Always and *Eventually* in Object Requirements

Paul Gibson, Dominique Méry,
LORIA-UMR n^o 7503-CNRS &
Université Henri Poincaré
BP 239, 54506 Vandoeuvre-les-Nancy, France
email: (gibson/mery)@loria.fr

(This work was supported by the contract —
n$^\circ$:96 1B CNET-FRANCE-TELECOM & CRIN-CNRS URA262.)

May 27, 1998

Always and *Eventually* in Object Requirements

Abstract

Object oriented models and methods encompass a set of techniques which have been, and will continue to be, applied in the successful production of complex software systems. The methods are based on the simple concepts of abstraction, encapsulation, classification and polymorphism. The formal verification of *logical* properties of such models is difficult to integrate into the traditional operational view. Furthermore, most, if not all, of the object oriented formalisms are based on the specification of safety properties and, as such, they do not provide an adequate means of expressing liveness conditions.

We examine a mixed semantic framework in which a state-based object oriented semantics is extended with the concepts of *always* and *eventually*. These semantics have been integrated to provide a set of re-usable *fair object* templates whose graphical representation can be formalised by the client for use during synthesis and analysis of their requirements. Proof of invariant and eventuality properties is done by translation to PVS and TLA, respectively. The utility of such an approach is illustrated through a simple telephone feature case study.

1 Introduction

The goal of our work is to integrate the temporal logic concepts of *always* and *eventually* into an object oriented requirements language such that the formal verification of this class of properties can be automated. The diagram below illustrates our semantic framework.

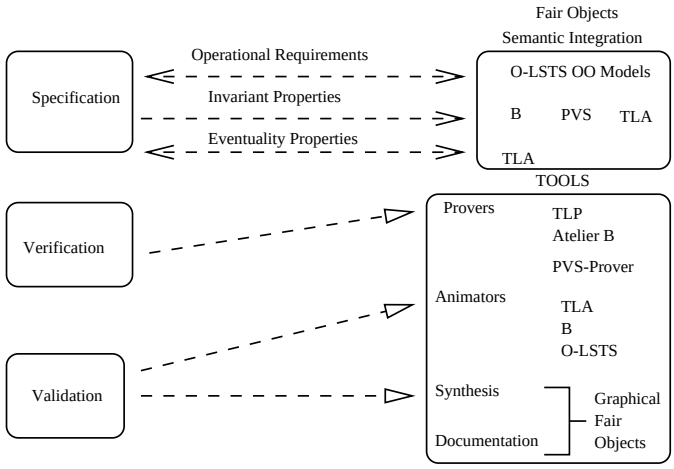

Figure 1: Integrating logical and operational requirements

Examination of all parts of our system is beyond the scope of this paper. Instead, we concentrate on three parts:

- In section 2 we give an overview of the formal object oriented semantic model and explain how we intend to extend this model with the notions of *always* and *eventually*.

- In section 3 we examine the specification and verification of class invariant properties using PVS.
- In section 4 we examine the the specification and verification of object eventuality properties using TLA

To finish, in section 5, we illustrate the application of our approach within the domain of telephone feature specification.

2 Object Oriented Semantics: An Overview

2.1 Formalising Semantics

Labelled state transition systems are often used to provide executable models during analysis, design and implementation stages of software development [7, 9, 10]. In particular, such models are found in the classic analysis and design methods of [3, 6, 8]. However, a major problem with state models is that it can be difficult to provide a good system (de)composition when the underlying state and state transitions are not easily conceptualised. The object oriented paradigm provides a natural solution to this problem. By equating the notion of class with the state transition system model and allowing the state of one class to be defined as a composition of states of other classes, we obtain a means of specifying such models in a constructive fashion.

We adopt a simple object-labelled state transition system semantics (O-LSTS[1]) which regards an object as a state transition machine [13]. These semantics permit us to view objects at different levels of abstraction. Firstly, using an abstract data type (ADT) we can specify the functionality of an object at a level of abstraction suitable for requirements capture[23]. Secondly, we can transform our ADT requirements into a parameterised process algebra (LOTOS[2, 30]) specification for the design stage. Finally, as we approach an implementation environment, we can view the objects in our designs as clients and servers in a distributed, concurrent network. At each of these levels of abstraction we provide a means of specifying and verifying *always* and *eventually* properties. In this paper, we report only on the requirements capture phase of development. Furthermore, we do not deeply explore the OO issue of inheritance but stay, in general, within a composition-based framework.

2.2 A Simple Telephone

The O-LSTS specification of a telephone is given below.

The `Phone` is a class which uses three other classes in its specification (`ID`, `signal` and `on-off`). Every O-LSTS class encapsulates its behaviour behind an *interface* which defines a

[1] Although there exist other similar and better known semantics, we have been using O-LSTS for many years and continue our promotion of it.

