Mapping Object Descriptions to Behaviours[1]

Working paper

O.M. Nierstrasz

University of Geneva
Centre Universitaire d'Informatique
12 Rue du Lac, CH-1207 Geneva, Switzerland
UUCP: mcvax!cernvax!cui!oscar
BITNET/EARN: oscar@cgeuge51

Abstract

There is a lack of good formalisms and tools for describing the semantics of object-oriented and concurrent programming languages. We propose a computational model for objects in which *events* are synchronous communications between concurrent agents, *computations* are partial orderings of events, and *behaviours* are the possible event unfoldings in which an agent, or a system of concurrent agents, may participate. Furthermore, we introduce a language called *Abacus* for defining executable behaviour expressions, and we speculate how this language may be used as part of a practical system for defining the formal semantics of programming languages.

1 Introduction

To date there have been many attempts to provide a formal framework for defining the semantics of programming languages. Although some tools have been developed, notably VDM [Bjørner and Jones 1978], there are none that have gained the wide-spread acceptance of parser-generators, which use a formal definition of the syntax of a language to automatically generate the front-end of a compiler or interpreter (i.e., the part *not* concerned with implementing the semantics of a language). The consequence is that the bulk of new programming languages are designed without a formal semantics, and language implementors and programmers are therefore left with inadequate indications of what language constructs or programs mean.

Our position is that recent work in object-oriented and concurrent programming languages provides some insight that may lead to a solution to this problem. Work in programming language semantics has gradually shifted from operational descriptions to more abstract language definitions, with the intent of capturing what programs really mean in a machine non-specific manner, rather than how they might be implemented. Object-oriented languages capture the essence of this principle, in that objects are *encapsulations*. The essence of an object is its input/output "behaviour", not its internal data structures or the implementation of its operations.

We propose that the semantics of programming languages, particularly that of concurrent, object-oriented languages, would be best described in terms of a computational model that formal-

[1]In *Active Object Environments*, ed. D.C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva June 1988, pp. 106-113.

izes the notion of behaviour in terms of synchronous communication amongst systems of concurrent agents. Instead of understanding programs as functions that map inputs to outputs, we would view both programs and inputs as behaviours (i.e., "objects") that yield a computation when they are concurrently composed.

Furthermore, we propose that a simple language for describing behaviours would be an appropriate target for object denotations. A tool for translating object descriptions to a behavioural language could be used not only for automatically generating language interpreters, but also for providing a formal basis for writing compilers that preserve the intended language semantics.

In the following sections we shall outline the desired properties of our computational model, we will give some examples of directly executable behaviours written in a language we have implemented called *Abacus*, and we shall outline the practical problems that we hope to address with our approach.

2 Towards a Computational Model for Objects

We shall begin with an informal discussion of our computational model for objects (which we call *CMO*) and its relationship to other, comparable models.

Computation, as we normally understand it in machine terms, refers to the translation of inputs encoding the statement of a class of problems to outputs encoding their solutions. A *program* encoding the translation algorithm may be either viewed as part of the input, or as a separate, finite control. The complete description of the input and program encodings is the initial configuration (state) of the system, or abstract machine. The execution of the program can be viewed as sequence of steps that transform the state of the system.

Our view is that the correct way to understand the "state" of a system is directly in terms of how it may be transformed into future states, i.e., in terms of its *behaviour*. Furthermore, we feel that in order to capture the progress of a concurrent computation, we must view the state-transformation events as being partially ordered in time.

The model we propose can be informally described as follows. A *computation* is a partial order of *events*, where events are names constructed from a finite event alphabet. A computation may be infinite (non-terminating), but any initial component must be finite. A *system* is a collection of *agents*, whose concurrent composition will yield a (possibly empty) computation. The *behaviour* of an agent is a set of *input* or *output offers* (i.e., guards [Dijkstra 1975]) with respect to some event, followed by a replacement behaviour in case that event takes place. The behaviour of a system is basically the concurrent behaviour of its constituent agents, where matching input and output offers of separate agents yield events in a computation, provided the offers are visible to one another. The replacement behaviour for an agent may be that of a system, thus introducing more concurrency into the computation. A *nil* behaviour, that is, one with no offers, represents termination of an agent. (We shall see a simple example in the following section.)

The idea of this approach is that both programs and inputs are modeled as behaviours. A computation will occur when these behaviours are combined. The end result of a computation will be a new behaviour representing the output. Ultimately, our goal is to give formal semantics of programming languages by showing how programs (or objects, in object-oriented languages) map to behaviours. This approach differs slightly from the standard denotational approach in that we

do not translate programs into functions, but into behaviours that can then be composed with other behaviours. This is especially important in concurrent systems where the behaviour of a program may be affected by that of other programs running in the system.

The model that most closely resembles CMO is Milner's Calculus of Communication Systems (CCS) [Milner 1980]. Since CCS was developed in order to study notions of behavioural equivalence, the model always assumes the existence of a single observer. In CMO, however, multiple observers are essential to make sense of concurrent computations as partial event orderings (each observer sees a different total order of the same computation).

