1988

An Introduction to the Specification Language Spec

Berzins, V.; Luqi

Naval Postgraduate School

V. Berzins and Luqi, "An Introduction to the Specification Language Spec", Technical Report NPS 52- 88-031, Computer Science Department, Naval Postgraduate School, 1988.
https://hdl.handle.net/10945/65283

T23

NPS52-88-031

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AN INTRODUCTION TO THE SPECIFICATION LANGUAGE SPEC

Valdis Berzins

Luqi

September 1988

Prepared for:

Navel Postgraduate School
Monterey, CA 93943

The work reported herein was supported in part by the National Science Foundation and the Naval Postgraduate School Research Foundation.

This report was prepared by:

[redacted]

LUQI
Assistant Professor
of Computer Science

Reviewed by:

[redacted]

ROBERT B. MCGHEE
Chairman
Department of Computer Science

Released by:

[redacted]

KNEALE T. MARSHALL
Dean of Information
and Policy Science

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-88-031	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL *(If applicable)* 52	7a. NAME OF MONITORING ORGANIZATION National Science Foundation
6c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA. 93943	7b. ADDRESS *(City, State, and ZIP Code)* Washington DC 20550
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Naval Postgraduate School 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER O&MN, direct funding
8c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA. 93943	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
AN INTRODUCTION TO THE SPECIFICATION LANGUAGE SPEC (U)

12. PERSONAL AUTHOR(S)
BERZINS, Valdis, LUQI

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Progress	FROM 87/10 TO 88/09	1988, Sept	35

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	specification, rapid prototyping, computer aided software engineering, real-time embedded system, ada, software design

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This paper presents a language for giving black-box specifications in the early stages of software design. This language is suitable for describing parallel programs, distributed systems, and real-time constraints. The underlying computational model combines temporal events with message passing to support descriptions of bothe active and reactive systems. The Features of the language, especially those important for large scale design, are presented by means of examples.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Luqi	22b. TELEPHONE *(Include Area Code)* (408)646-2735	22c. OFFICE SYMBOL 52Lq

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
☆ U.S. Government Printing Office: 1986—606-243
UNCLASSIFIED

An Introduction to the Specification Language Spec

Valdis Berzins
Luqi
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

This paper presents a language for giving black-box specifications in the early stages of software design. This language is suitable for describing parallel programs, distributed systems, and real-time constraints. The underlying computational model combines temporal events with message passing to support descriptions of both active and reactive systems. The features of the language, especially those important for large scale design, are presented by means of examples.

1. Introduction

The early stages of software development are concerned with building conceptual models of the problem domain and the proposed software system. These processes are critical for the development of large systems because the usefulness of such systems is limited by their conceptual complexity, and because errors in the early stages are substantially more expensive to correct if they are discovered at the end of the development effort rather than at the beginning. Conceptual complexity can be controlled via careful use of abstractions [3] while errors can be reduced by applying computer-aided design tools at the early stages [6]. Formal specifications are needed to apply these techniques effectively and to support rigorous reasoning about the consequences of a specification. Such a capability is essential for producing reliable software products with predictable properties.

Abstractions must have precise black-box descriptions to enable them to be used and understood independently from the details of any particular mechanism for realizing them. This allows users to be insulated from implementation details, and designers and maintainers considering a given level of the implementation to be insulated from details at lower levels. The black-box description of each abstraction must be explicitly recorded because the definer of the abstraction and its users must agree on its expected behavior to avoid errors. Informal descriptions are inadequate for this purpose because they are often ambiguous, requiring direct communication between the users and the definer when the need for clarifications is discovered. In large projects this can become a significant drain on the definer's time, and can lead to serious problems for the users of the abstraction after its definer has left the project. Such problems are most apparent in the maintenance phase.

Automated analysis and checking is needed for large systems because manual checking is too unreliable and requires too much time. Software tools are most effective for checking an aspect of a notation if that aspect has been explicitly and unambiguously defined [5]. Specifications with a completely defined syntax and semantics are needed to support computer-aided design tools capable of detecting an appreciable fraction of the errors occurring at the early stages development.

At the current state of the art different kinds of notations are needed for expressing specifications and programs. Programming languages such as Ada are not well suited for writing black-box specifications because they have been designed for describing the algorithms and data structures realizing a module rather than the behavior a module presents at its interface. Specification languages are also not well suited for describing programs because they have not been designed to specify internal data structures and algorithms, and current compilers are not capable of producing efficient implementations from unrestricted black-box specifications without considerable guidance from a programmer on choosing those internal details.

Formal specifications can be used in a computer-aided software engineering environment for quality assurance and synthesis. In addition to enabling early checks on the consistency of a proposed design, specifications are needed for describing the intended behavior of a software system for any quality assurance activity relating the actual behavior of a system to the intended behavior. This includes both testing and correctness proofs. Formal specifications have long been recognized as a necessary part of developing a proof of correctness. However, current practice emphasizes testing rather than correctness proofs in large scale software development because developing computer-checked proofs of correctness is expensive and requires specialized training and sophisticated software tools. Since large numbers of test cases are needed to achieve reliability for large systems, an important application of formal specifications is automatic classification of test results. A test oracle is a program that determines whether the results of running a test case represent an instance of correct behavior or a failure. Generating executable test oracles from formal specifications is much easier than generating efficient implementations. The availability of a test oracle enables randomly generated test cases to be executed and evaluated in large numbers without human intervention, allowing more thorough testing than manual classification of test results.

Spec is a formal specification language for software systems. The language is primarily designed for representing black-box specifications, and it has a subset suitable for describing domain models [6]. Domain models are developed in the initial stages of requirements analysis, and serve to define the types of objects in a problem domain along with their properties. Spec can be used together with other notations for recording goals and constraints in the requirements analysis phase. Black-box specifications are used in the functional specification stage of software development for defining external interfaces of a proposed system, and in the architectural design stage for defining internal interfaces of a proposed system. Spec can be be used together with any programming language for describing the internal structure of modules during detailed design and implementation. The use of Spec together with Ada for this purpose is described in [6].

A survey of previous work on formal specifications can be found in [8]. Much of the work on formal specifications has focused on the problem of proving the correctness of programs, and has addressed problems encountered in small scale programming. Spec has been intended primarily for supporting the use of abstractions in the design of software systems, and has been designed to apply to large scale systems. Spec has evolved from an earlier specification language [2] and a rapid prototyping language for the design of large real-time systems [12], guided by extensive classroom experience in using formal specifications in multi-person projects [3]. The most important advances over the earlier specification language are the integration of time into the underlying model, the incorporation of an inheritance mechanism, and the separation of granularity and control state considerations from the event-level interfaces of a module. The Spec language is suitable for specifying parallel, distributed, or time sensitive systems as well as conventional systems.

Spec differs from algebraic specification languages such as Larch [9] because it is based on models rather than theories. While it is feasible to write Spec axioms in the conditional equation form commonly used in algebraic approaches, the use of conceptual models and axioms of other forms can sometimes lead to simpler specifications. The restricted form of Larch is helpful for supporting automated tools for program verification, while the expressiveness of Spec is useful in developing large scale designs. Larch provides general purpose facilities for defining immutable data types along with a framework for adding an implementation-language dependent layer for defining state changes and concrete interfaces, motivated by the premise that interfaces involving state changes are inherently dependent on the implementation language. Spec is based on the premise that interfaces with state changes, exceptions, concurrent interactions, and time dependencies can all be specified independently of implementation language, and that the definition of a language dependent concrete interface is a matter of packaging rather than semantics. This reflects the difference between the prescriptive nature of specifications used as a design tool and the descriptive nature of specifications used primarily to prove properties about systems.