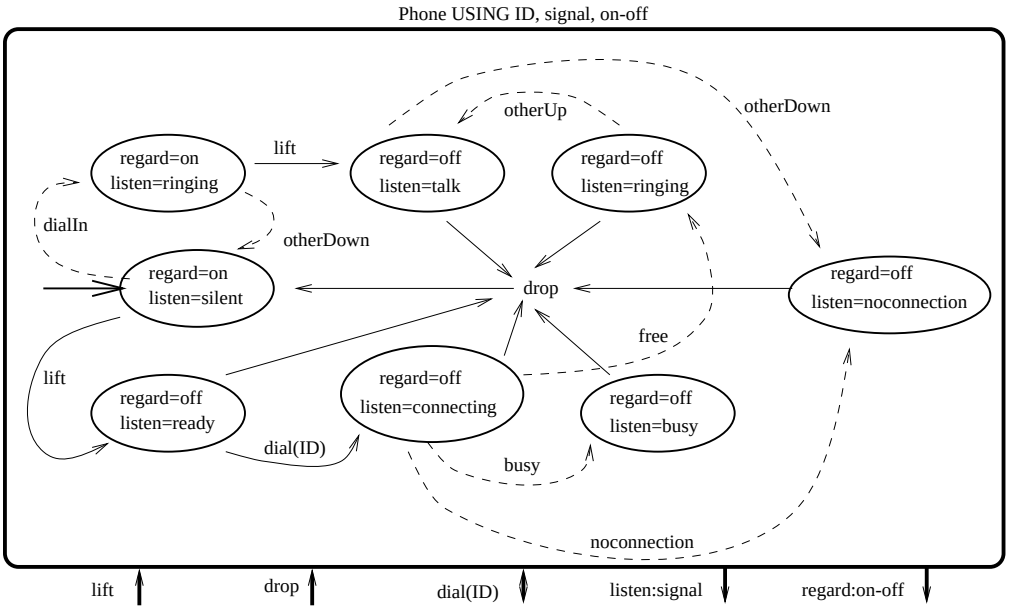

Figure 2: An O-LSTS telephone specification

set of *services* which every object in the class must offer. The `Phone` class *interface* offers five services:

- `lift`: allows the user to lift a phone which is currently on hook.
- `drop`: allows the user to drop a phone which is currently off hook.
- `dial(ID)`: allows the user to dial the number specified by the `ID` parameter.
- `listen`: corresponds to the current `signal` being given to the user.
- `regard`: corresponds to whether the phone is currently on or off hook.

We say that `lift`, `drop` and `dial` are *transformer* services because they change the state of the phone. *Transformers* are not necessarily available to the user of the object at all times; for example, `dial` is *enabled* only when in the state `off-ready`. The services *listen* and *regard* do not change the state of the phone but do return some value to the phone user. They are called *accessors*, and in an O-LSTS model such services are always enabled. (In the O-LSTS model there is a third type of service which combines accesor and transformer functionality — these are called *duals* and they are not used in the `Phone` specification.)

The `Phone` class contains eight *member objects*[2]: all phone instances[3] must be in one of these eight different (memeber) states. The diagram below illustrates the difference between the object instances and object members.

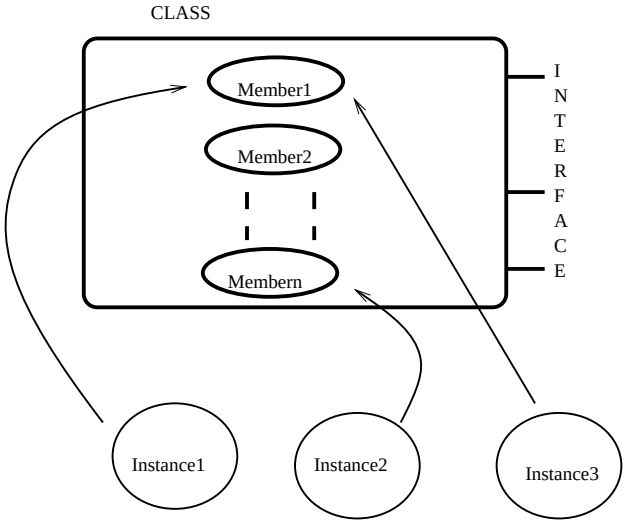

Figure 3: Object instances and members

For each class member we must defi ne the value of the accesor operations. Furthermore, we must defi ne the next state which arises after a transformer operation is requested. The O-LSTS specifi cation is naturally represented as a state transition diagram. For the phone we have chosen the initial state to be `on` and `silent`. Starting from this state we can now validate a suite of test traces with the client. These traces are animated using a graphical representation[18] of our object states and service requests.

To conclude this simple example, we note that there are dotted state transitions which do not correspond to interface services. These *internal* transitions, `free`, `busy`, `noconnection`, `otherUP` and `otherDown`, represent the interaction between the phone and the telephone network, which the phone user cannot see or influence. For example, when I phone someone the 'system decides' whether the other person is free, busy or a connection cannot be made. From the point of view of the user, the choice between these transitions is nondeterministic.

[2]Member objects are sometimes known as substates or (dynamic) subclasses.

[3]The term object can normally be used to mean either a class instance or a class member, without risk of ambiguity.

2.3 Other semantic issues

There are many other aspects to the O-LSTS semantics which the simple phone class does not illustrate, for example:

- Composition — where a structured class can be defined as a composition of other classes. In section 5 we define the `Phone` as a composition of a `Signal` and an `On-Off`.
- Invariants — for a structured object we can define relationships between the state of component objects which must always be true. In section 5 we use the structure `Phone` specification and define invariants between the state of each of the components. For example, if the `signal` is `talking` then the `on-off` must be `off`.
- Subclassing[4] — where one class can be defined as an *extension* or *specialisation* of another class. In [27] we define a hierarchy of telephone features in this way.
- An extension adds new services to a class which respect the invariant properties of the original class. For example, a simple extension of the `Phone` could provide a *recall last number* service.
- A specialisation strengthens the invariant properties of a class and guarantees that all the services of the original class respect the new invariant property. The `Phone` cannot be specialised because there is no *partition* of the member set which could be closed, with respect to the invariant, for all the transformer services [15].
- Exceptions — for defining *exception cases* which should not yet be fully extrapolated in the initial requirements capture phases, otherwise we would risk making premature implementation decisions.

