Journal of Universal Computer Science, vol. 6, no. 11 (2000), 1071-1087
submitted: 16/6/00, accepted: 19/9/00, appeared: 28/11/00 © Springer Pub. Co.

Use of E-LOTOS in Adding Formality to UML

Robert Clark
(University of Stirling, Scotland, UK
rgc@cs.stir.ac.uk)

Ana Moreira
(Universidade Nova de Lisboa, Portugal
amm@di.fct.unl.pt)

Abstract: E-LOTOS, a new version of the ISO standard specification language LOTOS, is currently being developed. We describe how it can be used to give a formal meaning to, and to discover inconsistencies in, UML models. As part of this work, we give mappings from UML constructs to E-LOTOS. Emphasis is placed on dealing with UML use case, class and interaction diagrams as these play the dominant part in the development of a UML analysis or high-level design model. Requirements are usually inconsistent and incomplete and we deal with how this can be modelled in a formal language.
Key Words: UML, E-LOTOS, formal modelling, inconsistent specifications.

1 Introduction

Although the use of formal methods within the software development process is a major academic research topic, its impact on industrial practice continues to be disappointing. Even semi-formal methods have not had the impact that might have been hoped for. However, there are signs that things are changing. Object-oriented programming languages are becoming the dominant implementation paradigm. An interesting feature of object-oriented program development is that it is one area where analysis and design methods have a high profile, not only amongst academic researchers, but also amongst practitioners.

Many different object-oriented analysis and design methods have been proposed and they differ markedly in their degree of formality. The major advance is that there is now general agreement on a standard notation to be used in the object-oriented analysis and design phases, namely the Unified Modeling Language (UML) [see Rumbaugh et al. (1999)]. UML is a notation rather than a method, but having an accepted standard notation means that it is now clear when two methods differ in their proposed process rather than in just the way in which they express the same idea. It also leads to better tool support as tool builders can concentrate their energies on a single notation.

UML is not a formal notation although it does include state machines as one of its modelling techniques. The widespread use of UML makes it imperative that its semantics are well defined and understood [see Kent et al. (1999)]. Work has been done on the semantics of its class diagrams [see France (1999)], the pUML Group are a focus for research on the precise use of UML [see Evans, Kent (1999)], and an associated Object Constraint Language (OCL) has been designed [see Warmer, Kleppe (1999)].

Our research is into the introduction of formality into the object-oriented analysis process and we have developed the Rigorous Object-Oriented Analysis Method (ROOA) [see Moreira, Clark (1994), Moreira, Clark (1996), Clark, Moreira (1999)]. A major aim of this work has been to show how formal methods can complement methods such as OMT and OOSE which are widely used in industry [see Rumbaugh et al. (1991), Jacobson (1992)]. It is mainly the notations used in OMT and OOSE that have been combined to form UML.

2 E-LOTOS

LOTOS is an ISO standard formal specification language [see Brinksma (1988)] with a new version called E-LOTOS currently under development [see ISO/IEC (2000)]. E-LOTOS has a behavioural part, defined using a process algebra based on a combination of CSP [see Hoare (1985)] and CCS [see Milner (1989)], and a data typing part that is in the style of a functional language. LOTOS tools enable a specification to be analysed for syntactic and semantic errors and the specification can form the basis for formal reasoning. Our original development of ROOA used the 1988 LOTOS standard and the object-oriented notation was that of OMT [see Rumbaugh et al. (1991)], but it is straightforward to replace LOTOS with E-LOTOS and OMT with UML. We have developed a prototype translator from UML to LOTOS[1].

E-LOTOS has several advantages over LOTOS. For example:

- its data typing part is much easier to use,
- it has a better integration between the data typing and behavioural parts,
- it has subtyping in the data typing part,
- it has exception-handling facilities,
- it incorporates explicit control structures which makes it easier to represent UML interaction diagrams and state machines,
- it can specify real-time behaviour,
- it includes a module construct corresponding to a package in UML.