Model based approaches such as VDM [7] have a few similarities to Spec. However, Spec has been designed to handle systems with a wide range of features, e.g. concurrency and time dependent constraints, while VDM is primarily intended for specifying sequential systems [8].

Spec is based on the event model of computation, and uses predicate logic to define the desired behavior of modules independently of their internal structure. The most important features of this language are constructs for localizing information, controlling the sharing of specification concepts, supporting reuse of specification components, defining granularity of concurrent actions, and specifying timing constraints. Spec supports reuse of abstractions via inheritance and generic modules. Spec supports the specification of large systems via black-box descriptions, abstractions, import/export controls for defined concepts, and inheritance.

2. The Event Model

The Spec language uses the event model to define the black-box behavior of software modules. In the event model, computations are described in terms of modules, messages, events, and alarms. A module is an active black box that interacts with other modules only by sending and receiving messages. A message is a data packet that is sent from one module to another. An event occurs when a message is received by a module at a particular instant of time. An alarm defines an instant of time at a module.

Modules can be used to model external systems such as users and peripheral hardware devices, as well as software components. Modules have no visible internal structure. The behavior of a module is specified by describing its interface, which consists of the set of stimuli recognized by the module and the associated responses. A stimulus is an event, and the response is the set of events directly triggered by the stimulus. The events in the response consist of the arrivals of the messages sent out by the module because of the stimulus. State changes triggered by a stimulus are manifested in responses to future stimuli. The response of a module to a message is influenced only by the sequence and arrival times of the messages received by the module since it was created. This means there is no action at a distance: all interactions must involve explicit message transmissions. This restriction formalizes the requirement that each module must correspond to an independent abstraction, since it implies the behavior of a module can be influenced only via the operations provided by its interface.

Messages can be used to model user commands, system responses, and interactions between internal subsystems. Messages represent abstract interactions that can be realized in a wide variety of ways, including procedure call, return from a procedure, Ada rendezvous, coroutine invocation, external I/O, assignments to non-local variables, hardware interrupts, and exceptions. Each message has a condition, a name, a sequence of zero or more data values, and an origin. The condition has the value **normal** for messages representing normal interactions, and the value **exception** for messages representing abnormal interactions such as exceptions. The name of a message identifies the service requested by a normal message or the exception condition announced by an exception message. The data values represent either inputs or results, and may be present for any kind of message. The origin of a message is the event or alarm that caused the message to be sent. The origin records causal relationships in a computation history, and is used to identify destinations of reply messages in the Spec language.

Events are used to record and describe the behavior of a system. Each event consists of a module, a message, and a time, and is uniquely identified by these three properties. The time records the instant at which the module accepted the message. Events at the same module happen one at a time, and occur in a well-defined sequence. Events can be classified as **reactive** or **temporal**, depending on whether the origin of the message that arrived in the event is an event or an alarm. Reactive events represent responses to external stimuli, while temporal events represented actions initiated by the module based on the absolute time. Temporal events can be used to represent both regularly scheduled actions and actions initiated at unpredictable intervals by independent agents such as human users.

A reactive event in an airline reservation system is illustrated in Fig. 1. The event E1 is the stimulus causing the response event E2. E1 represents the arrival of a find_flights command from the travel agent at the airline reservation system. E2 represents the arrival of the message flights_3 at the travel agent module.

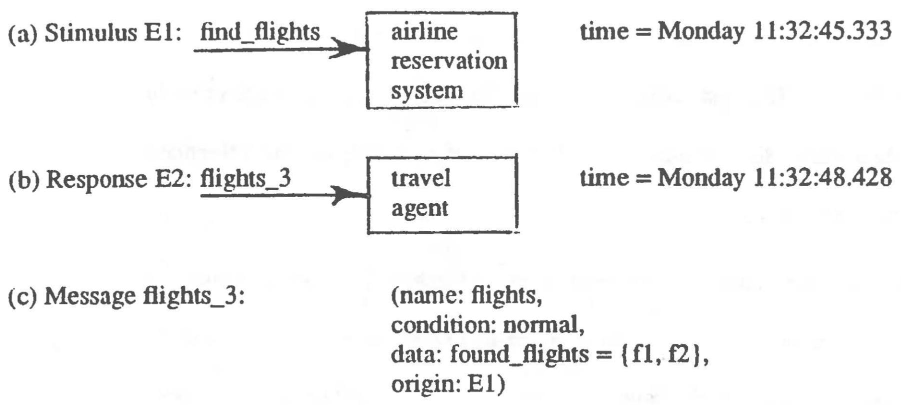
(a) Stimulus E1: find_flights
airline reservation system
time = Monday 11:32:45.333
(b) Response E2: flights_3
travel agent
time = Monday 11:32:48.428
(c) Message flights_3:
(name: flights,
condition: normal,
data: found_flights = {f1, f2},
origin: E1)

Fig. 1 A Reactive Event

This message contains the set of found flights, and is identified as a response to the command arriving in the event E1 via the origin attribute of the message. The set of events {E1, E2} represents a fragment of a computation history for the airline reservation system.

A temporal event is illustrated in Fig. 2. The alarm A1 defines the time at which the weekly run for generating paychecks is enabled at the payroll system. The temporal event E3 occurs when the Generate_Paychecks message is received by the payroll system, representing the instant when the process of generating paycheck actually starts. The scheduling delay between the alarm A1 and the event E3 can

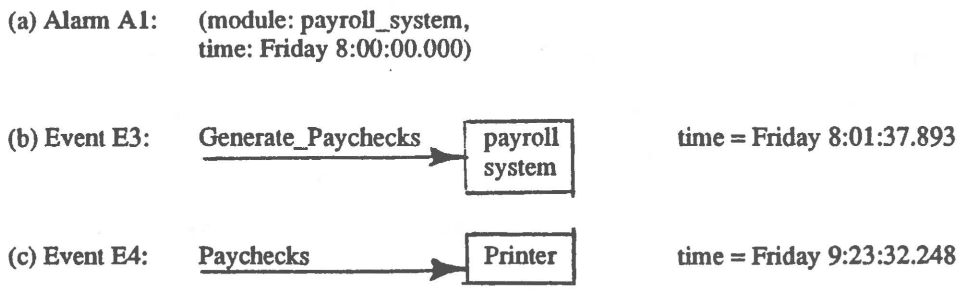
(a) Alarm A1:
(module: payroll_system,
time: Friday 8:00:00.000)
(b) Event E3: Generate_Paychecks
payroll system
time = Friday 8:01:37.893
(c) Event E4: Paychecks
Printer
time = Friday 9:23:32.248

Fig. 2 A Temporal Event

be constrained by the specification, and in the extreme case could be required to have zero length. The reactive event E4 occurs when the paychecks actually arrive at the printer. At this level of modeling the set of paychecks is treated as a single unit that arrives at an instant of time. In actuality printing is an extended process. The time required to print the checks is not distinguished from message transmission delay at this level of modeling, so that the arrival of the set of paychecks corresponds to the instant when the printing job is completed.

Alarms represent points in time when temporal events are triggered. Each alarm consists of a module, a message, and a time. An alarm causes the module to send the message to itself at the given time. A temporal event happens when the message arrives at the module, which can happen at or after the time the message was sent. Alarms serve as reference points for specifying constraints on scheduling delays for temporal events. Each module has a clock which measures local physical time. The model uses local physical time to support specifications of events that must happen at given absolute times (e.g. at 3am every Sunday). The time of an event or alarm is determined using the clock of the module at which the event occurs.