The validation of a similar O-LSTS model of a system of telephones is reviewed in [18]. The graphical synthesis and analysis of user-oriented O-LSTS requirements models is reviewed in [20]. The verification of the completeness and consistency of an O-LSTS class specification without invariants is reviewed in [13]. In this paper, we concentrate on the problems of:

- Specifying and proving the invariant properties.
- Specifying liveness properties on the nondeterministic transitions and using these properties in a constructive manner to prove user *eventuality* requirements.

[4] See [15] for details.

3 Invariants

There are many ways in which we could extend our O-LSTS models with the concept of invariant properties. We chose to keep the notion of invariant as simple as possible by restricting our attention to the relation between the state of components of structured objects.

3.1 Structured Classes: A Composition Example

In the diagram below, we illustrate how an O-LSTS can be specified as a composition of other O-LSTSs. The intended system behaviour is as follows:

> *The system is to be composed from two stack components (each offering LIFO behaviour through services* `push` *and* `pop`*). The system will offer a service* `push` *for pushing an element onto the first stack, a service* `pop` *for popping an element of the second stack, and a service* `move` *for popping an element of the first stack and pushing it onto the second stack.*

The graphical O-LSTS model of the system is given below:

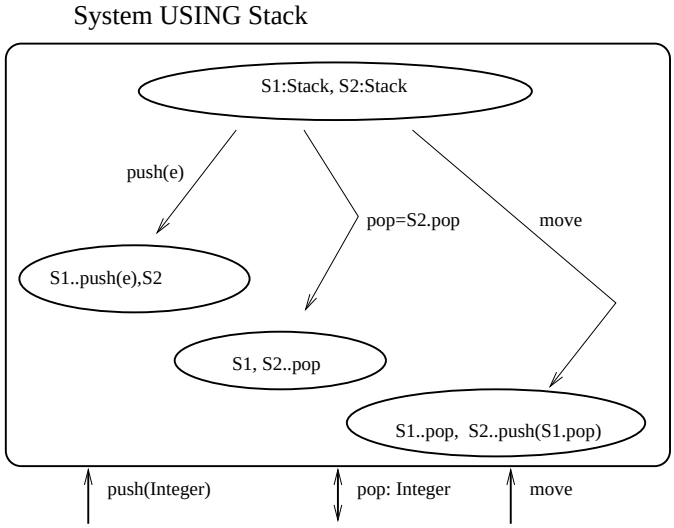

Figure 4: A system composition of two stacks

The equivalent textual specification, written in OO ACT ONE [15] illustrates the abstract data type *flavour* of our language:

```
CLASS System USING Stack
STRUCTURE: Stack, Stack
DUALS: pop-> Integer
```

```
TRANSFORMERS: move, push(Integer)
EQNS
System(S1, S2)..push(e) = System(S1..push(e), S2);
System(S1, S2)...pop = System(S1, S2..pop) RETURNS S2.pop;
System(S1, S2)..move = System(S1..pop, S2..push(S1.pop))
ENDCLASS System
```

The following comments explain the notation:

- Given an object `obj` which offers an accessor `acc`, the value returned from `obj` when the `acc` service is requested is represented by `obj.acc`.
- Given an object `obj` which offers a transformer `tr`, the state of the `obj` after the `tr` service is requested is represented by `obj..tr`.
- Given an object `obj` which offers a dual `dl`, the state of the `obj` after the `dl` service is requested is represented by `obj..dl` and the value returned is represented by `obj.dl`.
- Service parameters have their class-types inferred automatically.

From the formal specification we can deduce a number of things about the `Stack` class (which are verified automatically):

- The elements stored on the stack must be integers[5] since the input and output parameters for the system are integers.
- A `Stack` offers a `push` transformer.
- A `Stack` offers a `pop` dual.

This example illustrates the composition mechanism. However, it does not illustrate the need for an invariant: there is no *obvious* requirement relating the state of each stack component which must always be true for the system class to be *correct*.

3.2 Invariant Example

To illustrate the notion of an invariant property, consider a class `Joints` which specifies a class of students who study two subjects as part of their degree. The class has two `Subject` components. There is one accessor service for testing if a `Joints` student studies a given subject. The textual specification is given below:

```
CLASS Joints USING Subject
STRUCTURES: Subject, Subject
```

[5] In fact, with the polymorphic semantics we can also allow the elements to be members of any subclass of the integer class.

```
ACCESSORS: studies(Subject) -> Bool
EQNS
Joints(Subject1, Subject2).studies(Subject3) =
(Subject1.eq(Subject3)).or(Subject2.eq(Subject3))
ENDCLASS Joints2
```

With such a specification we have an additional requirement which is best represented by an invariant property: A `Joints` student must *always* study two different `Subjects`. Without such an invariant, `Joints(Maths, Maths)` could be seen as a valid member of the `Joints` class. To specify that this is forbidden, we have two options:

- Explicitly list all joint degree combinations which are valid.
- Define an invariant property on the `Joints` class structure to specify that the first component cannot be the same as the second component.