[1] Work supported by Systems Engineering Research Institute (SERI), Teajon, Korea

As LOTOS and E-LOTOS specifications are executable, simulation tools can be used to demonstrate their behaviour and show up inconsistencies, contradictions, ambiguities, misunderstandings and omissions in the requirements [see Eertink, Wolz (1993)]. When LOTOS or E-LOTOS is used to model an object-oriented system, it can therefore be used to validate the requirements and to give feedback to the requirements capture. That, of course, requires us to have a mapping from object-oriented constructs to their equivalent in LOTOS or E-LOTOS and that forms an important part of ROOA.

There are, in fact, three different aspects to our work:

- The addition of rigour to the object-oriented development process and how use case, class and interaction diagrams can be used in the development of a formal specification.
- The use of E-LOTOS to give a formal meaning to, and uncover inconsistencies in, a particular specification expressed in UML.
- Provision of a mapping from UML constructs to E-LOTOS to provide a semantics for certain aspects of UML. As well as determining the semantics of the constructs in each kind of diagram, we deal with the integration of the views provided by the different UML diagrams.

It is these last two aspects that are emphasised in this paper.

3 UML Models and E-LOTOS Specifications

3.1 How UML is Used

UML is a notation used in analysis and in both high-level and low-level design. A major feature of a requirements specification or of a high-level design is that it purposefully ignores implementation detail. A given high-level design can then be realised by different implementations. Using a formal specification language such as E-LOTOS complements UML well because E-LOTOS gives us precision, while encouraging us to stay at a high level of abstraction so that we are not tempted to become involved prematurely in implementation details.

Let us consider the way in which UML is used in analysis or high-level design and, even more importantly, how it is not used. Most, if not all, such UML models contain a class diagram, a set of use cases and a set of either sequence or collaboration diagrams. However, the actual parameters of the messages in sequence diagrams will usually either be omitted or be symbolic variables rather than actual values. Also, diagramming notations such as state diagrams are often not involved because, in the early stages of development, it is important to model the operations that an object offers and the operations that it requires from other objects rather than the details of its internal behaviour. Our work therefore concentrates on use case, class and interaction diagrams so that it is possible to have a formal meaning for UML models that only contain these components.

It is not that E-LOTOS is unsuitable for representing state diagrams; in fact it can be argued that, of all the UML models, state diagrams are the most straightforward to represent in a process algebra like E-LOTOS. However, we believe that in the analysis and high-level design phases, the most important use of a UML model is to provide a suitable architecture and that the most important behavioural feature is the way that the objects in this structure interact with one another. That is best shown by means of either sequence or collaboration diagrams.

3.2 UML Models and their Relation to ROOA

UML is composed of a large number of different models. For example, use case diagrams model users' expectations; class diagrams model static structure; interaction diagrams (sequence and collaboration diagrams) model dynamic object interactions while state diagrams model the internal behaviour of individual objects. The different diagrams are used in conjunction to capture requirements and to model a solution.

Although a UML specification typically contains a single use case model to describe the user requirements and a single class diagram to show the static structure of the specification, a large number of separate sequence, collaboration and state diagrams are required to model dynamic behaviour. The ROOA approach builds a single formal model, originally using LOTOS and, more recently, E-LOTOS. This model not only specifies the behaviour of the proposed system, but also shows its static structure. A single unified formal specification therefore corresponds to the set of models used in UML.

Moreover, to help us guarantee that the final formal object-oriented specification is according to the requirements, we also formalise the interactions between the actors and the system, i.e. the use case model. Both specifications are then composed in parallel and the final result executed so that we can check that they display the same behaviour.

Whenever we have a set of different models, problems arise of ensuring their compatibility. UML CASE tools can indicate some situations where models are incompatible, e.g. where an interaction diagram uses an operation that has not been defined in the corresponding class diagram, but they do not show up instances where the models define inconsistent or incompatible behaviour. If, on the other hand, a single formal specification is constructed from a set of UML models, then it is possible to identify logical inconsistencies and contradictions.