The event model is an extension of Hewitt's actor model, which is summarized and surveyed in [1]. The event model, like the actor model, is based on message passing and assumes a reliable buffered asynchronous communications system. All interactions between modules are explicit and are described in terms of a uniform communication mechanism. We have used buffered asynchronous communication rather than the unbuffered synchronous communication of Hoare's communicating sequential process model [10] because synchronous communication is difficult to implement in distributed systems, and we did not want to require the overhead of synchronized communication in cases where it is not needed. Synchronized communication can be readily expressed in terms of asynchronous communication using acknowledge messages in cases where it is semantically necessary. Another difficulty with unbuffered synchronous communication is that recursive communications patterns necessarily lead to deadlocks [1].

Both the event model and the actor model are designed for describing concurrent and distributed processes, with sequential or centralized processes are special cases. The event model extends the actor model by introducing temporal events, a quantitative treatment of time, and atomic multi-event transactions. These features are important for describing real-time systems and distributed systems with multiple communications protocols that must be protected from interference. Spec notation differs from the scripts used in actor programming languages because it describes behavior using preconditions and postconditions rather than algorithms for producing output messages.

3. The Relation between Spec and the Event Model

The basic building blocks in Spec are modules and messages. The Spec language assumes message transmission is reliable, which means every message sent eventually arrives at its destination. Constraints on transmission delays can be specified explicitly, and messages without such constraints can have arbitrarily long and unpredictable transmission delays.

The event model and the Spec language admit nondeterminism due to partially specified communication delays or partially specified responses. Complete specifications admit only deterministic behavior. In Spec it is possible to specify that a response must be deterministic (repeatable) without completely specifying the other properties of the response.

Each module has the potential of acting independently, so that there is natural concurrency in a system consisting of many modules. Modules can be used to model concurrent and distributed systems, as well as systems consisting of a single sequential process. The event model helps to expose the parallelism inherent in a problem, since a stimulus can have a set of unordered responses occurring at different locations. Since events happen instantaneously and the response of a module is not sensitive to anything but the sequence of events at the module, the event model implies concurrent interactions cannot interfere with each other at the level of individual events. Atomic transactions can be used to specify constraints on the order in which a module can accept events. This capability is useful for defining systems with modes in which only subsets of the system commands are available, and for specifying synchronization constraints involving chains of events in distributed systems. Atomic transactions must be used with care, because they can interact with each other or with timing constraints to produce unsatisfiable specifications. Deadlocks are a well known example of such situations.

The rest of this section briefly describes how the semantics of Spec can be defined in terms of the event model via computation histories. A computation history consists of a set of alarms and a set of events. A specification determines the set of legal computation histories for a system. The set of legal computation histories is determined via a set generative constraints and a set of restrictive constraints derived from the specification. A generative constraint says every legal computation history must contain events or alarms with given properties, while a restrictive constraint says every event or alarm in a legal computation history must satisfy a given property.

The generative constraints of a specification are derived from the event definitions in the specification. Every event definition determines a set of pairs containing a precondition and a postcondition. For each event E satisfying a precondition of a reactive event there must be an event in the history which satisfies the corresponding postcondition and whose origin is E. For each alarm A satisfying a precondition of a temporal event there must be an event in the history which satisfies the corresponding postcondition and whose origin is A. The origin of an event is the same as the origin of the message that arrived in the event. An alarm must occur at a module for each time that satisfies the precondition of a temporal event at the module.

The restrictive constraints of a specification are derived from the definitions of the atomic transactions associated with each module. An atomic transaction restricts the order of events at a module. There are also restrictive constraints to ensure all of the events at a module occur at distinct times and every event has an origin that precedes the event and corresponds to an event specification. The detailed derivations of the generative and restrictive constraints for Spec are beyond the scope of this paper.

4. Specifying Software using Spec

The Spec language provides a means for specifying the behavior of three different types of modules: functions, machines, and types. There are also three different types of messages distinguished in Spec: normal messages, exceptions, and generators. These types of modules and messages form a simple set of primitives sufficient to describe all common varieties of software components. The properties of these kinds of modules and messages are described below, with examples of each.

It is useful to classify modules as mutable or immutable because immutable modules are easier to analyze and are subject to fewer restrictions when used in an implementation. A module is **mutable** if the response of the module to at least one message can be affected by previous messages it has received, and is **immutable** otherwise. Mutable modules behave as if they had internal states or memory, while the behavior of immutable modules is independent of the past. Immutable modules can be shared by the implementations of two separate processes without any risk of interference, and can be replicated without changing their semantics, while mutable modules cannot be. The distinction between mutable and immutable modules is a property of the behavior of a module rather than a property of its internal structure. It is possible to implement immutable modules using mutable components if the components are properly protected against unintended interactions.

In Spec, all functions are immutable modules. Machines are intended to be mutable, although Spec does prevent the specification of trivial machines which are immutable because they have only a single state. Types can be either mutable or immutable in Spec, and both kinds are useful in practice.

4.1. Functions

The response of a function module is influenced only by the most recent stimulus, so that function modules do not exhibit internal memory. Completely specified function modules calculate single-valued functions in the mathematical sense, while incompletely specified function modules can exhibit nondeterministic behavior. An example of a specification for a square_root function is shown below.

```
FUNCTION square_root {precision: real} WHERE precision > 0.0

 MESSAGE(x: real)
   WHEN x >= 0.0  -- label: positive
    REPLY(y: real)
    WHERE y >= 0.0 & approximates(y * y, x)
   OTHERWISE REPLY EXCEPTION imaginary_square_root
    -- label: negative

 CONCEPT approximates(r1 r2: real)
    -- True if r1 is a sufficiently accurate approximation of r2.
    -- The precision is relative rather than absolute.
   VALUE(b: boolean)
    WHERE b <=> abs((r1 - r2) / r2) <= precision
END
```

The basic unit in a Spec description specifies required responses to a stimulus. The keyword MESSAGE introduces the description of a stimulus recognized by a module, which consists of an incoming message. A message can have a name and zero or more formal arguments representing input values. Message names are used to distinguish different types of stimuli, corresponding to requests for different services. Most function modules provide a single service, and are usually designed to accept anonymous messages, i.e. messages whose name is the null string. The square_root function accepts anonymous messages containing a single real number denoted by the formal argument x. The Spec language requires the types of all data values to be declared to allow type consistency checks. This does not impose any restrictions on the designer because Spec has union types, and types can have subtypes. There is a universal type called *any* which is the union of all other types and can be used to write untyped specifications and express general laws. A mature specification environment for supporting the use of the Spec language is expected to have type inferencing capabilities for automatically filling in and maintaining type declarations in the cases where they can be determined from the context.

The response of a module to a message can be defined with several cases introduced by WHEN clauses. The example illustrates such a case analysis with two cases, one corresponding to a normal response and the other to an exception. The predicate after each WHEN is a precondition, describing the conditions under which the associated response must be triggered by an incoming message with a given name and condition. The preconditions in each WHEN statement are stated independently, so that the order of the WHEN statements does not matter.

OTHERWISE is an abbreviation for the case where none of the other WHEN statements apply. In the example the OTHERWISE means the same thing as WHEN $x < 0.0$. In the Spec language each series of WHEN statements must be terminated by an OTHERWISE, to make sure all cases are covered. If a case is to be left undefined, the designer must say so explicitly.

