



Calhoun: The NPS Institutional Archive
DSpace Repository

Faculty and Researchers

Faculty and Researchers' Publications

1987

Specifying Large Software Systems in Spec

Luqi; Berzins, Valdis

Naval Postgraduate School

V. Berzins and Luqi, "Specifying Large Software Systems in Spec", Technical Report NPS 52-87-033, Computer Science Department, Naval Postgraduate School, 1987.
<https://hdl.handle.net/10945/65242>

Downloaded from NPS Archive: Calhoun



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

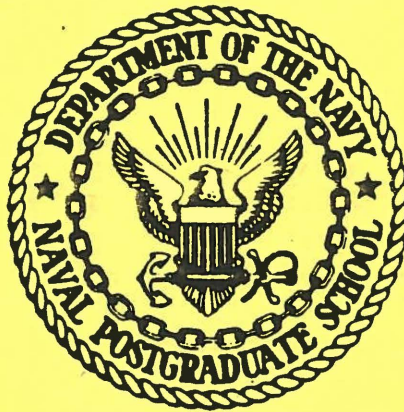
Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

<http://www.nps.edu/library>

712
NPS52-87-033

NAVAL POSTGRADUATE SCHOOL

Monterey, California



SPECIFYING LARGE SOFTWARE SYSTEMS IN SPEC

Valdis Berzins

Luqi

July 1987

Approved for public release; distribution unlimited

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

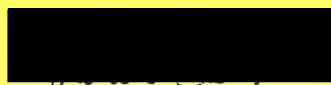
Rear Admiral R. C. Austin
Superintendent

D. A. Schradly
Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

Reproduction of all or part of this report is authorized.

This report was prepared by:



VALDIS BERZINS
Associate Professor
of Computer Science

Reviewed by:

Released by:



VINCENT Y. LUM
Chairman
Department of Computer Science



KNEALE T. MARSHALL
Dean of Information and
Policy Science

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 unlimited	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-87-033		7a. NAME OF MONITORING ORGANIZATION	
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b. OFFICE SYMBOL (If applicable)	7b. ADDRESS (City, State, and ZIP Code)	
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8a. NAME OF FUNDING / SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)	10. SOURCE OF FUNDING NUMBERS	
8c. ADDRESS (City, State, and ZIP Code)		PROGRAM ELEMENT NO.	PROJECT NO.
		TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) Specifying Large Software Systems in Spec			
12. PERSONAL AUTHOR(S) Valdis Berzins, Luqi			
13a. TYPE OF REPORT Technical	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) July 1987	15. PAGE COUNT 13
16. SUPPLEMENTARY NOTATION			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP	
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. The underlying computational model combines message passing with temporal events in a precisely defined way. The features of the language, especially those important for large scale design are presented by means of examples.			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Valdis Berzins		22b. TELEPHONE (Include Area Code) 408-646-2461	22c. OFFICE SYMBOL

Specifying Large Software Systems in 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. The underlying computational model combines message passing with temporal events in a precisely defined way. The features of the language, especially those important for large scale design are presented by means of examples.

1. Introduction

Spec is a formal language for writing black-box specifications for components of software systems. Black-box specifications are essential for realizing the benefits of abstractions in the software development process [2]. The critical early stages of software development are dominated by the tasks of building conceptual models of the proposed software and defining its interfaces. The Spec language is used in the functional specification stage for recording black-box specifications of the external interfaces of the proposed system, and in the architectural design stage for recording black-box specifications of the internal interfaces of the proposed system.

A formal specification language such as Spec is needed for defining the desired behavior of the proposed system before it is built, because English and other informal notations are too imprecise. Precision is important because in a large project many people have to agree on the interpretation of the specifications to produce a correct implementation. Written specifications are attractive as a communications medium in very large projects because the effort of writing a formal specification is independent of the number of people reading it, whereas communications overhead tends to increase with the size of the project in more informal techniques. Formal notation is important because it enables mechanical processing, opening the way to higher levels of computer-aided design than are currently used in software development. Programming languages such as Ada are formal, but 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.

There has been much previous work on providing programming language support for abstractions [4,6,8,13,18]. Spec has been intended primarily as a design tool. Much of the previous work on formal specifications has been focused on the problem of proving the correctness of programs [5,7,10,15,20]. Spec has evolved from an earlier specification language [1] and a rapid prototyping language for the design of large real-time systems [14], guided by extensive classroom experience in using formal specifications in multi-person projects [2]. The most important advances over the earlier language are the integration of time into the underlying model, the development of an inheritance mechanism [3], and the separation of granularity and control state considerations from the event-level interfaces of a module.

