOTOS specifications it is not difficult to identify its equivalent constructors in the UML model due to the name conventions.

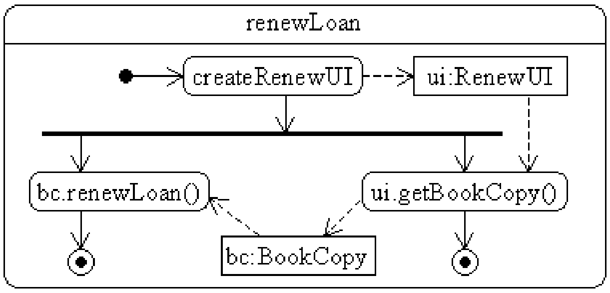


Figure 24: Modelling scenario 1.

The potential problem identified in Scenario 1 is related to the proper use of object flows in activity diagrams. Scenario 2 is an example of a potential problem relating the activity diagram in Figure 25 with the class diagram in Figure 5. The `ln` Object in Figure 25 is of the `Loan` Class that is aggregated to the `BookCopy` Class. Therefore, the associated instance of `BookCopy` should be instantiated at the time the `ui.getLoan()` ActionState is reached, which does not happen. Once again the presented problem could be identified by CADP as a deadlock. However, the analysis of such a problem in terms of the original UML models is a complex problem. The main difference between the deadlock in scenarios 1 and 2 is that in scenario 2 the deadlock appeared in the specification of a process representing a Class rather than in a process representing an Activity.

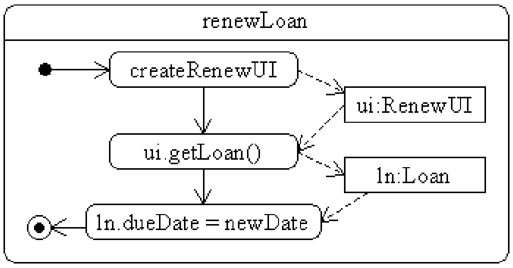


Figure 25: Modelling scenario 2.

An ad-hoc analysis of LOTOS specifications using CADP was able to identify semantic problems in scenarios 1 and 2. A systematic identification of the potential problems that can be verified by the proposed LOTOS-based semantics in other scenarios, however, is out of the scope of this paper. Moreover, a systematic interpretation in terms of UML diagrams of problems identified in LOTOS specification is also out of the scope of this paper.

9 Conclusions

This paper has demonstrated that it is possible to define a function that translates UML metamodel elements into LOTOS operators. Further, it has demonstrated that it is possible to verify indirectly the correctness of UML models verifying the produced LOTOS specification using available LOTOS tools. Moreover, the proposal presented is more comprehensive than other available semantics for UML (e.g., [5, 7, 8, 21, 24]). Indeed, it covers the major constructors of class diagrams and activity diagrams. The proposal also provides a formal specification for object flows, although this is incomplete even in terms of the informal specification in the UML specification. Considering that deployment diagrams are supported by class diagrams, and that statecharts are activity diagrams where transitions are restrict to those within a single object we observe that our approach can potentially be extended to cover the entire specification of a UML semantics.

Another benefit of the proposal presented is that it makes clear some well-known shortcomings of UML. For instance, it makes clear the the informal specification of CreateAction does not provide enough details to describe how it should be specified formally. Even worse is the informal specification of DestroyAction, which may have an impact on the whole application specification rather than being local to a single object. Thus, the presented proposal can be used for people to describe possible semantics for these two important actions.

A third benefit of the proposal presented is that it can be used as a framework to integrate other proposals of a semantics for UML. Indeed, such a framework could be used to incorporate the contributions of other semantic approaches covering parts of the UML specification not contemplated in previous work. There are many reasons for considering our proposal a feasible one for such an integration. First, LOTOS is a powerful specification language that already handles the specification of distributed systems. Second, the proposal uses the code-metamodel approach [8] at the same time as it uses an adapted mathematical theory of Breu et al. [5], as described in Section 5. Third, a LOTOS specification for statecharts can be incorporated into our approach re-using some definitions of Wang et al. [39]. Doing this statechart integration, the semantics considerations of UML statecharts presented in Latella et al. [21] can be integrated in our approach.

In terms of future work, there are two important directions that can be followed. The first direction is towards a complete specification of the UML semantics using the simplest approach available. In this sense, we believe that the presented proposal should be considered since it is based on a single specification language that provides verification facilities. The second direction is towards an automatic description of the verification results obtained from LOTOS tools in terms of UML models. The importance of the second direction is to make the verification facilities of LOTOS accessible for a wide community, in this case the UML community.

Acknowledgements. The author would like to thank Norman W. Paton and Enrico Franconi for their valuable comments in this work.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] A. Artale, E. Franconi, N. Guarino, and L. Pazzi. Part-Whole Relations in Object-Centered Systems: An Overview. *Data & Knowledge Engineering*, 20(3):347–383, 1996.

[3] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Network ISDN Systems*, 14(1), 1987.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, Reading, MA, 1999.

[5] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, and B. Rumpe. Towards a Formalization of the Unified Modelling Language. In *Proceedings of ECOOP'97*, LNCS, pages 344–366, Jyväskylä, Finland, June 1997. Springer-verlag.