A REPLY describes the message sent back in response to a stimulus. The reply is sent to the module originating the message acting as the stimulus, which can be determined from the implicit origin attribute of the message. A stimulus can have only a single REPLY, corresponding to the call/return interface convention followed by most subprograms. A REPLY can have zero or more data components associated with it, representing output data values that are all delivered at the same time. In the example the reply for the normal case has no name and a single data component, while the reply for the exceptional case has a name but no data components. If REPLY is followed by EXCEPTION then the condition of the reply message is **exception**, representing an exceptional event, and otherwise the condition of the reply message is **normal**, representing a normal response. EXCEPTION can also appear after MESSAGE in the specification of an exception handler, indicating that the stimulus must be an exception condition.

An outgoing message such as a REPLY can have a WHERE clause, which describes a postcondition that must be satisfied by the outgoing message. The WHERE keyword is followed by a statement in predicate logic describing the required relation between the contents of the message that was received and the contents of the reply message. This predicate states how to recognize a correct result, but it does not specify how to compute the required output. In the example the normal reply must contain a positive value whose square is approximately equal to the input. This provides sufficient information to distinguish correct outputs from incorrect ones, but does not give any hint about how to implement the required function. This is desired when specifying black-box behavior. In later stages of design the black-box specification can be augmented with annotations containing implementation advice, such as the name of an algorithm for realizing the module.

The behavior of a module can be summarized in a stimulus-response diagram for review purposes. Such a diagram for the square_root module is shown in Fig. 3. The diagram shows the incoming and outgoing messages for each case of the response. The responses have been labeled with mnemonic names derived from the comments. Normal messages are shown using solid arrows while exception messages are shown using dotted arrows. Responses involving state changes are shown as squares, while responses without state changes are shown as circles. In this case there are no state changes, so all nodes are round.

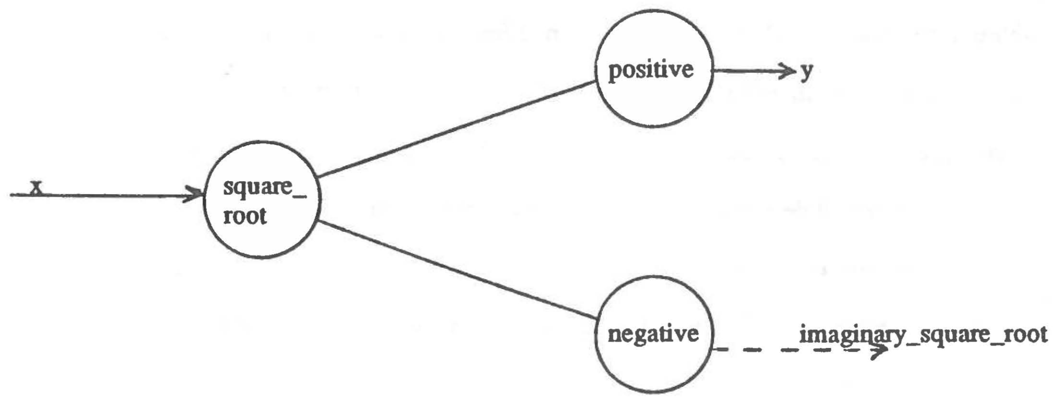

Fig. 3 Stimulus-Response Diagram for Square_Root

A CONCEPT in Spec introduces a new predicate symbol, a new function symbol, a new constant symbol, or a new type symbol, and defines the intended properties of the new symbol. Concepts represent abstractions which are needed to explain the behavior of a system but do not represent parts of the system being specified. In the example the concept *approximates* defines the intended meaning of "sufficiently accurate approximation" in terms of the generic parameter *precision*. Some notion of approximation is needed to specify a practical square root function because it is not possible to implement exact square roots using machine arithmetic. In this case the size of the acceptable interval is defined relative to the size of the input value rather than as an absolute constant. The generic parameter allows a single definition for a square root module to be adapted to many applications with different precision requirements. Introducing an explicitly defined concept modularizes the specification. This helps simplify the postcondition and supports stepwise refinement and localization of information. The definition of a concept can be delayed or left as an informal comment when the concept is identified and the postcondition is developed.

4.2. Machines

A machine is a module with an internal state, i.e. machines are mutable modules. An example of a machine representing a simplified inventory control system for a warehouse is shown below.

```
MACHINE inventory
  -- assumes that shipping and supplier are other modules.
 STATE(stock: map{item, integer})
 INVARIANT ALL(i: item :: stock[i] >= 0)
 INITIALLY ALL(i: item :: stock[i] = 0)

 MESSAGE receive(i: item, q: integer)
   -- Process a shipment from a supplier.
  WHEN q > 0
   TRANSITION stock = bind(i, *stock[i] + q, stock)
     -- Delayed responses to backorders are not shown here.
  OTHERWISE REPLY EXCEPTION empty_shipment

 MESSAGE order(io: item, qo: integer)
   -- Process an order from a customer.
  WHEN 0 < qo <= stock[io]
   SEND ship(is: item, qs: integer) TO shipping
    WHERE is = io, qs = qo
   TRANSITION *stock = bind(i, stock[i] + q, stock)
  WHEN 0 < qo > stock[io]
   SEND ship(is: item, qs: integer) TO shipping
    WHERE is = io, qs = stock[io]
   SEND back_order(ib: item, qb: integer) TO supplier
    WHERE ib = io, qb + qs = qo
   TRANSITION stock = bind(io, 0, *stock)
```

```
    OTHERWISE REPLY EXCEPTION empty_order
  END
```

A data flow diagram showing the context for this system is shown in Fig. 4. The behavior of a machine is described in terms of a conceptual model of its state, which serves to summarize the aspects of previous messages received by the machine that can influence its future behavior. States are localized: the state of a machine can change only at an event in which the machine receives a message. Conceptual models are described in terms of a finite set of state variables, whose types are declared after the keyword STATE. In the example there is just one state variable, *stock*, whose value is a map from items to integers. Map is a generic pre-defined type in Spec. A map is a function with a finite range which can have an unlimited number of elements. The formal parameters *from* and *to* represent the types of the elements in the domain and range of the map, respectively. The notation stock[i] is a shorthand for the map operation fetch(stock, i) which denotes the value of the map *stock* at the domain element i. In the example stock[i] represents the quantity of the item i on hand in the current state of the inventory. The map overwrite operation *bind* produces a new map that differs from the old one at a single point, as described by the equation

$$\text{bind}(x, y, m)[z] = \text{if } z = x \text{ then } y \text{ else } m[z].$$