This is especially so when the resulting specification is executable as is the case with LOTOS and E-LOTOS.

The class diagram is the central co-ordinating component of any UML model and so the structure of our E-LOTOS object-oriented specifications is derived directly from class diagrams. However, class diagrams show static structure, not dynamic behaviour and, when dealing with semantics, we must focus on behaviour. As it is a process algebra, E-LOTOS is well suited to the definition of dynamic behaviour. However, it is essential that the definition of dynamic behaviour fits in, is compatible with and can be interpreted within the static structure defined from the class diagram. Formally modelling the constructs of a UML class diagram in conjunction with the use case and sequence diagrams is therefore a central part of our work.

4 UML Concepts in E-LOTOS

4.1 Classes and Objects

A class is specified by an E-LOTOS process definition and an object is represented by a process instance. The formal model produced by applying the ROOA method is a LOTOS or E-LOTOS specification that describes behaviour in terms of a set of communicating concurrent objects, i.e. as a set of communicating process instances. Each object has a distinct identity. We model this in E-LOTOS by allocating each object a unique constant when the object is instantiated. This constant is known as the object identifier and is held as a parameter of the process representing the class.

An E-LOTOS process is like a black box and its externally observable behaviour is its interactions with other processes. Processes interact by synchronising on events at a gate. In an event, a value declaration has the form `!v`, where `v` is a value expression, while a variable declaration has the form `?v` where `v` has been declared to be a variable of some type. If a process offers the following event at gate `g`:

```
g(!2, !true)
```

and another process offers the event:

```
g(?x, ?b)
```

where `x` has been defined to be an `Int` and `b` to be a `Bool` then, as the two events have the same structure, the processes may synchronise on these events. During synchronisation, we have communication as the value `2` is bound to the variable `x` and `true` to the variable `b`.

Fig. 1 shows a simple UML class diagram describing part of a simple warehouse system. The warehouse includes an inventory which maintains a list of

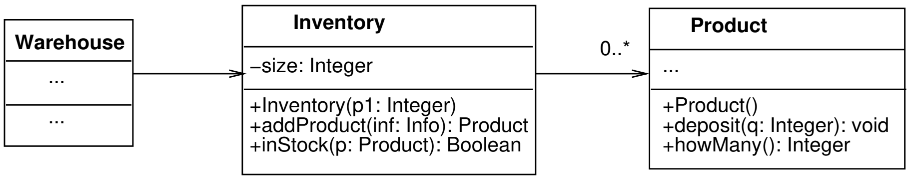

Figure 1: *Simple Class Diagram*

products. This is represented in the diagram by classes `Warehouse`, `Inventory` and `Product` with arrows showing the navigability of their associations.

The outline E-LOTOS definition corresponding to `Inventory` is:

```
process Inventory[g, gProduct](id: InventoryId, p1: Int) is
   var size: Int := ...,
       productListId: ListProductId := [] in
      loop
         var p: ProductId,
             inf: Info in
               g(!addProduct, !id, ?inf);
               ...
               g(!rtn_addProduct, !id, any: ProductId)
            []
               g(!inStock, !id, ?p);
               ...
               g(!rtn_inStock, !id, any: Bool)
         endvar
      endloop
   endvar
endproc (*Inventory*)
```

The private attributes of an object are modelled as the local variables of an E-LOTOS process. That provides information hiding. The first parameter in a process heading specifies the object identifier while the others correspond to the parameters of the constructor in UML. We describe the record structure of an object identifier such as `ProductId` later. Abstract data types (ADTs) for `Int` and `Bool` are defined in the E-LOTOS library while the `Info` ADT is defined in the data typing part of the specification.