Spec is based on the event model of computation, and uses predicate logic for the precise definition of the desired behavior of modules. The most important ideas of this language are modules, messages, events, parametrization, and defined concepts. Spec also has a number of features that become important only for specifying very large systems, such as import/export controls for defined concepts, and view and inheritance mechanisms.

2. The Event Model

The Spec language uses the event model to define the behavior of black box software modules. The event model has been influenced by the actor model [9, 21]. The main differences from the actor model are the treatment of time and temporal events, and the treatment of multi-event transactions [1]. In the event model, computations are described in terms of modules, messages, and events. A module is a 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.

Modules can be used to model external systems such as users and peripheral hardware devices, as well as software components. Modules are active black boxes, which have no visible internal structure. The behavior of a module is specified by describing its interface. The interface of a module consists of the set of messages it accepts, along with its response to each kind of message it accepts. Each response consists of the messages sent out by the module in response to the most recent incoming message and the destinations of those messages.

Any module accepts messages one at a time, in a well-defined order that can be observed as a computation proceeds. Message transmission is assumed to be reliable, which means every message that is sent eventually arrives at its destination. While restrictions on message delay and ordering can be added for particular applications, these are not inherent in the event model. Unless explicitly stated otherwise, messages can have arbitrarily long and unpredictable transmission delays. The order in which messages arrive is not normally under the control of the designer.

In the event model each module has its own local clock. The local clocks of different modules are not necessarily synchronized with each other. This lack of synchronization is realistic since perfect synchronization of clocks at different locations is not possible in practice. Each event occurs at a well-defined instant of time, which is the time at which the destination module receives a message, according to its own local clock.

An event can be uniquely identified by specifying the module at which it occurred and the local time of the event. Each event determines a single message that arrived at the event. The attributes of an event are its location (the module at which it occurred), the time at which it occurred, and the message that arrived.

Each message has a sequence of zero or more data values associated with it. When messages are used to model subprogram invocations, these data values correspond to the arguments of the subprogram call. The other attributes of a message are its name, its condition, and its origin. The name of a message determines the kind of service being requested from the destination module. Modules which provide only a single service can accept anonymous messages. Formally the name of an anonymous message is the empty string. Exceptions are modeled as messages by means of the condition attribute, which can take on the values **normal** and **exception**. The condition of a message expressing a normal request for service is **normal**. The condition of a message reporting an abnormal event somewhere is **exception**, in which case the name of the message is the name of the exception condition. The origin attribute of a message identifies the event which triggered the sending of a message. When messages are used to model subprogram invocations, the location of the origin event determines the destination of the reply message.

The response of a module to a message is completely determined by the sequence of messages received by the module since it was created. The event model and the Spec language do not admit nondeterministic behavior other than that caused by unpredictable communication delays. Modules whose behavior is partially specified are treated as arbitrary members of the class of deterministic modules satisfying the specification.

Definition 1

A module is **mutable** if the response of the module to at least one message it accepts can depend on messages that arrived before the most recent incoming message.

Definition 2

A module is **immutable** if the response of the module to every possible message is completely determined by the most recent message it has received.

Mutable modules behave as if they had internal states or memory, while immutable modules behave like mathematical functions. A module is immutable if and only if it is not mutable.

Each module has the potential of acting independently, so that there is natural concurrency in a system consisting of many modules. 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 with a module cannot interfere with each other at the level of individual events. The response of a module to a message is under the control of the designer.

Events can also be triggered at absolute times. Such events are called **temporal events**. Formally a temporal event occurs when a module sends a message to itself at a time determined by its local clock. Temporal events are the means by which modules can initiate actions that are not direct responses to external stimuli.

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, because the only time orderings specified are those which are unavoidable and are agreed on by all observers. Because there is no global time reference, the only observable time orderings are derivable from discrete sequences of the following types of steps:

- (1) A message arrived at a module before a second message arrived at the same module, as determined by the local clock at the common location of both events.
- (2) The event which triggered the sending of a message happened before the event in which that same message is received by its destination.

3. Specifying Software in Terms of Events

The Spec language provides a means for specifying the behavior of four different types of modules: functions, state machines [16], abstract data types [11], and iterators [12,17]. The properties of these four kinds of modules are described below, with examples of each.