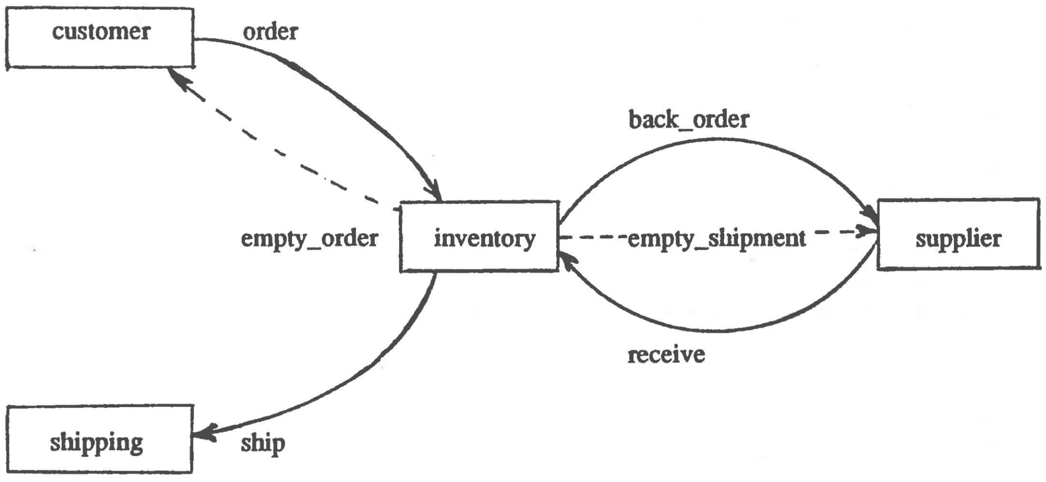

Fig. 4 Data Flow Diagram for Inventory

The description of a conceptual model includes invariants and initialization constraints. Invariants are restrictions on the set of meaningful states which appear after the keyword INVARIANT. Initialization constraints are restrictions on the initial state which appear after the keyword INITIALLY. Invariants must be satisfied in all reachable states, while initialization restrictions must be satisfied only in the first state. In the example the invariant says that the quantity on hand must be non-negative for every item at all times, and the initialization constraint says there are no items in stock at the beginning.

State changes are described by predicates after the keyword TRANSITION. In such statements, a state variable x refers to the value of the x component of the conceptual model for the current state (just after the arrival of the stimulus), while the expression *x refers to the value of x in the previous state (just before the arrival of the stimulus). The transitions in the example are equations rather than assignment statements. Equations can describe the transition either forwards or backwards in time, whichever is simpler (cf. the first two transitions in the example). The first transition in the example increases the amount of the item i on hand to reflect the arrival of an incoming shipment, while the second transition decreases the amount of the item i on hand to reflect the departure of an outgoing shipment. The *x notation can only be used in TRANSITIONS and in WHERE clauses describing the output in terms of the new state. The Spec language follows the convention that state variables of a machine or instance variables of an abstract data type do not change unless the variable is explicitly mentioned in a TRANSITION clause.

Messages sent to destinations other than the origin of the incoming message are described using SEND instead of REPLY. A SEND statement means that a message satisfying the description must be sent to the specified module. SEND statements are useful for describing distributed systems with a pipeline structure. There can be more than one SEND statement in the description of a response to cover the case where several messages must be sent to different destinations. In such cases the outgoing messages can be sent out concurrently or one at a time in any order, without waiting for any responses. The inventory example shows such a multiple response for the *order* message, in the case where there are not enough items on hand to fill the order completely. In this case there are two messages in the response, one of which goes to the module *shipping*, representing the shipping department, and the other of which goes to the module *supplier*, representing the supplier for the items in the warehouse. The first message represents a request to send out a partial shipment, while the second message represents a backorder for the items in the unfilled part of the order.

4.3. Types

A type module defines an abstract data type. An abstract data type consists of a value set and a set of primitive operations involving the value set. The elements of the value set are called the instances of the type. In the event model, a type module manages the value set of an abstract data type, creating all of the values of the type and performing all of the primitive operations on those values. Each message accepted by the type module corresponds to one of the operations of the abstract data type. The messages of a type module usually have names, since abstract data types usually provide more than one operation.

Recall that modules are mutable if and only if they have internal states. Both immutable and mutable types are useful in practice, and both can be specified using Spec. The value set of an immutable type is fixed, and the properties of the individual instances of the type cannot be changed. A mutable type can have operations that modify the value set or change the properties of existing instances. In particular, mutable types can have operations which create new instances or modify existing instances.

The difference between mutable and immutable types becomes most apparent when an instance of a type is shared by several program variables. Mutating operations affect all of the variables denoting the modified instance of the type. Mutable types must be used with care because shared instances of mutable types can introduce hidden interactions between modules. All of the components of the conceptual representation in the specification of a machine or type should be instances of immutable types to ensure that only independent abstractions are specified.

Each instance of a mutable type has a permanent identity, which remains fixed despite arbitrary changes to the properties of the instance. A mutating operation without a REPLY changes the properties of an instance and does not affect the identity of the instance bound to any program variable. In contrast, an assignment to a program variable affects the identity of the instance bound to the variable without affecting the properties of either the instance bound to the variable in the old state or the instance bound to the variable in the new state. The choice between a function with a returned value and a procedure with an output variable for realizing an operation with an output value is a matter of packaging which has no effect on whether the operation can mutate instances of a data type, and subprograms with output variables can be used to implement operations of both mutable and immutable types.

An example of a specification for an immutable abstract data type is shown below.

```
TYPE rational
 INHERIT equality{rational}
 MODEL(num den: integer)
 INVARIANT ALL(r: rational :: r.den ~= 0)

 MESSAGE ratio(num den: integer)
  WHEN den ~= 0
   REPLY(r: rational)
   WHERE r.num = num, r.den = den
  OTHERWISE REPLY EXCEPTION zero_denominator

 MESSAGE add(x y: rational) OPERATOR +
  REPLY(r: rational)
  WHERE r.num = x.num * y.den + y.num * x.den,
   r.den = x.den * y.den

 MESSAGE multiply(x y: rational) OPERATOR *
  REPLY(r: rational)
  WHERE r.num = x.num * y.num, r.den = x.den * y.den

 MESSAGE equal(x y: rational) OPERATOR =
  REPLY(b: boolean)
  WHERE b <=> (x.num * y.den = y.num * x.den)
END
```

Data types have conceptual models, which are used to visualize and describe the instances of the type. The conceptual model is used to specify the behavior of a type, and forms the mental picture of the type for the programmers who use the operations of the type. The conceptual model is chosen for clarity, and is usually different than the data structure used in the implementation. In case the data type must be re-implemented to improve performance, the data structure used in the implementation will change, but the conceptual model will not. The conceptual model consists of a finite set of components called instance variables. The types of the instance variables are declared after the keyword MODEL. In the example there are two instance variables, *num* and *den*, corresponding to the numerator and denominator of a fraction representing a rational number.

Each instance of the type can be represented as a tuple containing the values of the instance variables. Restrictions on the components of the model are described in the INVARIANT. The INVARIANT is a predicate that must be true for all meaningful conceptual representations.

In the example we are using the standard mathematical model for rational numbers. The invariant must exclude pairs with zero denominators, because the interpretation of the pairs as ratios does not make sense in that case. The infix operator "~=" represents the standard "not_equal" operation associated with the integer type, which is specified in the pre-defined type library associated with the Spec language [6]. It is not necessary for there to be a 1 : 1 correspondence between conceptual representations and values of the abstract data type, although in cases without such a correspondence the model is not fully abstract and some extra care must be taken in defining the operations to avoid unintended nondeterminism. Our example does not have unique conceptual representations, because the pairs (1, 2), (2, 4), and (-1, -2) all represent the same rational number. This lack of uniqueness is reflected in the equal operation, where equality on rationals is defined in terms of equality on integers. It is incorrect to say that two rationals are equal if and only if corresponding components are equal unless the invariant is strong enough to give unique conceptual representations. Since the standard interpretation of equality is a single-valued predicate, and hence deterministic, we must ensure the operation is defined to give the same result for all valid conceptual representations of any fixed pair of rational numbers. Some additional restrictions that would make make the conceptual representation unique in the example are that the denominator must be strictly positive and that the fractions must be reduced to lowest terms.