Parameters or returned values that are references to objects are modelled as object identifiers. An association is modelled as an attribute that is either an object identifier or, as in this case, a list of object identifiers depending on the multiplicity of the association. As generic lists are in the E-LOTOS library, we can define a list of `ProductId` as:

```
type
   ListProductId is List of ProductId
endtype
```

We use the navigability information in a class diagram to determine the direction of an association. Hence, in our example, `Inventory` knows about `Product` objects and so its process definition has an attribute that is a list of `ProductId`, but `Product` does not know about `Inventory` and so the definition of `Product` does not have an `InventoryId` attribute. We require a bi-directional association to be represented as two unidirectional associations.

Public operations are offered as alternatives using the E-LOTOS choice operator `[]`. In E-LOTOS, we can use a loop in the process body to ensure that once an operation has been carried out, the process is again ready to satisfy a call of one of its operations. In LOTOS, we had to instantiate a new process instance.

Operation `inStock` returns a Boolean value, but the class diagram does not specify which value. We deal with this using the LOTOS construct **any**`: Bool` which pattern matches with any value of type `Bool`. This can lead to *value generation* in which an arbitrary value is chosen nondeterministically.

4.2 Message Passing

E-LOTOS processes communicate by synchronising on events offered at gates. We model message passing (i.e. the call by a client object of an operation offered by a server object) by event synchronisation. We use a stylised structure for the events. They have the following form:

<gate>(<message name>, <object identifier>, <optional parameters>)

Event synchronisation in E-LOTOS is symmetric, there is no concept of a client and a server. However, we *model* the client and server processes differently. The object identifier, for example, always refers to the server object and all the operations offered by a server class are represented by event offers at a single gate (our convention is always to use the formal gate `g` in the process definition of the server). When a client makes calls of operations offered by an object of a class such as `Product`, the process definition of the client class shows this occurring at formal gate `gProduct`.

For two process instances to communicate, they must be composed using an E-LOTOS parallel operator. Our convention is that when a client object of a class such as `Inventory` is composed with a server object of a class such as `Product`, we name the actual gate `agProduct`. Hence the composition of an instance of the client `Inventory` with an instance of the server process `Product` has the form:

```
Inventory[agInventory, agProduct](...)
|[agProduct]|
Product[agProduct](...)
```

The E-LOTOS parallel operator `|[agProduct]|` specifies that the objects communicate with each other on gate `agProduct`. Note that in the instantiation of process `Inventory`, the actual gates `agInventory` and `agProduct` have been substituted for the formal gates `g` and `gProduct` respectively.

When a server object receives a call, it may make subsequent calls to carry out the required service. We therefore model an operation as two event synchronisations. The call is modelled by an event that includes an operation name such as `addProduct`, while the return is modelled by an event that includes the operation name `rtn_addProduct`. Returning a value is modelled as a parameter of the `rtn_addProduct` event. The corresponding client object must have consecutive `addProduct` and `rtn_addProduct` events, i.e. it must not carry out any actions between making the call and receiving the response.

4.3 Object Creation

Objects can be created dynamically using object generators. For example, an object generator for class `Product` has the form:

```
process Products[g](max: Nat) is
   var newId: ProductId := (class => theProduct, obj => max) in
      g(!create, !newId);
      ( Product[g](newId)
        |||
        Products[g](max + 1)
      )
   endvar
endproc (*Products*)
```

The operator `|||` is the E-LOTOS interleaving operator. A new process `Product` is instantiated with a unique object identifier of type `ProductId` and process `Products` is reinstantiated so that it can handle the next creation request. Object identifiers are modelled as extensible records with at least two fields. One field specifies the class and the other specifies the instance number of an object. We model a null object reference by an object identifier whose instance number is `0`.

The overall behaviour defined by an E-LOTOS object-oriented specification is defined by composing process instances in an E-LOTOS behaviour expression. When a class has a single instance then it is created by an instantiation of its defining process in the behaviour expression. When a class may have multiple instances, the composition includes an instance of an object generator rather than an individual process instance. In our simple example [see Fig. 1], the composition might be:

```
Warehouse[agWarehouse, agInventory](...)
|[agInventory]|
Inventory[agInventory, agProduct](...)
|[agProduct]|
Products[agProduct](1)
```

When an object of class `Inventory` initiates the creation of a `Product` object as, for example, part of its `addProduct` operation, the E-LOTOS definition of `addProduct` will contain an event such as:

```
gProduct(!create, ?pId)
```

During synchronisation, the variable `pId` is set to the value of the new object identifier allocated in the object generator. In that way, the `Inventory` object is informed of the identity of the new `Product` object so that it can take part in future communication.