3.1. Functions

A function module calculates the value of a function in the mathematical sense. Function modules are immutable. Usually function modules provide only a single service, accepting anonymous messages. 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
      REPLY (y: real)
        WHERE y >= 0.0 & approximates(y * y, x)
    OTHERWISE
      REPLY EXCEPTION imaginary_square_root

  CONCEPT approximates(r1 r2: real)
    VALUE (b: boolean)
      WHERE b <=> abs((r1 - r2) / r2) <= precision
END
```

The `square_root` function accepts anonymous messages containing a single real number. The response of a module to a message can be defined with several cases introduced by `WHEN` clauses.

The predicate after each WHEN is called the precondition, and describes the conditions under which the associated response will 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 above, 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 that all cases are covered.

A REPLY describes the message sent back in response to an event. The reply message is sent to the module originating the message that arrived in the event. In terms of the event model, the destination of the reply message is the location of the origin event for the message that triggered the reply, which is contained in the origin attribute of the message. 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.

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 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.

Whenever a message arrives which matches a MESSAGE header of a module and satisfies the precondition (WHEN) of one of the cases, then a response must be sent which matches the REPLY header and satisfies the associated postconditions (WHERE). A message matches a header if the message has the specified name, condition, and number of data values, and if each data value belongs to the specified data type. A message satisfies a predicate if the predicate is true for the data values in the messages mentioned in the predicate. Preconditions mention only an incoming message, while postconditions mention both an incoming and an outgoing message. Messages without any WHEN clauses have a single case whose precondition is always true. If the precondition for more than one case is satisfied, all of the associated responses must be sent and the constraints of all the associated postconditions must be met simultaneously. Overlapping preconditions are not recommended because they can lead to inconsistencies.

3.2. Machines

A machine is a module with an internal state, i.e. machines are mutable modules. An example of a machine is shown below.

MACHINE inventory

% assumes that shipping and supplier are other modules.

STATE (stock: map{from :: item, to :: integer})

INVARIANT FOR ALL (i: item :: stock[i] >= 0)

INITIALLY FOR ALL (i: item :: stock[i] = 0)

MESSAGE receive_shipment(i: item, quantity: integer)

WHEN quantity > 0

TRANSITION *stock[i] = stock[i] + quantity

OTHERWISE REPLY EXCEPTION empty_shipment

MESSAGE order(item_ordered: item, quantity_ordered: integer)

WHEN $0 < \text{quantity_ordered} \leq \text{stock}[\text{item_ordered}]$

SEND shipping_order(item_shipped: item, quantity_shipped: integer)

TO shipping

WHERE item_shipped = item_ordered, quantity_shipped = quantity_ordered

```

TRANSITION *stock[item_ordered] + quantity_ordered = stock[item_ordered]
WHEN 0 < quantity_ordered >= stock[item_ordered]
SEND shipping_order(item_shipped: item, quantity_shipped: integer)
  TO shipping
  WHERE item_shipped = item_ordered,
        quantity_shipped = stock[item_ordered]
SEND back_order(item_backordered: item, quantity_backordered: integer)
  TO supplier
  WHERE item_backordered = item_ordered,
        quantity_backordered = quantity_ordered - stock[item_ordered]
% storage of and delayed response to backorders are not shown here
TRANSITION *stock[item_ordered] = 0
OTHERWISE REPLY EXCEPTION empty_order
END

```

The behavior of a machine is described in terms of a conceptual model of its state, rather than directly in terms of the messages that arrived in the past, because such descriptions are usually shorter and easier to understand. The components of the conceptual model of the state are declared after the keyword STATE, and restrictions on the set of meaningful states are given after the keyword INVARIANT. Restrictions on the initial state are given after the keyword INITIALLY. The restrictions after INVARIANT must be satisfied in all reachable states, while the restrictions after INITIALLY must be satisfied only in the first state.

State changes are described by predicates after the keyword TRANSITION. In such statements, variables of the form *x refer to the value of x in the new state (just after the arrival of the most recent message), while variables of the form x refer to the value of x in the old state (just before the arrival of the most recent message). 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). The *x notation can only be used in the INVARIANT, the TRANSITIONS, and in WHERE clauses describing the output in terms of the new state. The Spec language follows the convention that components of the state of a machine or the model of an abstract data type do not change unless there is a change explicitly described in a TRANSITION clause.

The SEND statement is used instead of REPLY to describe messages sent to destinations other than the origin of the incoming message. A SEND statement means that a message satisfying the description must be sent to the given destination. There can be only one REPLY, but there can be any number of SEND's. If there is more than one SEND, the message transmissions can be performed concurrently or one at a time in any order, without waiting for any responses. SEND statements are useful for describing distributed systems with a pipeline structure.

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

```

TYPE rational
  MODEL (num den: integer)
  INVARIANT den ~ = 0

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

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

MESSAGE equal (x y: rational)
  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 value set 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.