The invariant on the conceptual representation should be adjusted to make the descriptions of the operations as simple as possible. The invariant on the conceptual representation does not involve the implementation data structure and does not restrict the designer's choice of implementations. The invariants on the implementation data structures are often much more complicated than the conceptual invariants, because implementation invariants often determine efficiency. Most knowledge about data structures is really about the art of choosing implementation invariants that enable efficient algorithms.

Inside the module defining an abstract data type, predicates describing the effects of the operations can be written in terms of the conceptual representation. Inside the module defining an abstract type instances of the type can be described as if they were tuples containing the components specified in the MODEL. The notation x.y can be used to refer to the y component of the conceptual representation for the abstract data value x. The specifications of other modules may describe the instances of abstract types only in terms of the MESSAGEs it provides and the CONCEPTs it EXPORTs.

It is sometimes convenient to express complicated conditions as lists of independent constraints. The predicates after INVARIANT, WHEN, and WHERE can be lists of expressions separated by commas. A list of statements is true if and only if all of the statements in the list are true individually, so that in this context a comma means the same thing as &. The comma has a lower precedence than all of the other operators, so that it can be used to separate statements at the top level without need for parentheses.

The INHERIT keyword introduces a list of modules to be inherited. In the example the standard properties of the equality operator, such as reflexivity, transitivity, and symmetry, are inherited from the predefined Spec module *equality{t}*, along with a *not_equal* operation with the standard relationship to the *equal* operation. The inherited definitions are combined with the explicitly given ones. If an operation with a given name is both inherited and explicitly defined, then the constraints introduced by both definitions must be satisfied simultaneously. The semantics of inheritance in Spec is described in more detail in [4]. Inheritance is used to avoid repeating standard definitions, and is useful for ensuring consistent treatment of standard concepts such as equality across a large number of components. Such a facility can be important for specifying uniformity constraints for the interfaces in different subsystems of very large software systems.

The OPERATOR keyword introduces special infix notations for user-defined operations as a convenience to the designer. These infix operators must be chosen from a fixed set of operator symbols with pre-defined operator precedences. In the example the standard symbols for addition (+), multiplication (*), and equality (=) are introduced in this fashion. These symbols can only be associated with binary operations and are interpreted according to the OPERATOR declarations only when they appear as binary infix operators. The * symbol always refers to values in the previous state when it appears as a unary prefix operator (e.g. in Spec TRANSITION clauses).

Spec provides facilities for specifying mutable types because they are used for efficiency reasons in internal interfaces of many systems. We recommend avoiding mutable types in user interfaces. An example of a definition for a mutable type is shown below.

```
TYPE queue{t: type}
 INHERIT mutable{queue{t}}
  -- Inherit definitions of the concepts "new" and "defined".
 MODEL(e: sequence{t})
  -- The front of the queue is at the right end.
  INVARIANT true
    -- Any sequence is a valid model for a queue.

  MESSAGE create
     -- A newly created empty queue.
    REPLY(q: queue{t}) WHERE q.e = [ ]
    TRANSITION new(q)

  MESSAGE enqueue(x: t, q: queue{t})
     -- Add x to the back of the queue.
    TRANSITION q.e = append([x], *q.e)

  MESSAGE dequeue(q: queue{t})
     -- Remove and return the front element of the queue.
    WHEN not_empty(q)
      REPLY(x: t)
      TRANSITION *q.e = append(q.e, [x])
    OTHERWISE REPLY EXCEPTION queue_underflow

  MESSAGE not_empty(q: queue{t})
     -- True if q is not empty.
    REPLY(b: boolean) WHERE b <=> (q.e ~= [ ])
 END
```

In mutable types the instances of the type have internal states, and operations are provided for changing the internal states of the instances. TRANSITION clauses are allowed in types as well as machines. A type is mutable if and only if it has a non-trivial TRANSITION clause (i.e. a TRANSITION that implies *x ~= x for some component x). Mutating operations, such as *enqueue* in the example above, are described using TRANSITION clauses.

Object identity is an important issue for mutable types because all of the program variables bound to the same mutable object will be affected if a state changing operation is applied to the object. In the example the *create* operation is specified to return a newly created instance of the type queue{t} via the predicate *new*. A newly created object is guaranteed to be distinct from all objects defined in the previous state. The concept *new* is not part of the Spec language, but it is provided by the predefined generic module *mutable* whose instances can be inherited by any mutable type. This is illustrated in the example, which inherits the module mutable{queue{t}}. The definition of this standard generic module is shown below.

```
 DEFINITION mutable{t: type}

  CONCEPT new(x: t)
    VALUE(b: boolean)
    WHERE b <=> x IN t & ~(x IN *t),
       -- An object is new if it belongs to the type in the current state
       -- and it did not belong to the type in the previous state.
      ALL(a c: t :: new(a) & c IN *t => id(a) ~= id(c))
      -- A new object is distinct from any object existing in the previous state.

   CONCEPT id(x: t)
    VALUE(n: nat)
    WHERE ALL(y z: t :: id(y) = id(z) => y = z),
     ALL(y: t :: id(y) = id(*y))
      -- Every object has a permanent unique identifier.
  END
```

This is an example of a definition module. Definition modules can contain only concept definitions, and are used for providing convenient access to widely shared concepts. The effect of inheriting a definition module is the same as importing all of the concepts defined in that module.

4.4. Generators

A generator is a message that generates a sequence of values one at a time. An example of a specification for a generator is shown below.

```
  FUNCTION primes
   IMPORT prime FROM nat
   IMPORT sorted FROM sequence{nat}

   MESSAGE(limit: nat)
    GENERATE(s: sequence{nat})
    WHERE increasing_order(s),
     ALL(i: nat :: i IN s <=> 1 <= i <= limit & prime(i))

   CONCEPT increasing_order(s: sequence{nat})
    VALUE(b: boolean)
    WHERE b <=> sorted{less_or_equal@nat}(s)
  END
```

The "@" is used in Spec to determine the type of an overloaded operator or constant in places where it is not clear from the context. The GENERATE keyword means the same thing as a REPLY except that the result is a sequence whose elements are delivered one at a time rather than all at once. This means that the elements will be generated one at a time, and processed incrementally, rather than being generated all at once and returned in a single data structure containing all of the elements, as would be the case for a REPLY of type sequence. In a program a generator is used to control a data driven loop. Generators can also be used in specifications of other modules, for example to define the range of a quantified variable. Generators are interpreted as sequence-valued functions when they appear in specifications. The distinction between GENERATE and REPLY corresponds to the choice of whether to represent a sequence as a time series or as a data structure.

Any message with a GENERATE is a generator, so that generators can be defined as operations of an abstract data type or a machine. Generators are important in this context because they can be used to provide an efficient way to scan all of the elements of an abstract collection without exposing the data structure used to implement the collection. Generators are used most often in the internal interfaces of software systems.

5. Features for Specifying Large Systems

This section discusses some features of the Spec language that make it appropriate for specifying large systems. The most important requirements for supporting large scale design are the ability to localize information, the ability to isolate the details relevant for a single purpose, and the ability to mechanically detect interactions between different parts of the system. Spec addresses these issues via modules, concepts, atomic transactions, and inheritance. The underlying event model is designed to make interactions between modules explicit and easy to describe. The application of Spec concepts, atomic transactions, and inheritance to the design of large systems is discussed in this section. An example illustrating the development of a complete system using Spec and a more detailed description of the language can be found in [6].

