1 Introduction

Object orientation, as it appears in Simula 67 [1], was motivated by two main concerns: To achieve good structural correspondence between discrete event simulation programs and the systems being modelled. And to provide language mechanisms for the construction of reusable program components while maintaining good computer efficiency. We shall attempt to view these mechanisms in the light of formal specification and reasoning techniques.

Object orientation has proved to be a successful structuring strategy also outside the area of system simulation. This is due to the fact that objects are useful general purpose tools for concept modelling, and can lead to better program decomposition in general, as well as new kinds of reusable program components. It is worth noticing that the class concept of Simula 67 is used to represent “modules” and “packages” as well as object classes. In the following we interpret the term object orientation in this wide sense.

It has been said many times that algorithm and data belong together; the one is meaningless without the other. Together they comprise a dynamic system which exists in time and space. An algorithm needs time to take place and data to operate on; data need space on storage media to exist and associated operations to be accessed and updated. It is reasonable therefore that program components in general should specify data as well as operations. In object oriented programming languages time/space duality is explicit for the two main kinds of program components: a procedure typically interacts with the program environments through data-like parameters, and an object interacts through associated operations. This is illustrated by the following figures, where circles and squares represent (patterns for) operation-like and data-like entities respectively.
If we abstract away the concepts of time and space we are left with the more purely mathematical notions of *functions* and (timeless) *function applications* and *types* of (spaceless) *values*. In Algol-like languages typed values are of insignificant volume, say, one computer word or less, thus one can afford to implement "spacelessness" by allowing value copying and the storage of anonymous intermediate results at the discretion of the compiler. (In contrast potentially high-volume structures like arrays are not treated as spaceless values, but rather as composite program variables.) It might have been prudent to reserve the term "function" for operations without side effects and taking insignificant time, but that is not usually done.

In the following we assume a strongly typed programming language and use the following terminology: A *procedure* is a pattern for timed action sequences (with associated data), called procedure *activations*. A *function procedure* is a procedure which returns a value and has no other effect. A *class* is a pattern for stored data structures (with associated procedures), called class *objects* (or simply objects if confusion with the non-technical term is unlikely). The data components of an object are called representation *variables*. The associated procedures are called procedure *attributes*. We assume that the representation variables are invisible from outside the object, only the procedure attributes should normally be accessible. The reason for this is twofold:

- The representation variables may be assumed to satisfy an invariant, the representation invariant, which is essential for the correct operation of the procedure attributes. By preventing interference from outside one can obtain a kind of correctness guarantee or, rather, if the invariant does get destroyed the error is necessarily located within the object.

- The object as a whole may represent some kind of "abstract" entity, for which the actual representation is or ought to be irrelevant. By hiding it, different representation schemes (classes) for the same abstract concept can be interchanged without hurting the program logic.

It is worth noticing that a simple program variable can be seen as an object with associated operations for assigning and reading its value, the latter usually implicit. Conversely, an object can be seen as a generalized variable, whose value, or "state", is a data record consisting of the values of the representation variables. The procedure attributes are either initializing or updating procedures, or they are "observer functions". The latter are function procedures accessing or computing