4.4 Aggregation and Composition

In our simple example, we could have decided to model the association between `Inventory` and `Product` as a UML *whole-part* association, i.e. as an aggregation. We model aggregation as the *whole* knowing about the *part* and so that would have required no change in the E-LOTOS specification. That is because defining a UML association to be an aggregation gives conceptual information, but its semantics remain the same as an ordinary association.

Composition, on the other hand, does change the semantics. Composition is a strong form of aggregation in which a component object may be part of only one composite object. If the composite object is destroyed then all its component objects are destroyed with it. We model a composite class as an E-LOTOS process whose body includes the instantiation of its components.

For example, consider the composition relationship depicted in Fig. 2. The E-LOTOS process which specifies that composition is:

```
process Product[g](id: ProductId) is
   var pInfoId: PInfoId := (class => thePInfo, obj => 1),
       pPartListId: ListPPartId := [] in
      hide agPInfo, agPPart in
         <definition of Product body>
         |[ agPInfo, agPPart]|
         ( PInfo[agPInfo](pInfoId, ...)
```

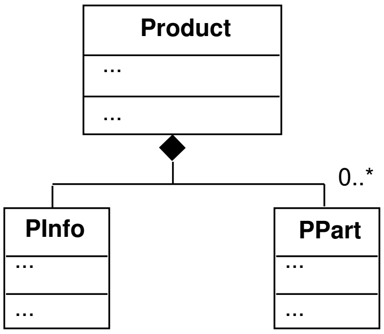

Figure 2: *Composition*

```
            |||
            PParts[agPPart](1)
          )
       endhide
    endvar
where <definitions of component classes>
endproc (*Product*)
```

This captures the intended UML semantics. Component objects are either created at the same time as the composite or they are created by the composite object. Also, component objects can only be accessed by the composite object of which they are a part. The object identifiers of components within different composite objects do not have to be distinct as they are local to a particular composite. Also, clients of class `Product` are not aware that it is a composite class.

The E-LOTOS interleaving operator `|||` shows that `PInfo` and `PPart` objects do not interact directly with one another. All their interactions are with the composite through the hidden gates `agPInfo` and `agPPart` and these interactions cannot be observed from outside the composite. Notice that due to the multiplicity of the associations between the composite class and each of its components, `PInfo` is directly instantiated while for `PPart` we need the object generator `PParts`.

4.5 Dynamic Binding

Dynamic binding can be modelled in E-LOTOS using object identifiers. Extensible records are a feature of E-LOTOS that did not exist in LOTOS. The object identifier `ProductId` is defined as:

```
type ProductId is
   (class => Product, obj => Nat, etc)
endtype
```

The presence of `etc` means that this is an extensible record type with *at least* two fields. E-LOTOS uses structural type equivalence and an extensible record is the supertype of its extensions. The object identifier of a subclass is an extension of the object identifier of its superclass, e.g.

```
type EdibleProductId is
   (class => Product, obj => Nat,
      subclass1 => EdibleProduct, etc)
endtype
```

A value of type `EdibleProductId` can be used anywhere that a value of type `ProductId` is expected and so when, for example, an object of class `Inventory` makes a call of operation `deposit` this is modelled in the E-LOTOS process `Inventory` as:

```
gProduct(!deposit, !pId, !inf)
```

The type of the variable `pId` may be either `ProductId` or `EdibleProductId`. Even when its type is `ProductId`, the current value of `pId` may be of type `EdibleProductId`. Synchronisation can therefore occur with a server object that is a subclass of `Product` and the decision is delayed until execution time and depends on the current value of `pId`.