5.1. Concepts

A facility for introducing named concepts with explicit definitions and interfaces is important for organizing and simplifying descriptions of complex software systems. It is not a good idea to express a complicated constraint as a single very long expression in predicate logic, just as it is not a good idea to implement a large system as a single monolithic program: the result is too difficult for people to understand. Concepts have the same purpose in a specification language that subprograms do in a programming language, namely to provide a mechanism for orderly decomposition.

Concepts in the Spec language correspond to abstractions needed to explain or describe the behavior of a system, and are distinguished from system components that realize that behavior. Concepts are important for structuring complex formal specifications into bite-sized parts that can be understood without undue effort. Specifications in predicate logic can be relatively easy to follow if they are expressed using primitives at a level of abstraction close to that of system being specified. Spec concepts can be used to write formal specifications whose structure matches that of the informal explanations given by people.

Concepts are building blocks for explaining and understanding a system or problem domain. The process of analyzing and designing large systems for particular problem domains can be simplified by building up libraries of concepts relevant to the domain. A notation such as Spec is needed to allow such a library to be constructed. The existence of such a library would raise the level at which a designer can work, by matching the primitives available to the problem domain. This provides a means for tailoring a standard language to many different application areas.

Concepts can also be used to mix formal and informal specifications. by providing a formal definition of a precondition, postcondition, invariant, or transition in terms of some concepts, writing definition skeletons for the concepts without formal definition bodies, and giving informal descriptions for the concepts as comments. The formal definitions of the concepts can be filled in later, when the design has stabilized, or can be left out entirely if the details are not critical. The ability to mix formal and informal specifications in a disciplined manner can be important in practical projects with tight schedules. The most important place for formality and precision is in the most difficult or complicated parts of the system, because these are most likely to be misunderstood. Informal descriptions are often adequate for the well understood parts of the systems.

Concepts are also important for explaining and testing the behavior of modules, and should be reflected in reference manuals and test oracles as well as in the formal specifications. Concepts represent properties of the software that are needed to describe or test the intended behavior of the software system. Concepts are delivered to the customer in the manuals explaining how the system is supposed to operate, where they may be explained less formally than in the functional specifications and architectural design. Concepts do not normally represent components of the code to be delivered, although it may be useful to implement them for testing purposes.

Concepts are immutable, and cannot introduce any run-time interactions between different parts of the system. Shared concepts can introduce conceptual dependencies between different parts of a large software system. Such dependencies can become important when the system evolves, and some of the system concepts are redefined. It is important to record and trace such dependencies to aid in estimating the impact of a proposed change and to identify the parts of the system that must be redesigned or reimplemented.

In Spec every concept is attached to some module, and is local to that module unless it is exported or inherited. Only concepts can be exported. If a concept is exported, then it can be explicitly imported by other modules and used in their definitions. The export/import mechanism is used to record logical dependencies between modules, so that mechanical aid can be provided for tracing the impact of a proposed change to a definition. This is very important for analyzing and maintaining large systems.

Making concepts local by default avoids the need for maintaining globally unique names for concepts, and simplifies the designer's view of the system by limiting the names visible in a specification module to just those that are needed. Requiring non-local concepts to be imported explicitly avoids ambiguity and eliminates the possibility of surprises caused by implicit scoping rules. The mechanism is intended to be used in the context of a computer-aided specification system with tools for displaying definitions of concepts without regard for whether they are local or imported, for retrieving library concepts, for inserting import links by pointing to intended choices, and for locating all modules affected by a change to a given concept.

5.2. Atomic Transactions

An atomic transaction specifies constraints on the order in which a module will accept messages. Spec allows the specifications of such constraints to be separated from the specifications of the responses to individual messages. This simplifies the design of distributed systems by separating granularity considerations from local views of module behavior. Some granularity considerations are mutual exclusion and waiting for expected events. For example, modules with interactive transactions involving multiple messages may need to keep transactions involving different users from overlapping to prevent interference. Another example is a protected bounded buffer data type, which delays read operations as long as the buffer remains empty and delays write operations as long as the buffer remains full.

Atomic transactions can also be used to specify the behavior of complex systems with modes, where different subsets of the system commands are available in each mode. The separation of granularity constraints from behavior makes it easier to ensure that the semantics of each command is the same in all modes where the command is available. We believe such a restriction is necessary to enable people to use complex systems with multiple modes.

Atomic transactions can be defined in Spec using conditional guards, alternative choices, sequencing, repetition, and recursion. The facilities are described in detail and illustrated with examples in [6].

5.3. Views and Inheritance

Inheritance is useful in designing large systems. The Spec language has an inheritance mechanism which can support re-using common specification fragments, specifying constraints common to the interfaces of many modules, and providing incremental views of a system. The first two uses of inheritance were illustrated in section 4.3 in the context of sharing the definition of the concept of a newly created data value and in providing a standard interpretation of equality operations for many different data types. Standard interpretations for messages that can be inherited by many different modules provide a means of achieving uniformity in large systems, which is important for making them easier to use, understand, and design.

Inheritance also provides a mechanism for combining incremental views of a system. This allows the definition of a system to be organized in many smaller pieces that provide just the information needed for a given purpose. For example, the interface of a system to each class of users can be a separate view of the system, and each view can be specified as a distinct piece of Spec text. This makes it easier to partition the work of specifying a large system, because the views corresponding to different interfaces can be developed by different designers or different teams. A total picture of the system is formed by expanding the definition of a module that inherits all of the individual views. Such an expansion can be done mechanically, and the resulting combined specification can be subjected to global consistency checks to provide early indications of coordination problems in team design efforts.

Inheritance can also be used to support design by stepwise refinement, by making the structure of the specification correspond to the structure of the refinements. This is especially useful for separating the user view of a system from the implementor's view. The implementor's view can be a separate specification unit that inherits the information in the user's view. Keeping the two views in two clearly identifiable text units makes it easier to keep track of which details are visible to the users and which are not. This makes it easier to write user manuals, because the information that should be reflected by the manual is easier to identify. Such an application of the inheritance mechanism is illustrated in [6]. The same idea can be used to represent the specifications for a series of releases of a system in a way that clearly reflects the changes from one release to the next.

The Spec inheritance mechanism and the rules for combining different versions of messages and concepts inherited from multiple parents are described in more detail in [4].

6. Time

Time appears in Spec in two contexts: temporal events and timing constraints. Temporal events in software systems usually represent regularly scheduled activities, such as generating paychecks, performing monitoring and control functions in embedded systems, or compacting disk space. The two contexts are related because temporal events are often subject to real-time deadlines, as in the first two of the three examples mentioned above. Triggering conditions for temporal events are defined in Spec via predicates involving the current absolute time, while the associated responses are specified in the same way as responses to reactive events. Examples of specifications for temporal events can be found in [6].

Timing constraints are specified as predicates involving the delay and the period associated with a message. The delay is the length of the time interval between the origin event or alarm associated with a message and the event where the message arrives. The period associated with a message is the length of the time interval between the event where the message arrives and the previous event where a message of the same type arrived at the same module, or to the beginning of the computation history if there was no such previous event. Restrictions on the delay constrain the performance of the system, and usually appear in the postcondition of a message, while restrictions on the period constrain the behavior of the system's environment, and usually appear in the precondition of a message. Timing constraints can be associated with atomic transactions as well as with individual messages.