The second option is better since an explicit statement of the invariant property improves the *understandability* of the specification. This invariant is easily validated by the client and it is the analyst's responsibility to verify that it is *correct*; i.e it cannot be broken by any of the object state transformers.

Using an invariant property follows the principle of encapsulation and makes the specification simpler to extend. For example, if the `Subjects` class is to be extended to include a new subject then this change should be possible without affecting the classes which use the `Subjects` class. This is not possible with the first option, in which the principle of encapsulation has to be broken for the behaviour of the degree class to be well defined. With an invariant there is no need to change the `Joints` class every time a new `Subject` is created. The `Joints` behaviour is respecified in class `Joints2` to incorporate the new invariant property.

```
CLASS Joints2 USING Subject
STRUCTURES: Subject, Subject
INVARIANTS: Joints2(Subject1, Subject2) REQUIRES Subject1.neq(Subject2)
ACCESSORS: studies(Subject) -> Bool
EQNS
Joints2(Subject1, Subject2).studies(Subject3) =
(Subject1.eq(Subject3)).or(Subject2.eq(Subject3))
ENDCLASS Joints2
```

Invariant properties which are not formally verified introduce the possibility of 'run time' errors in an execution model. For example, consider an *extension* to the `Joints2` class in which a new transformer operation allows either of the `Subject` fields to be changed. Now, a transformer service may result in a new state which does not fulfil the invariant property. To avoid such problems we must:

- Formally verify that all transformer (and dual) operations are closed with respect to the invariant property. In other words, if an invariant is true before a state transformation then it must be true after the state transformation.
- Check that all subclassing relations defined by *extension* introduce only new services that respect the existing invariants.
- Prove that all subclassing relationships defined by *specialisation* do not introduce an invariant which can be broken by the original services.

The `Joints2` invariant is called a *structure invariant* because the invariant property is defined in terms of properties of the components of a structure of the class. It is also often desirable to be able to define an invariant on a whole class rather than a structure in a class. The syntax of such a *class invariant* is illustrated by the `MathsJoints` specification below.

```
CLASS MathsJoints SPECIALISES Joints2 WITH
INVARIANTS: MathsJoints.studies(Maths)
ENDCLASS MathsJoints
```

Note that the class invariant mechanism is simply syntactic sugaring for defining sets of structure invariants. Furthermore, we now have an explicit subclassing relationship[6] between `MathsJoints` and `Joints2`, which we do not have between `Joints2` and `Joints`.

The examples above do not illustrate other important aspects of the O-LSTS model which are influenced by our invariant concept:

- Classes can be defined to have recursive structures.
- Classes can be defined to have multiple structures.
- Classes can be verified as having *static structure* such that transformer operations cannot change the composition of an object.
- Classes can be allowed to have *dynamic structure*.
- Invariants for non-structured members of a class (the literals) can be verified through simple syntactic inspection.

We now address the proving of such composition invariant properties in O-LSTS specifications. There are two parts to this problem. Firstly, we must consider the *property language* for specifying invariants. Secondly, we must consider how to use a theorem prover for the *automated verification* of these properties.

[6] This is important when we consider polymorphism.

3.3 Invariant Property Language

The language used for expressing invariant properties is very simple. An invariant must be a boolean expression whose sub-expressions can be constructed using:

- Any operations of the boolean class
- Any of the component objects of the structured class
- Any of the accessor services offered by the component objects

In this way, we enforce the encapsulation of the state of our component objects. The containing object cannot *look inside* a component to check if an invariant is true, it can utilise only the accessor services provided by its components. Furthermore, with these simple rules, we have an operational means of testing any given invariant for any object in the class. Thus, we can check our invariants dynamically as the system executes: if we generate a state in which the invariant is broken then we can say that the system specification is *incorrect*.

During validation, it is possible to test invariants dynamically. Using model-checking techniques, we can also check that all traces being tested respect our invariant requirements. However, with large systems, especially those containing complex subclassing hierarchies, it is almost impossible to verify our invariants in this way. A modular approach requires a means of:

- Statically proving invariants for each class in a system
- Re-using invariant proofs in subclasses of classes which have already been verified

To reach this goal, we decided to translate our specifications towards a more proof-theroetic framework.

3.4 Translation to PVS for verification

The proof verification system (PVS) [25, 26] is a powerful tool for the verification of specifications. It consists of a specification language, a number of predefined theories, a theorem prover, various documentation tools, and several examples that illustrate different methods of using the system in several application areas. PVS's highly expressive specification language facilitates powerful automated deduction; for example, some elements of the specification language are made possible because the typechecker can use theorem proving.

The specification language of PVS is based on classical, typed higher-order logic. The built-in types such as the booleans, integers, reals, and ordinals, can be extended by user-defined types. The data type constructors include functions, sets, tuples, records, enumerations, and recursively-defined abstract data types, such as lists and binary trees.

As our O-LSTS semantics are implemented by translation to ACT ONE [15], we concerned ourselves with how to map the ACT ONE specifications to the PVS data typing language. This mapping turns out to be a simple syntactic transformation and requires little explanation. Figure 5 illustrates, informally, the syntactic manipulation which needs to be done during translation.