5 Behavioural Models

A UML specification typically consists of a single class model to define the static structure, but uses different dynamic models to specify behaviour. UML uses:

- use case diagrams to describe users' expectations of the services that the system will provide,
- sequence and collaboration diagrams to describe object interactions,
- state machines to describe the internal behaviour of single objects,
- activity diagrams to show flow of control.

As these diagrams can overlap in the information that they provide, they can give contradictory information. Hence, as well as determining mappings for constructs in each kind of diagram, we must ensure that the information provided by each behavioural model is compatible both with the information provided by the other behavioural models and with the class diagram. A further problem is that, to form a complete UML specification, a significant number of diagrams must be constructed for each of the different kinds of dynamic model. That can lead to further inconsistency problems.

The different kinds of UML model are very useful in providing different views to help us understand and model a problem. However, if we wish to be able to reason about the overall behaviour of a specification then it is much easier if we have a single integrated formal model.

5.1 Use Case Model

A UML model not only describes the structure and behaviour of an object-oriented system, it also describes how it is to be used, i.e. the environment in which the system is to exist. It is the UML use case model which defines how a system is to be used. We can formalise the UML use case model by building an E-LOTOS specification of the interaction between the actors, i.e. the external users of the system, and the system itself. These interactions can be represented in terms of events and so an event-oriented formal language such as E-LOTOS is ideal. We call this specification the user-centred model. We have shown how this can be done in LOTOS [see Clark, Moreira (1999)].

The user-centred model can then be used in conjunction with the UML class and interaction diagrams to help us create the formal object-oriented specification. As both the user-centred model and the object-oriented specification are expressed in E-LOTOS, the two models can be composed to form a single formal model. In that way, we can ensure that the behaviour expected by the use case model is compatible with the behaviour offered by our final formal specification.

5.2 Sequence Diagrams

In our object-oriented E-LOTOS model, a process instance defines the behaviour of a single object. An important part of this behaviour is its interaction with other objects. We do not just map from UML constructs to a set of E-LOTOS process definitions, the overall behaviour of a UML model is specified by the composition of E-LOTOS process instances. We therefore produce a single integrated E-LOTOS model that defines static structure as well as the complete dynamic behaviour.

Object interactions are defined in UML sequence or collaboration diagrams. As these two kinds of diagram are equivalent, we will only consider sequence diagrams. Typically in a sequence diagram, an object receives a message and, as a consequence, may send one or more messages to other objects. This is shown in Fig. 3. An object of class `Inventory` receiving the message `addProduct` from an object of class `Warehouse` is modelled by the E-LOTOS process instances that are modelling these objects synchronising on an event `addProduct` at gate `agProduct`. The sequence diagram shows that as a consequence of receiving message `addProduct`, the `Inventory` object first creates a `Product` object and then sends a `deposit` message to it.

Earlier, we showed the outline process definition for `Inventory` generated from the class definition. It included the outline code:

```
g(!addProduct, !id, ?inf);
...
g(!rtn_addProduct, !id, any: ProductId)
```

Using the information in the sequence diagram, we can now expand this to:

```
var pId: ProductId in
  g(!addProduct, !id, ?inf);
  gProduct(!create, ?pId);
  gProduct(!deposit, !pId, !inf);
  gProduct(!rtn_deposit, !pId);
  ...
  g(!rtn_addProduct, !id, !pId)
endvar
```