The current local time of a module refers to physical time at the current location of the module. Some care is required in comparing times at different locations because the local clocks of different modules cannot be precisely synchronized with each other. This is most clearly apparent for distributed systems that span multiple time zones. Another consideration stems from the special theory of relativity, which states that the difference in the readings of two clocks at different locations depends on the motion of the observer. This implies a set of clocks cannot be precisely synchronized unless all of the clocks have fixed positions relative to each other, and even in that case they will appear to be synchronized only to observers that are not moving with respect to the clocks. However, the effects of motion are negligibly small in most practical situations.

One consequences of this lack of synchronization are that the order of occurrence of two events cannot be determined by simply comparing the local times of the events, This problem can be solved in principle by applying a transformation to each of the local clocks to convert them to readings from a clock in a standard location (e.g. Greenwich mean time). If the effects of motion on time can be ignored, then the resulting ordering is guaranteed to be consistent with the orderings observed by any physical means outside the software system.

Another consequence of the lack of synchronization of the clocks in different modules is that message delays cannot be calculated by taking the difference between their times of occurrence. A transformation to convert times to readings from a standard clock will also solve this problem in principle. However, any practical scheme for reading remote clocks or synchronizing clocks at different locations involves messages with finite delays that are not completely predictable. While message delays can be specified in theory, such delays can be measured only approximately unless the origin and destination of the message are at the same location. The limitation on the accuracy with which hardware clocks at separated locations can be synchronized in practice using signals appreciably slower than the speed of light is analyzed in [11].

7. Conclusions

Spec is a specification language with a broad range of applications. The language is primarily intended for recording black box interface specifications in the early stages of design. The main advance over previous languages is the integration of facilities for supporting the design of large scale systems with provisions for specifying distributed systems and real-time constraints. The facilities for supporting large scale design include an underlying semantic model that limits non-local interactions, a syntax that localizes design decisions in the specification text and allows factoring of independent concerns into separate units, a facility for explicitly recording conceptual dependencies, and an inheritance mechanism for supporting views and incremental specification.

The Spec language was designed to support a software development paradigm that uses abstractions for simplifying complex systems and computer-aided design tools for detecting or preventing errors at the early stages of software development. This approach uses predicate logic to achieve black-box descriptions for the behavior of complex systems. This represents a fundamental difference from more classical approaches that use data flow diagrams as the primary specification tool. Data flow diagrams are typically used to record problem decompositions which do not have any behavioral descriptions for the bubbles except at the lowest levels of detail. Thus detailed questions about system behavior can only be answered by expanding the decomposition to the lowest level of detail. This can be a problem in very large systems because there are a huge number of bubbles at the lowest level, and the descriptions of those bubbles are expressed in terms of low-level details somewhat removed from what the user will see. The classical approach does not scale up to very large systems because it does not support self-contained abstractions, and does not provide a means for describing behavior precisely without introducing a maze of detail. The Spec language avoids functional decompositions, and instead partitions the specification according to the structure of interactions visible to the users. Low level details are suppressed without sacrificing precision by introducing abstractions with explicitly defined properties. Spec also provides facilities for defining a concept hierarchy which allows predicates to describe intended behavior at a high level, while providing concrete and precise definitions of abstract concepts in terms of more specific ones at many levels of detail. This structure can be used to answer many questions without expanding to the lowest level of detail, and supports the use of stepwise refinement for analysis, specification, and design.

The Spec language has been used to specify an airline reservation system [6]. Earlier versions of the language have been used to develop and enhance systems such as a text editor, a test support system for Pascal, a project management support system, and an electronic mail system in the context of classroom team projects with up to 15 people working for 20 weeks. Our experience has shown that it is sufficiently powerful to allow the specification of many kinds of software systems, and sufficiently flexible to allow software designers to express their thoughts without forcing them into a restrictive framework. We discovered that such specifications significantly reduced the need for verbal communication during the implementation phase, and that it was possible to design major extensions to a system by examining only the specification of the original architectural design, without access to the code implementing the specified modules.

The Spec language is sufficiently formal to support mechanical processing. Some tools for computer-aided design of software that are currently under investigation are syntax-directed editors, display generators, consistency checkers, design completion tools, test case generators, prototype generators. and tools for synthesizing partial implementations,

1. G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1987.

2. V. Berzins and M. Gray, ''Analysis and Design in MSG.84: Formalizing Functional Specifications'', *IEEE Trans. on Software Eng. SE-11*, 8 (Aug. 1985), 657-670.

3. V. Berzins, M. Gray and D. Naumann, ''Abstraction-Based Software Development'', *Comm. of the ACM 29*, 5 (May 1986), 402-415.

4. V. Berzins and Luqi, *The Semantics of Inheritance in Spec*, Computer Science, Naval Postgraduate School, 1987. NPS 52-87-032.

5. V. Berzins and Luqi, ''Languages for Specification, Design and Prototyping'', in *Handbook of Computer-Aided Software Engineering*, Van Nostrand Reinhold, 1988.

6. V. Berzins and Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley, 1988.

7. D. Bjoerner and C. Jones, *Formal Specification and Software Development*, Prentice Hall, Englewood Cliffs, NJ, 1982.

8. B. Cohen, W. T. Harwood and M. I. Jackson, *The Specification of Complex Systems*, Addison Wesley, Reading, MA, 1986.

9. J. V. Guttag and J. J. Horning, ''Report on the Larch Shared Language'', *Science of Computer Programming 6* (1986), 103-134.

10. C. A. R. Hoare, ''Notes on Communicating Sequential Processes'', Oxford University Computing Laboratory Technical Monograph PRG-33, Aug. 1983.

11. L. Lamport, ''Time, Clocks, and the Ordering of Events in a Distributed System'', *Comm. of the ACM 21*, 7 (July 1978), 558-565.

12. Luqi, V. Berzins and R. Yeh, ''A Prototyping Language for Real-Time Software'', *IEEE Trans. on Software Eng.*, October, 1988.

Initial Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268	1
Office of the Chief of Naval Operations Code OP-941 Washington, D.C. 20350	2
Office of the Chief of Naval Operations Code OP-945 Washington, D.C. 20340	2
Commander Naval Telecommunications Command Naval Telecommunications Command Headquarters 4401 Massachusetts Avenue NW Washington, D.C. 20390-5290	2
Commander Naval Data Automation Command Washington Navy Yard Washington, D.C. 20374-1662	1
Office of Naval Research Office of the Chief of Naval Research Attn. CDR Michael Gehl, Code 1224 Arlington, VA 22217-5000	1
Director, Naval Telecommunications System Integration Center NAVCOMMUNIT Washington Washington, D.C. 20363-5100	1
Space and Naval Warfare Systems Command Attn: Dr. Knudsen, Code PD50 Washington, D.C. 20363-5100	1

Ada Joint Program Office	1
OUSDRE(R&AT)	
The Pentagon	
Washington, D.C. 230301	
Naval Sea Systems Command	1
Attn: CAPT Joel Crandall	
National Center #2, Suite 7N06	
Washington, D.C. 22202	
Office of the Secretary of Defense	1
Attn: CDR Barber	
The Star Program	
Washington, D.C. 20301	
Naval Ocean Systems Center	1
Attn: Linwood Sutton, Code 423	
San Diego, CA 92152-5000	
National Science Foundation	1
Division of Computer and Computation Research	
Washington, D.C. 20550	
Director of Research Administration	1
Code 012	
Naval Postgraduate School	
Monterey, CA 93943	
Chairman, Code 52	1
Computer Science Department	
Naval Postgraduate School	
Monterey, CA 93943-5100	
LuQi	150
Code 52Lq	
Computer Science Department	
Naval Postgraduate School	
Monterey, CA 93943-5100	