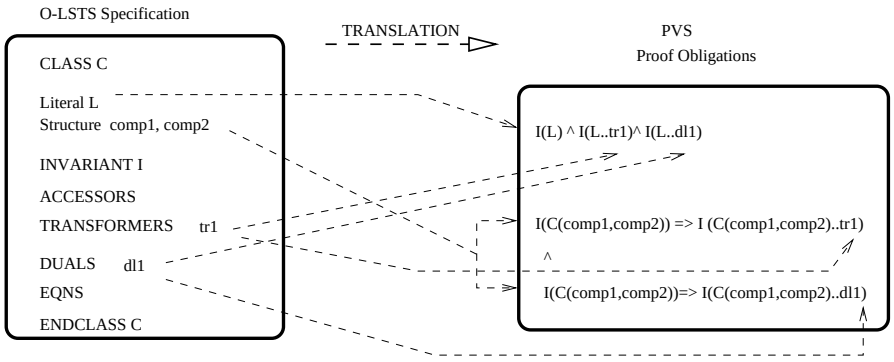

Figure 5: Translating to PVS

The PVS theorem prover provides a collection of powerful inference routines that can be called upon interactively, under user guidance, within a sequent calculus framework. The primitive inferences include propositional and quantifier rules, induction, rewriting, and decision procedures for linear arithmetic. In our experience, perhaps due to our being *PVS beginners*, proving our invariants requires a great deal of tool interaction and our proofs continue to be constructed in an ad-hoc manner.

PVS *does* provide a means of formalising user-defined procedures that can be re-used during the proof process. These can be combined with primitive inferences to yield higher-level proof strategies. However, our experience (or lack of experience) has not led us to discover such useful strategies for our O-LSTS invariant proofs.

Proof scripts provide a useful documentation facility: that can be edited, attached to additional formulas, and rerun. In the future we hope to be able to use such scripts to obtain a compositional proof method based on our classification hierarchies.

4 Liveness, fairness and eventuality

4.1 Nondeterminism

Using the O-LSTS semantics, in co-operation with the proof of invariant properties, we can specify and verify *safety* properties: which state that during system execution *something bad* will never happen. We do not have a means of specifying *liveness* properties: which state that *something good* will *eventually* happen.

Consider the specification of a shared database object. This database must handle multiple, parallel requests from clients. The order in which these requests are processed is required to be nondeterministic. This is easily specified without liveness. However, if the requirements are now refined to state that every request must be *eventually* served (this is a fairness requirement which we cannot express in a safety-only semantic framework). Our only choice is to *over-specify* the requirement by defining how this fairness is to be achieved (for example, by explicitly queueing the requests). This is bad because we are enforcing implementation decisions at the requirements level.

4.2 A Nondeterministic O-LSTS example

Let us reconsider the system of two stacks, seen in section 3.1. We change the system by requiring the `move` service to be nondeterministic. This is illustrated in the figure below, where we also show the composition diagram for the system:

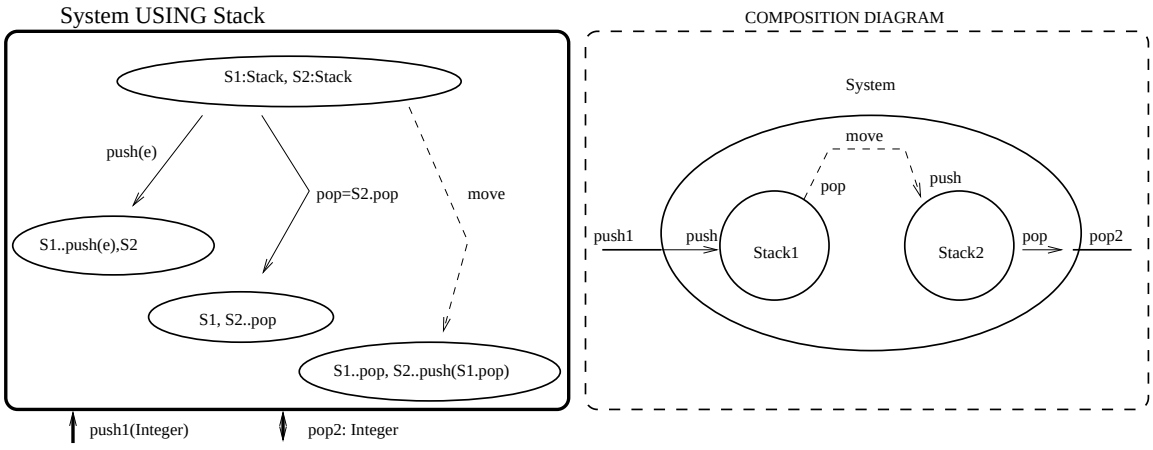

Figure 6: Nondeterminism in a system of 2 stacks

With such a specification, the system environment can never force an element to be moved from the first stack onto the second stack, since the `move` is no longer part of the external *interface* of the class. The system may decide to do such moves whenever it likes; but it may also decide to never perform a move. A typical high-level requirement for such a system is:

> All elements that get popped onto the first stack must *eventually* be pushed onto the second stack, by the move transition.

To specify and verify such a requirement, without making premature implementation details, we require some sort of temporal semantics for expressing the *liveness* property.