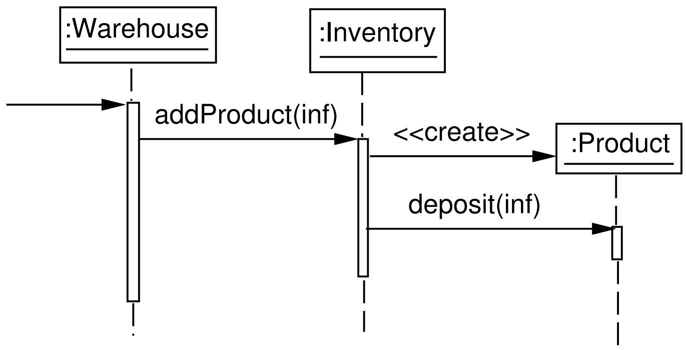

Figure 3: *Simple Sequence Diagram*

Typically, a UML model does not specify particular values for attributes or for the actual parameters of messages in sequence diagrams. This is dealt with in E-LOTOS by either using symbolic values or by using value generation to nondeterministically generate an arbitrary value. When tracing execution using actual values, the number of possible paths becomes so large that the problem becomes intractable. By using symbolic or arbitrary values, we concentrate on showing that the paths exist and that is the main need during analysis and high-level design. An important point is that this can be done without requiring extra detail to be added to a UML model. After all, the main advantage of UML is that it enables the requirements to be organised and understood at a high level of abstraction.

5.3 Dealing with Inconsistencies

Requirements are often inconsistent or incomplete and that will be reflected in the initial UML model. It is important that potential inconsistencies can be modelled in our specification language, but this should be done in such a way that potential problems will be highlighted.

A simple example is where a second sequence diagram shows an `Inventory` object responding differently to the receipt of an `addProduct` message, e.g. it might show that it sends a `howMany` rather than a `deposit` message to a `Product` object. The choice could depend on the current state of `Inventory` or on the value of an `addProduct` parameter, and this can be shown in a sequence diagram by attaching a guard condition to the message. However, a UML model will not always show that level of detail.

Alternatively, having two possible subsequent behaviours may be due to an inconsistency in the requirements. When the reason for the alternative behaviours is given, it can be modelled using the E-LOTOS `case` operator, when it is not, we can allow either behaviour by writing:

```
g(!addProduct, !id, ?inf);
gProduct(!create, ?pId);
(  gProduct(!deposit, !pId);
   gProduct(!rtn_deposit, !pId);
   ...
 []
   gProduct(!howMany, !pId);
   gProduct(!rtn_howMany, !pId, any: Int);
   ...
);
g(!rtn_addProduct, !id, !pId)
```

An E-LOTOS construct of this nature indicates a possible problem. By integrating the information from different diagrams into a single process definition we highlight such possible problem areas and this is a major help in enabling possible inconsistencies to be identified. The problem with a large UML model is not in resolving inconsistencies, it is in identifying them.

Another example is where two sequence diagrams specify a different order in which a series of operations should be carried out. It may be that the order is unimportant or that the order depends on the object state, but it is also possible that it is due to an inconsistency in the requirements. The E-LOTOS interleaving operator can be used to model this situation:

```
a; (b ||| c ||| d); e
```

Here, event `a` occurs first followed by `b`, `c` and `d` in any order and finally followed by `e`. Again, the presence of such a construct indicates a possible problem area. Other inconsistencies or omissions will be less obvious, but will normally lead to deadlocks in the E-LOTOS specification. These can be detected using simulation tools.

E-LOTOS is therefore an ideal tool for identifying and resolving contradictions that arise from the use of different UML models. By providing a single integrated explanation of behaviour rather than having the explanation distributed over separate models, we are able to ensure that each possible path of object interactions is complete and consistent thereby highlighting problems of incompleteness or inconsistency in the UML model.

6 Concurrency

In the early stages of developing a UML model, we do not need to state whether objects are active or passive and it can be useful to regard a UML model as the description of a set of communicating autonomous objects. That is the approach that we follow and, although an E-LOTOS model is expressed in terms of a set of communicating concurrent objects, an eventual solution may be sequential. As a UML model is refined, explicit decisions may be made that certain objects are to be active, i.e. have their own thread of control. Many of the main problem areas in UML semantics are concerned with active objects and, due to its concurrent nature, E-LOTOS is well suited to resolving such difficulties.