[6] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*, volume 6 of *EATCS Monograph Series*. Springer-Verlag, 1985.

[7] A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a Formal Modeling Notation. In *Proceedings of the UML'98*, volume 1618 of *LNCS*, pages 336–348, Mulhouse, France, June 1998. Springer-Verlag.

[8] A. Evans and S. Kent. Core Meta-Modelling Semantics of UML: The pUML Approach. In *Proceedings of the UML'99*, volume 1723 of *LNCS*, pages 140–155, Fort Collins, CO, October 1999. Springer-Verlag.

[9] J. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP - A Protocol Validation and Verification Toolbox. In *Proceedings of CAV'96*, volume 1102 of *LNCS*, pages 437–440, New Brunswick, NJ, July 1996. Springer.

[10] A. Hall. Taking Z Seriously. In *ZUM '97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 89–91, Reading, UK, April 1997. Springer.

[11] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[12] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. Software Engineering and Methodology*, 5(4):293–333, October 1996.

[13] D. Harel, A. Pnueli, J. Schmidt, and R. Sherman. On the Formal Semantics of Statecharts (Extended Abstract). In *Proceedings of the LICS'87*, pages 54–64, Ithaca, NY, June 1987. IEEE Computer Society.

[14] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[15] G. Holzmann. The model checker SPIN. *IEEE Trans. Software Engineering*, 23(5):279–295, 1997.

[16] ISO/IEC-JTC1/SC21/WG1/FDT/C. *LOTOS, a Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, February 1989. IS 8807.

[17] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley, Reading, MA, 1992.

[18] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Seccess*. ACM Press, New York, NY, 1997.

[19] C. Kobryn. UML 2001: A Standardization Odyssey. *Communications of the ACM*, 42(10):29–37, October 1999.

[20] S. Kovacevic. UML and User Interface Modeling. In *Proceedings of UML'98*, pages 235–244, Mulhouse, France, June 1998. ESSAIM.

[21] D. Latella, I. Majzik, and M. Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[22] J. Lilius and I. Paltor. Formalising UML State Machines for Model Checking. In *Proceedings of UML'99*, volume 1723 of *LNCS*, pages 430–445, Fort Collins, CO, October 1999. Springer.

[23] P. Markopoulos. *A Compositional Model for the Formal Specification of User Interface Software*. PhD thesis, Queen Mary and Westfield College, University of London, March 1997.

[24] S. Mellor, S. Tockey, R. Arthaud, and Ph. Leblanc. An Action Language for UML: Proposal for a Precise Execution Semantics. In *Proceedings of the UML'98*, volume 1618 of *LNCS*, pages 307–318, Mulhouse, France, June 1998. Springer-Verlag.

[25] R. Milner. *Communication and Concurrency*. Prentice Hall, New York, NY, 1989.

[26] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, II. *Information and Computation*, 100(1):41–77, 1992.

[27] Object Management Group. *OMG Unified Modeling Language Specification*, June 1999. Version 1.3.

[28] F. Paternò. A Theory of User-Interaction Objects. *Journal of Visual Languages and Computing*, 5(3):227–249, 1994.

[29] D. Petriu and Y. Sun. Consistent Behaviour Representation in Activity and Sequence Diagrams. In *Proceedings of UML2000*, volume 1939 of *LNCS*, pages 369–382, York, UK, October 2000. Springer.

[30] P. Pinheiro da Silva and N. W. Paton. UMLi: The Unified Modeling Language for Interactive Applications. In *Proceedings of UML2000*, volume 1939 of *LNCS*, pages 117–132, York, UK, October 2000. Springer.

[31] P. Pinheiro da Silva and N. W. Paton. User Interface Modelling with UML. In *Information Modelling and Knowledge Bases XII*, pages 203–217, Amsterdam, The Netherlands, 2001. IOS Press.

[32] T. Quatrani. *Visual Modeling with Rational Rose and UML*. Addison-Wesley, 1998.

[33] M. Richters and M. Gogolla. On formalizing the uml object constraint language ocl. In *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, volume 1507 of *LNCS*, pages 449–464. Springer, 1998.

[34] J. Robbins, D. Hilbert, and D. Redmiles. ARGO: A Design Environment for Evolving Software Architectures. In *Proceedings of ICSE'97*, pages 600–601, Boston, MA, May 1997. ACM Press.

[35] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[36] J. Seemann and J. Gudenberg. Extension of UML Sequence Diagrams for Real-Time Systems. In *Proceedings of the UML'98*, volume 1618 of *LNCS*, pages 225–233, Mulhouse, France, June 1998.

[37] J. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International Series of Computer Science. Prentice-Hall International, Englewood Cliffs, NJ, March 1992.

[38] The precise UML group. http://www.cs.york.ac.uk/puml/.

[39] E. Wang, H. Richter, and B. Cheng. Formalizing and Integrating the Dynamic Model within OMT. In *Proceeding of ICSE'97*, pages 45–55, Boston MA, May 1997. ACM Press.

[40] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise modeling with UML*. Addison-Wesley, 1999.