4.3 Fair Objects: TLA and O-LSTS

TLA [22] provides a simple and effective means of expressing liveness properties. The semantics incorporate the notions of *always* (represented by the $\Box$ operator) and *eventually* (represented by the $\Diamond$ operator). Using these, we can specify different categories of liveness within the object oriented framework. The ability to model nondeterminism at different levels of abstraction is the key to TLA's utility in requirements modelling. Unfortunately, TLA does not provide the means for easily constructing and validating initial customer requirements. By combining TLA and object oriented semantics we can alleviate these problems. It is beyond the scope of this paper to examine the integration of the two different semantic models: see [19] for more details. The important concepts, found in the case study in section 5, are:

- *Weak fair objects*, where any internal transition which is always enabled must eventually be taken.
- *Strong fair objects*, where any internal transition which is enabled an infinite number of times must eventually be taken.
- *Possible fair objects*, where any internal transition which may be enabled an infinite number of times will eventually be taken.
- *Progressive fair objects*, where any object which is part of a structured system will eventually carry out an internal action if such an action is enabled.

4.4 Eventuality requirements

Within our O-LSTS framework, we formalise the fairness requirements as eventuality constraints which hold between object servers and their object clients.

We have identified five different types of *client eventuality requirements* which provide high level reusable concepts, and can be automatically formalised using TLA. A client may require that a service request be serviced **immediately**. A client may require that a service is carried out **eventually**. A client may wish a service to be performed **immediately on condition** that if it cannot be done without delay then it will be informed. A client may wish a service to be performed **eventually on condition** that if it cannot be guaranteed to be done in a finite period of time then it will be informed. The client wants the service but places **no eventuality** requirements on when the service must be performed (if ever).

Each service that a server offers can be classified dynamically, during execution. An **immediate** service is now enabled. An **eventual** service is guaranteed to be enabled in a finite period of time. A **possible** service may be enabled but it depends on the environment of the server in forcing certain state transitions; no internal action can make the service impossible

to fulfil eventually. A **probable** service can possibly be enabled but it depends on some internal nondeterminism; no external service request can make it impossible to service the request (event ually). An **impossible** service will never be enabled.

Within the fair object system, the *server* properties can, we hope, be made to match the *client* requirements. This is the job of an *interface protocol* which seperates clients from servers. In a formal model of requirements we can prove that eventuality needs are fulfilled by servers. This is the role of the TLA theorem prover TLP[11].

4.5 Proof verification using TLP

TLP is a tool for assisting in writing, proving and reasoning about specifications using TLA, the Temporal Logic of Actions. It provides the user with a means to write specifications and structured proofs in a natural deduction style. TLP is based on two underlying tools, namely LP, a 'theorem prover for a subset of multisorted first-order logic' designed by Steve Garland and John Guttag at MIT as a part of the Larch Project[12], and a small tool for automatic checking of linear time temporal logic based on boolean decision diagrams, implemented by David Long. It translates from the TLP language to the languages of the *back-ends*, and there is an interactive *front-end* based on GNU Emacs.

TLP can be used for deriving proofs of invariance properties[7] eventuality properties under fairness assumptions. TLP provides a way to organize proofs following proof rules of TLA in the natural deduction style. Fundamentally, an eventually property proof requires at least a weak fairness on the global transition relation and the weak fairness proof rule states that at least one action must be enabled which leads to the desired eventual property. In [16] the translation of object oriented SDL models to TLA, for the verification of liveness properties, explains in some detail the approach we adopted.

5 Telephone feature case study

In this section we motivate the need for *always* and *eventually* semantics within the domain of telephone feature specification. The examples are explained informally: more formal treatments are found in [19, 16, 14, 17]. Here we use the examples to justify the need for a formal means of verifying temporal requirements in an object oriented specification architecture.

5.1 The feature interaction problem

Features are observable behaviour and are therefore a requirements specification problem [31]. We concentrate on the domain of telephone features [4, 5]. The feature interaction problem is

[7] We decided not to use TLP for this because the translation to PVS was already complete adn, we believe, is better suited to this task.

stated simply, and informally, as follows:

> A *feature interaction* is a situation in which system behaviour (specified as some set of features) does not as a whole satisfy each of its component features individually.

Most feature interaction problems can be (and should be) resolved at the requirements capture stage of development[14]. The telephone feature examples in this section are taken from a large list of specifications which we have developed using our *fair object* concepts[27]. Telephone feature specification is well suited to our semantic approach because it requires a high degree of structuring to cope with the highly compositional and incremental nature of such systems; and there is a clear need for fairness requirements [17].

5.2 Always and Eventually in a Telephone Specification

A telephone *invariant*

The telephone can be specified to consist of two object components: the *signal* and the *hook*. (In the specification in section 3 we did not specify the phone in such a compositional manner because we had not yet met the structuring mechanism.) A state invariant which defines a relation between the hook component and the signal component will include the following requirement:

> *Always*, if I am talking with someone then the telephone must be off hook

This can be formalised as a boolean invariant:

```
(signal = talking) => (hook = off).
```

In the simple finite telephone system, this is directly verifiable by checking that it is true in all the states. Since there are only 8 states we can do this *by hand* — translating to PVS is not recommended when all the states in a system can be easily verified.

A telephone *eventuality*

Even the simple telephone can benefit from the specification of an *eventuality* requirement. In the O-LSTS phone specification it is possible to be in the state `off-connecting` forever. In this state we have just dialled a number and we must wait for the system to tell us if the number is free, busy or unattainable. A high level eventuality requirement is:

> I will *eventually* leave the state `off-connecting` even if I do not force the state transition myself.