In some situations, concurrent behaviour is part of the problem definition rather than just a possible solution. An obvious example is where we are dealing with a distributed system such as an Internet application. Modelling all objects as E-LOTOS process instances, i.e. as potential active objects, allows us to delay decisions about which objects should be active in an eventual implementation and about the precise physical deployment of the objects.

7 Conclusions

Our work describes mappings from UML constructs to the formal language E-LOTOS and describes situations where E-LOTOS improves on LOTOS. In particular, it shows the effectiveness of E-LOTOS in integrating the static UML model with the different dynamic UML models so that inconsistencies between the different models can be identified. We concentrate on how UML is actually used in analysis and high-level design and so our work is centred on use case, class and interaction diagrams. The strength of UML is that it abstracts away from detail and that it provides an architecture from which an eventual solution can be developed. Our E-LOTOS specifications provide the same architectural framework and deal with behaviour at the symbolic level to ensure that the required collaborations are possible rather than dealing with particular values.

While UML constructs several models to deal with dynamic behaviour, our E-LOTOS specification integrates, in a single model:

- the interactions between the actors and the system (given in UML by the use case model),
- the static structure (given in UML by the class diagram) and
- the dynamic aspects of a system (given in UML by several interaction diagrams).

By obtaining a single formal model, we can identify inconsistencies in UML models that would be very hard to identify otherwise.

A major advantage of a formal description technique such as LOTOS or E-LOTOS is that the specifications they provide are executable. Therefore, rapid prototyping tools can be used to validate the final result. However, although there is good tool support for LOTOS, currently a major drawback in using E-LOTOS is the lack of tool support.

References

[Brinksma (1988)] Brinksma, E.: "LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour"; ISO 8807, 1988.

[Clark, Moreira (1999)] Clark, R.G., Moreira, A.M.D.: "Formal Specifications of User Requirements"; Automated Software Engineering, 6, 3 (1999), 217-232.

[Eertink, Wolz (1993)] Eertink H., Wolz D.: "Symbolic Execution of LOTOS Specifications"; Formal Description Techniques V, North-Holland (1993), 295-310.

[Evans, Kent (1999)] Evans, A., Kent, S.: "Core Meta-Modelling Semantics of UML: The pUML Approach"; Proc. 2nd Int. Conf. on the Unified Modeling Language, LNCS 1723, Springer-Verlag (1999)

[France (1999)] France, R.: "A Problem-Oriented Analysis of Basic UML Static Modeling Concepts"; Proc. OOPSLA'99, ACM Sigplan, 34, 10 (1999), 57-69.

[Hoare (1985)] Hoare, C.: "Communicating Sequential Processes"; Prentice-Hall (1985)

[ISO/IEC (2000)] ISO/IEC: "Enhanced LOTOS"; ISO/IEC 15437 (2000)

[Jacobson (1992)] Jacobson, I.: "Object-Oriented Software Engineering"; Addison-Wesley (1992)

[Kent et al. (1999)] Kent, S., Evans, A., Rumpe, B.: "UML Semantics FAQ"; Object-Oriented Technology, LNCS 1743 (1999), 33-56.

[Milner (1989)] Milner, R.: "Communication and Concurrency"; Prentice-Hall (1989)

[Moreira, Clark (1994)] Moreira, A.M.D., Clark, R.G.: "Combining Object-Oriented Analysis and Formal Description Techniques"; Proc. ECOOP '94, LNCS 821, Springer-Verlag (1994), 344-364.

[Moreira, Clark (1996)] Moreira, A.M.D., Clark, R.G.: "Adding Rigour to Object-Oriented Analysis"; Software Engineering Journal, 11, 5 (1996), 270-280.

[Rumbaugh et al. (1991)] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: "Object-Oriented Modeling and Design"; Prentice-Hall (1991)

[Rumbaugh et al. (1999)] Rumbaugh, J., Jacobson, I., Booch, G.: "The Unified Modeling Language Reference Manual"; Addison Wesley (1999)

[Warmer, Kleppe (1999)] Warmer, J., Kleppe, A.: "The Object Constraint Language"; Addison-Wesley (1999)