Hoare's model of Communicating Sequential Processes (CSP) also offers a similar notion of behaviour [Hoare 1986]. The main differences are that events are not strictly between pairs of agents, thus permitting synchronization between multiple agents, and that there is an explicit notion of *channels* for communications, thus providing an abstraction for the participants of events.

The approach in CMO to viewing computations as partial orders of events comes from the *Actor* model of computation, [Agha 1986]. The main difference is that actors communicate asynchronously by sending messages, whereas events in CMO represent synchronous communications. (An actor may send a message and proceed to take part in other events before that message has been received, whereas offers in CMO block behaviours until they have been accepted.)

3 A Language for Describing Object Behaviours

CMO is an extremely general model encompassing computations that are not finitely describable (in the same way that there are actor systems for which no finite programs exist). We therefore require a language that captures an interesting subset of possible behaviours. This language should be computationally complete (i.e., with the power of Turing machines), it should be expressive, that is, it should provide a small but powerful set of primitives appropriate for capturing behaviours of the languages whose semantics we wish to describe, and it should be interpretable. This last requirement just expresses the idea that our primitives should not be unrealistically powerful. In this way we can guarantee not only that the languages we describe will be implementable, but that we can automatically generate a prototype implementation by translating programs to the behavioural language.

To give a flavour of the kind of language we envisage, consider the following behaviour description written in *Abacus*, a prototype we have implemented in C using the UNIXyacc and lex compiler-writing tools:

```
True    = ! true . True
        + ? setFalse . False
        + ? setTrue . True
```

This example recursively defines a behaviour `True` using a *behaviour expression*. In the expression, `true`, `setFalse` and `setTrue` are event names, and `True` and `False` are behaviour names. The symbols `?` and `!` followed by event names stand, respectively, for input and output offers on the event names. An offer is followed by a behaviour expression indicating the replacement behaviour, in case the offer is matched *and* the event takes place. A period indicates that a named behaviour follows. The addition symbol (`+`) stands for non-deterministic choice when multiple offers

are matched.

In the example, an agent with the `True` behaviour simultaneously makes one output offer (`!true`) and two input offers (`?setFalse` and `?setTrue`). In case these offers are matched by other agents, one is chosen, the event takes place, and the agent receives the appropriate replacement behaviour named by `True` or `False`. (The communicating agent with the matching offer also replaces its behaviour.)

The `Negate` behaviour looks like this:

```
Negate  = ? false ! setTrue . nil
        + ? true ! setFalse . nil
```

where `nil` is the empty (i.e., terminal) behaviour. An agent with the behaviour `Negate` will accept `false` as an input and output `setTrue`, or it will accept `true` and output `setFalse`. That the replacement behaviours for the `?false` and `?true` input offers here are behaviour expressions rather than named behaviours. (Parsing is left-to-right.)

We can now initiate a computation by instantiating a system consisting the concurrent composition two agents with these behaviours as:

```
[ True & Negate ]
```

This will be executed by the Abacus interpreter, yielding the event sequence `true`, `setFalse` and the final configuration:

```
[ False & nil ]
```

where the `nil` can be discarded. Note that we distinguish between non-deterministic choice between offers and concurrent composition of agents. Communication is permitted across the `&` operator, but not across `+`.

The current power of Abacus is that of finite state automata, i.e., there is only a finite number of reachable states. Dynamic instantiation and parameterization are not currently supported, but the latter would definitely be an asset. Consider the following compact representation of both `True` and `False`:

```
Boolean( x: {true, false} )
        = ! x . Boolean(x)
        + ? setTrue . Boolean(true)
        + ? setFalse . Boolean(false)
```

Note that `x` is a parameter, and not a variable. It must be bound in order to give meaning to the definition. Once it is bound, the event unfolding is completely determined. The only notion of "state" is that of the behaviour itself. In particular, the behaviour `Boolean(true)` is identical to that of `True`, defined above. Parameterization is thus used to simultaneously define both the `True` and `False` behaviours.

It is not clear what set of primitives will be ideal, but we postulate that they should address the following issues: instantiation, concurrent composition, synchronous communication, parameterization, and encapsulation. It should be possible to describe not only the behaviour of standard control structures, but also mechanisms and concepts such as dynamic binding, inheritance and strong-typing. As a guiding principle, we feel that we must restrict our behavioural language to finite system descriptions and event alphabets, but permit arbitrary dynamic instantiation, thus mimicking Turing machines, which have finite descriptions, but unbounded memory.

Instantiation can be handled by introducing the concurrency operator (`&`) into behaviour expressions, indicating that an agent may replace its behaviour by that of two (or more) concurrent agents.

Parameterization may exist in many forms. In our example, the parameter `x` stood for an event name, but parameters could also represent behaviour names, behaviour expressions, or even whether an offer is an input or output offer.

