1 Introduction

An object and its associated operations may be specified in many ways. One way is to give an abstract representation of the object data structure (viz., representing a queue by a sequence) and the effects of various operations on this abstract representation [3,4,5]. Another way [2] is to leave the representation aspects unspecified but to give a set of equations that relate the effects of various operations (the equations define a congruence relation in a term algebra). The common goal of a specification, however, is to serve as a legal document that spells out a contract between a user and an implementer: The user may assume no more about an object than what is specified and the implementer must satisfy the specification through his implementation.

We view specification not merely as a legal contract but additionally as a means (1) to deduce properties of a specified object, (2) to deduce properties of other objects in which the specified object is a component (i.e., inheritance of properties) and (3) to implement the object by stepwise refinement of its specification. Therefore we require that a specification not merely be formal but also be in a form that admits of effective manipulation. This requirement rules out many specification schemes in which a program fragment, in some high level language, is used as a specification; typically such program fragments cannot be manipulated effectively.

In many specification schemes it is assumed that (1) each operation on an object is deterministic (i.e., applying the operation to a given state of the object results in a unique next state and/or unique values being returned), (2) an operation once started always terminates in every state of the object, and (3) operations are not applied concurrently. In many cases of interest arising in

*Partially supported by ONR Contracts N00014-86-K-0763 and N00014-87-K-0510, by a fellowship from The John Simon Guggenheim Foundation, and by a grant from The University of Texas URI-FRA.
applications such as operating systems, process control systems and concurrent databases, these assumptions are rarely met.

The purpose of this paper is to propose a specification scheme which allows effective manipulations of specifications and admits of nondeterministic, nonterminating and concurrent operations on objects. The method is illustrated by studying one example—a first-in-first-out queue, or a buffer—in detail.

A buffer object acts as an intermediary between a producer and a consumer, temporarily storing the data items output by the producer and later delivering them to the consumer. The buffer object is required to (1) deliver data in the same order to the consumer as they were received from the producer, (2) receive data from the producer provided its internal buffer spaces are nonfull, and (3) upon demand, send data to the consumer provided its internal buffer spaces are nonempty. Requests from the producer and the consumer may be processed concurrently: The producer is delayed until there is some space in the buffer and the consumer is delayed until there is some data item in the buffer. Observe that a request from the producer, to add a data item to the buffer may not terminate if the buffer remains full forever, and similarly, a request from the consumer may not terminate if the buffer remains empty forever.

The specification mechanism is based on “UNITY logic” as in Chandy and Misra [1]. We give a brief description of the notation and the appropriate concepts of the logic in the next section. Specifications in this notation have proved to be surprisingly succinct; for the buffer example, the specification consists of the three properties given above plus the description of the assumed protocol for data production and consumption. The inference rules of UNITY logic can be applied to deduce properties from specifications and to prove correctness of implementations. For instance, in Section 4 we show that concatenations of two buffers of sizes $M$ and $N$ result in a buffer of size $M + N$.

We give a brief introduction to UNITY in Section 2 including all the notations and logic to understand this paper. We describe a buffer program informally and then formally using UNITY logic, in Section 3. We demonstrate the usefulness of this specification in Section 4 by showing that the concatenation of two buffers of sizes $M$ and $N$ implements a buffer of size $M + N$. (The proof is given in some detail to emphasize that such proofs need not be excessively long or tedious, as is often the case with formal proofs.) A refinement of this specification, as a first step toward an implementation, is proposed in Section 5. The refined specification is used to implement a program in Section 6; its correctness is obvious (in fact, it follows almost mechanically) from the refinement suggested in Section 5. We close with a brief summary in Section 7. Some of the proofs are in Appendices A and B.