Using TLA, this requirement is represented by a weak fairness on the noconnection event:

$$WF(noconnection).$$

If the network *takes too long* to decide if the requested line is free or busy, then we will eventually get a noconnected signal.

5.3 Always and Eventually in a POTS Telephone Network

The Plain Old Telephone System (POTS)[20] is the standard model for the communication between two phone users.

A POTS *invariant*

In the network of many different telephones we use the POTS invariant to specify relations between pairs of phones. For example, a simple POTS state invariant requirement is:

Always the number of phones talking is even.

This is proved by showing that it is true in the initial state (where all phones are off and ready). Then we show that all possible state transitions maintain the required property (if it is true before the action occurs then it is true after the action occurs). This property cannot be checked through an exhaustive search of a system of an unbounded number of phones. It *can* be checked by proving that all transitions are closed with respect to the invariant. Consequently, we had to translate this into PVS and utilise the prover to verify the correctness of our specifi cation.

A POTS *eventuality*

Within the telephone network, the POTS component controls the synchronisation between pairs of phone connections. When one phone hangs up, for example, the POTS system must inform the other phone user by changing the state of their phone. The updating of state information must always eventually be carried out. For example:

When I dial the number of a phone which is free then it will *eventually* ring

These type of *eventuality* requirements can be expressed using the notion of *progression*, which arises from the way in which system components are considered as concurrent objects. In such systems we wish to specify that each of the components is *fairly scheduled*. In other words, internal actions of these components must eventually be executed so that their behaviour *progresses*. No component can stay in the same state forever when an internal action leaves that state.

5.4 A Feature Interaction *Eventuality* Example

Consider two well known features: call forwarding and answer-machine.

Call Forwarding

Informally, call forwarding can be used to transfer an incoming call to another line. Thus, if I am not at home I can, for example, forward my calls to my portable phone. Here the transfer is an internal action and it must be completed in a finite period of time, and so we once again have an *eventuality* requirement. In the specification, we make the requirement conditional on the forwarding number being connectable.

Answer machine

Without giving the actual specification, it is clear that an answering machine requires liveness semantics. The standard functionality is for the phone to ring for a finite period of time and then a message to be taken. An *eventuality* requirement is that when I ring someone with an answering machine I will eventually talk with them or get to leave a message. This requirement is proven with TLP, using a fair object model of the answering machine.

Answering machine with Call forwarding

Combining these two features illustrates the difference between a standard object model and one with *eventuality* semantics. Without fairness, when I telephone someone with an answering machine who has forwarded their call to another number then, if there is no answering machine at the second phone, I cannot leave a message if they do not reply. With the fair object requirements model, I must be able to leave a message if the person doesn't reply (independent of whether the call is forwarded or not). Thus, if the telephone at the forwarded address does not have an answering machine my specification requires that the call control is returned to the original phone so that a message can be left there.

5.5 A Feature Interaction *Invariant* Example

Consider two well known features: call waiting (CW) and call forward when busy (CFB).

Call Waiting:

When I activate CW its functionality is enabled only when I am talking to someone else. If I am talking to someone and a third person tries to call me then I can chose to hold them. Now I am talking to one person and holding another. I now have the ability to switch the talking and held persons. I can hang up on both calls at once and become `on-silent` in the normal `Phone` state. Similarly, either of the two other parties can hang up and I return to the state `off-talking`.

Call Forward When Busy:

With CFB, all my incoming calls are forwarded to another number if my phone is already in use. (This is usually done so that calls are forwarded to an answering service when a line is busy.)

CW with CFB

CW requires that I will *always* be given the choice of holding an incoming call if I am talking to someone when it arrives. However, the CFB feature may be executed *before* this choice can be made and so the CW requirement is contradicted in the system containing both features. Furthermore, a fairness condition in CFB states that *eventually* the incoming call will be forward. However, if CW is executed, the incoming call may be held by the user and never be forwarded. Thus we have another contradiction. This is illustrated in the figure below.

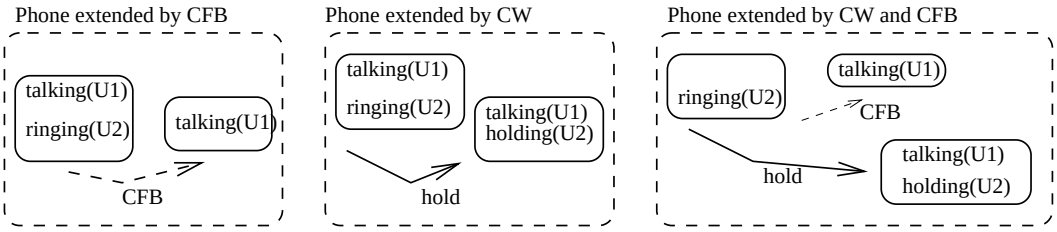

Figure 7: An Invariant Interaction

6 Conclusions, Related Work and Future Work

We have shown the utility of introducing the concepts of *always* and *eventually* into an object oriented framework. These concepts can be used in the formal specification of user requirements, and passed to theorem provers for verification. The proof process is not automatic: the tools require interaction with an *intelligent* user. However, the process is formal and the tools do help us to structure, and document, our verifications. We hope to be able to find re-usable proof strategies so that less burden is placed on the analyst when verifying these logical properties. Our approach has been successfully applied in the domain of telephone feature specification; we believe it is equally applicable in other problem domains where requirements are modelled as a system of interacting objects.