The most difficult issue, however, seems to be how to introduce a simple, yet expressive encapsulation mechanism. CCS offers two extremely powerful mechanisms that address parameterization and encapsulation, namely *relabeling* and *restriction*. The former relabels events in a behaviour expression, and the latter hides them. But relabeling is strictly more powerful than parameterization, since the former can be used to capture the behaviour of an unbounded stack, whereas the latter cannot. Relabeling and restriction, although powerful and expressive, pose some problems if our behavioural language is to be efficiently interpretable.

An encapsulation mechanism for Abacus would be used to express that offers should be visible only to certain agents. For example, the system:

```
[ True & Negate & True & Negate ]
```

yields several possible computations. The final output can be any of:

```
[ True & True ]
[ True & False ]
[ False & True ]
[ False & False ]
```

Encapsulation could be used to bind each instance of `Negate` to a separate instance of `True`, thus eliminating non-determinism and forcing a unique computation to result. Note that this computation (i.e., the one resulting in `[ False & False ]`), is a partial ordering of events rather than an event sequence, capturing the fact that the two sequences of `true`, `setFalse` events are concurrent. The same computation may be seen differently by multiple observers. (This is a different notion than that of non-deterministic choice, in which some *particular* event ordering is imposed, and seen in the same way by all observers.)

Compound events can be introduced to model the idea of channels in CSP, and thus obtain a form of encapsulation. In the example, we would assign each agent a unique name from a name space. Suppose that `N` stands for a set of identifiers (i.e., event names), and that `True` and `Negate` are redefined in terms of compound events:

```
True(y:N) = ! (y,true) . True(y)
          + ? (y,setFalse) . False(y)
          + ? (y,setTrue) . True(y)
```

Then we can re-write our system above as:

```
[ True(a) & Negate(a) & True(b) & Negate(b) ]
```

where `a` and `b` are names in `N`. Now only a single (concurrent) computation may result. The limitation of this approach to encapsulation is that we have no mechanism for generating new names when we have agents that "fork". In order to model the infinite tape of a Turing machine, for example, we need to be able to add new cells (agents) as the tape grows, and be able to tell the cells apart. For this we either need infinite name spaces, or more powerful encapsulation mechanisms (such as those of CCS) to focus our attention to a particular cell.

In addition to a more powerful version of the Abacus language, we would need a semantic mapping language for describing how programs and program fragments are to be interpreted as behaviours. We expect that ideas from denotational semantics will be useful [Tennent 1976; Gordon 1979]

4 Conclusions

The problem we would like to address is typified by the following example. Given a programming language such as *Hybrid*, a concurrent programming language supporting object-oriented language constructs in addition to various forms of synchronous and asynchronous message passing [Nierstrasz 1987], we would like to:

1. give a formal semantics of Hybrid by mapping object descriptions to behaviours,
2. automatically generate a prototype language implementation via an interpreter for behaviour expressions,
3. verify that a compiler that translates Hybrid object descriptions into some target language with its own concurrency primitives correctly preserves the original message-passing semantics.

The third issue requires that the semantics of the target language also be given in terms of behaviours. The problem then reduces to demonstrating that the behaviour of correct programs is always *equivalent* (in some sense yet to be defined) to the behaviour of their translation.

We also see two other applications of this approach. First, a semantic tool can be useful for developing a language in the same way that a parser generator can be used to aid a language designer in developing a consistent grammar for a new language. Both experimentation and formal rigour are simultaneously encouraged, since a prototype language implementation is always available. Second, the tool may be useful for system integration. Given two languages whose semantics we understand in terms of a common computational model, it will be possible to develop interfacing mechanisms

that allow programs in one language to communicate with those of another. For example, we may wish to interface an actor-based system with a user-interface package based on a very different model of concurrency and events.

The research we propose is as follows:

1. to develop a computational model for objects that adequately captures notions of concurrency, communication etc.,
2. to develop a interpretable language for defining behaviours in terms of a small, but expressive set of primitives,
3. to develop a meta-language for defining the semantics of programming languages in terms of behaviours, and for automatically translating programs into the behavioural language, and
4. to investigate notions of behaviour equivalence that can be used to prove compilers correct.

References

[Agha 1986] G.A. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, Massachusetts, 1986.

[Bjørner and Jones 1978] D. Bjørner and C.B. Jones (ed.), *The Vienna Development Method: The Meta-Language*, Lecture Notes in Computer Science 61, Springer Verlag, Heidelberg, 1978.

[Dijkstra 1975] E.W. Dijkstra, "Guarded commands, nondeterminacy, and formal derivation of programs", CACM, vol. 18, no. 8, pp. 453−457, Aug 1975.

[Gordon 1979] M.J.C. Gordon, *The Denotational Description of Programming Languages*, Springer-Verlag, 1979.

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

[Milner 1980] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag, 1980.

[Nierstrasz 1987] O.M. Nierstrasz, "Active Objects in Hybrid", ACM SIGPLAN Notices Proceedings OOPSLA '87, vol. 22, no. 12, pp. 243-253, Dec 1987.

[Tennent 1976] R.D. Tennent, "The Denotational Semantics of Programming Languages", Communications of the ACM, vol. 19, no. 8, pp. 437−453, August 1976.