Each instance of the type can be represented as a tuple containing the data components declared after the MODEL keyword. The restrictions on the components of the model are described in the INVARIANT, which selects a subset of the tuple data type defined by the MODEL to serve as the conceptual representation. 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, which are ratios of pairs of integers. The invariant must exclude pairs with zero denominators, because the interpretation of the pairs as ratios does not make sense in that case. It is not necessary for there to be a 1 : 1 correspondence between conceptual representations and values of the abstract data type, although in such cases the model is not fully abstract, and some extra care must be taken in defining the operations. Our example does not have unique conceptual representations, because the pairs [1, 2], [2, 4], and [-1, -2] all represent the same rational number, namely one-half. 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. Some additional restrictions that would 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 need not necessarily be satisfied by the implementation data structure and does not restrict the designer's choice of implementations. The invariants on the implementation data structures will often be much more complicated than the conceptual invariants, because the implementation invariants may critically affect efficiency. Most books on data structures are 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. The data values can be described as if they were instances of the tuple data type whose components are specified in the MODEL. The notation $x.y$ can be used to refer to the y component of the abstract data value x . Such references are allowed only inside the module defining the abstract data type. The specifications of other modules may describe the values of abstract types only in terms of the messages it provides and the CONCEPTS it exports.

The example illustrates that messages with a single case can be defined without using WHEN. 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.

An example of a definition for a mutable type is shown below.

```
TYPE queue{t: type}
  % mutable version
MODEL (e: sequence)
  % 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 = [ ], 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)
  WHERE q.e = append(*q.e, [x])
  TRANSITION FOR SOME (y: t :: q.e = append(*q.e, [y]))
  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 $*x \sim x$ for some component x). Mutating operations, such as *enqueue* in the example above, are described using TRANSITION clauses.

The distinction between mutable and immutable data types is subtle and is commonly misunderstood. The state changes due to the mutating operations of a mutable data type are associated with the data values rather than with program variables. If several variables share the same mutable value and a mutating operation is applied, then the change is visible in all of the variables sharing the mutable value. Correctly programming with mutable data types is difficult, so that mutable data types should be used only if required to faithfully model the behavior of a system containing objects whose properties change with time or to meet tight performance constraints.

Mutating operations are usually implemented as procedures with read-only parameters, because they change the properties of existing objects, rather than creating new data objects. Since a level of indirection is necessary to properly implement mutable data values in shared contexts, a mutable value is usually implemented as a pointer that directly represents the identity of the object and is not changed by mutating operations. The properties of such an object are represented only indirectly. Procedures with update parameters (e.g. in out parameters in Ada) are usually not mutating operations, unless the mutating operation also creates and returns a new object. Such procedures are usually used to implement operations of immutable types, under the restriction that the new output object must be delivered in the same variable as the input object [19]. Space efficiency is the usual justification for such restrictions.

The limited private types of Ada are also not usually used in implementing mutable abstract data types. Such programs are usually implementations of immutable types that use mutable concrete representations, for which sharing between variables cannot be allowed without causing errors. In such cases aliasing due to procedure calls must be avoided in addition to the assignment operators explicitly prohibited by the limited private declaration.

3.4. Iterators

An iterator is a module that generates a sequence of values one at a time. An example of a specification for an iterator is shown below.

ITERATOR primes

```

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

CONCEPT increasing_order(s: sequence{integer})
  VALUE (b: boolean)
  WHERE b <=> FOR ALL (i j: integer SUCH THAT 1 <= i < j <= length(s) :: s[i] < s[j])

CONCEPT prime(i: integer)
  VALUE (b: boolean)
  WHERE b <=> i > 1 & FOR ALL (k: integer SUCH THAT 1 < k < i :: i mod k > 0)
END

```

The YIELD 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 an iterator is used to control a data driven loop. Iterators are control abstractions, because they hide the details of the control sequence needed to generate the YIELDED values, and specify only the content of the sequence and the order in which the elements are to be generated.

Any message with a YIELD is an iterator, so that iterators can be defined as operations of an abstract data type or a machine. This is an important application of iterators, because it is

otherwise difficult to scan all of the elements of an abstract collection without exposing the data structure used to implement the collection.

Iterators are usually defined as black boxes at the architectural design level, and are usually called in the algorithms implementing higher level modules in terms of lower level modules, which are defined during the module design stage. An example of a module design for a function *factors* using the *primes* iterator of the previous example is shown below.