In [24] we see an approach, based on the B method[1], to detecting feature interactions as the breaking of invariant properties. In [28] we see a different technique for modelling features using object oriented semantics, which lack a means of specifying and verifying logical properties. In [29] we see the advantages of having a solid proof-theoretic environment when verifying the absence of interactions between features, but they do not address the problems of validating their models. In [21] we see another integration of operational object oriented models with logical requirements; their approach is based on SDL and the ObjectGeode tool but does not consider liveness issues.

Our method is far from being complete. We intend to try and relate our semantics to the UML so that we may approach a wider audience. Furthermore, we are not happy with our in-

ability to prove complex invariants in highly structured systems: in particular using the theorem provers is a less than pleasant experience which is far from being automated. Finally, we realise that the main weakness in our approach is the *mixed semantics* which have not been formally integrated.

References

[1] Jean-Raymond Abrial. *The B Book*. Cambridge University Press, 1996.

[2] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[3] G. Booch. *Object oriented design with applications*. Benjamin Cummings, 1991.

[4] L. G. Bouma and H. Velthuijsen, editors. *Feature Interactions In Telecommunications*. IOS Press, 1994.

[5] K. E. Cheng and T. Ohta, editors. *Feature Interactions In Telecommunications III*. IOS Press, 1995.

[6] P. Coad and E. Yourdon. *Object oriented design*. Prentice-Hall (Yourdon Press), 1990.

[7] L. Constantine. Beyond the madness of methods: System structure methods and converging design. In *Software Development 1989*. Miller-Freeman, 1989.

[8] Brad Cox. *Object oriented programming: an evolutionary approach*. Addison-Wesley, 1986.

[9] Geoff Cutts. *Structured system analysis and design method*. Blackwell Scientific Publishers, 1991.

[10] T. DeMarco. *Structured analysis and system specifi cation*. Prentice-Hall, 1979.

[11] U. Engberg. *TLP Manual-(release 2. 5a)*-PRELIMINARY. Department of Computer Science, Aarhus University, May 1994.

[12] Stephen J. Garland and John V. Guttag. A guide to lp: the larch prover. Technical report, MIT Laboratory for Computer Science, 1991. Also available as Digital Equipment Corporation Systems Research Center Research Report 82.

[13] J. Paul Gibson. Formal object based design in LOTOS. Tr-113, University of Stirling, Computing Science Department, Stirling, Scotland, 1994.

[14] J. Paul Gibson. Feature requirements models: Understanding interactions. In *Feature Interactions In Telecommunications IV*, Montreal, Canada, June 1997. IOS Press.

[15] J.Paul Gibson. *Formal Object Oriented Development of Software Systems Using LOTOS*. Tech. report csm-114, Stirling University, August 1993.

[16] J.Paul Gibson and D. Méry. *Telephone Feature Verifi cation: Translating SDL to TLA+*, pages 103–118. Elsevier Science, 1997.

[17] Mermet Gibson and Méry. Feature interactions: A mixed semantic model approach. In *Irish Workshop on Formal Methods*, Dublin, Ireland, July 1997.

[18] Paul Gibson. An object oriented requirements capture and analysis environment. Technical Report CRIN-98-R-010, CRIN, January 1998.

[19] Paul Gibson and Dominique Méry. Fair objects. In *OT98 (COTSR)*, Oxford, May 1998.

[20] Paul Gibson and Yassine Mokhtari. Pots: An OO LOTOS specification. Technical Report CRIN-98-R-013, CRIN, January 1998.

[21] Lejeune Kerbrat, Rodriguez-Salazar. *Interconnecting the ObjectGEODE and Caesar-Aldébaran toolsets*, pages 475–491. Elsevier Science, 1997.

[22] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[23] B. Liskov and Zilles S. Programming with abstract data types. In *ACM SIGPLAN Notices*, number 4 in 9, pages 50–59, 1974.

[24] B. Mermet and D. Mery. Detection of service interactions: An approach with b. In *AFADL97*, Toulouse (France), 1997.

[25] S. Owre, N. Shankar, and J. B. Rushby. *The PVS Specification Language*. Computer Science Laboratory, SRI International, CA, February 1993.

[26] S. Owre, N. Shankar, and J. B. Rushby. *User Guide for the PVS Specification and Verification System*. Computer Science Laboratory, SRI International, CA, February 1993.

[27] B Mermet P. Gibson and D. Méry. Specification of services in a compositional temporal logic. Rapport de fin du lot1 du marche no 961B CNET-CNRS CRIN, CRIN, 1997.

[28] H Prehofer. An object oriented approach to feature interaction. In *Feature Interactions In Telecommunications IV*, Montreal, Canada, June 1997. IOS Press.

[29] Rochefort and Hoover. An exercise in using constructive proof systems to address feature interactions. In *Feature Interactions In Telecommunications IV*, Montreal, Canada, June 1997. IOS Press.

[30] van Eijk, Vissers, and Diaz. *The Formal Description Technique LOTOS*. North-Holland, Amsterdam, 1989.

[31] Pamela Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer Magazine*, pages 18–23, August 1993.