```
function factors(i: integer)
  s: set{integer} := { }
  foreach p in primes(i) do
    if i mod p = 0 then s := union(s, {p}) fi
  od
  return (s)
end
```

In a program, an iterator is always used to drive a *foreach* loop. The body of such a loop is executed once for each element of the sequence generated by the iterator. The next element generated by the iterator is always bound to the loop control variable (p in the example above). The loop stops when the iterator runs out of values, if it ever does.

Iterators can also be used in specifications for other modules. In such a context, the iterator call is treated as if it were a function returning a sequence, and predicates are used to describe the relation between the sequence and something else. An example of such a definition is shown below.

```
CONCEPT big_prime_candidate(n: integer)
  VALUE (x: integer)
  WHERE FOR ALL (k: integer SUCH THAT k in primes(n) :: x mod k = k - 1)
```

Note that this is a specification rather than a program, and that the sequence has been used to specify the range of a bound variable in a quantifier rather than to control a loop.

4. Features for Specifying Large Systems

The Spec language contains a number of features that are needed mostly for specifying large systems. Some of these features include parameterized modules, defined concepts, and an inheritance mechanism.

4.1. Parameterized Modules

A parametrized module specifies a family of modules rather than an individual module. A parametrized module looks like an ordinary module definition except that there can be parameters after the module name, with an optional WHERE clause restricting the values of the parameters. The specifications for *square_root* and *queue* given in the previous section are examples of parametrized modules. Such a definition defines one module for each legal set of values for the parameters of the module. The parameters can range over either data values or modules (functions, types, machines, or iterators). Actual parameters that are functions and types can be concepts as well as modules.

Parametrized modules can be implemented in Ada as generic packages. Parametrized modules can be implemented in other languages by using a preprocessor to substitute actual parameter values for the formal parameters of the module. This can be done by means of a macro processor (such as m4 on unix) or by means of a script for an editor (such as sed on unix).

4.2. Concepts

A **concept** in the Spec language is a constant symbol, predicate symbol, or function symbol that can be used in constructing the logical assertions defining the behavior of modules. Concepts can be viewed as abbreviations or macros, with recursive definitions allowed. Concepts without

formal arguments are interpreted as constants. A constant can be either a symbolic name for a data value or a symbolic name for a data type. Concepts with formal arguments are interpreted as predicate symbols if they have one VALUE and its type is boolean, and as function symbols otherwise.

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.

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 module: 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 can also be used to mix formal and informal specifications, by a formal definition of a precondition, postcondition, invariant, or transition in terms of some concepts, and then providing informal definitions for the concepts. 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 very important in practical projects with tight schedules.

Concepts represent the properties of the software that are needed to explain or describe 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.

A function should be defined as a module of type FUNCTION if it is part of the model of the software system, and it should be defined as a concept that is part of a module if the function is needed to specify the behavior of the module, but is not part of the model of the system at the current level of description. If a function is needed to specify the behavior of a module at a high level of the architectural design, and is also one of the components used to realize that module at a lower level, then it should be defined as a concept attached to the module at the higher level and exported. At the lower level it should be specified as a FUNCTION module, which imports the concept from the higher level module and has a trivial definition in terms of the imported concept.

4.3. Views and Inheritance

The Spec language has an inheritance mechanism which can be used for specifying constraints common to the interfaces of many modules and for view integration. Specifying constraints common to many interfaces is essential for achieving interface consistency in very large systems. The interface of a system to each class of users can be a separate view of the system, perhaps specified by different designers. A total picture of the system is formed by expanding the definition of a module that inherits all of the individual views. The inheritance mechanism and the rules for combining different versions of messages and concepts inherited from multiple parents are described in more detail in [3].

5. 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 language has a precise semantics and a simple underlying model. 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. The 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, consistency checkers, design completion tools, test case generators, and prototype generators.

1. V. Berzins and M. Gray, "Analysis and Design in MSG.84: Formalizing Functional Specifications", *IEEE Trans. on Software Eng. SE-11*, 8 (Aug. 1985).
2. V. Berzins, M. Gray and D. Naumann, "Abstraction-Based Software Development", *Comm. of the ACM* 29, 5 (May 1986), 402-415.
3. V. Berzins and Luqi, "The Semantics of Inheritance in Spec", *submitted to the ACM Symposium on Principles of Programming Languages*, 1988.
4. "Ada Programming Language", American National Standards Institute/MIL-STD-1815A, DoD, 1983.
5. J. A. Goguen, J. W. Thatcher, E. G. Wagner and J. B. Wright, "Abstract Data Types as Initial Algebras and the Correctness of Data Representations", in *Proc. Conf. on Computer Graphics, Pattern Recognition, and Data Structures*, 1975, 89-93.
6. A. Goldberg and D. Robinson, *Smalltalk-80: The Language and its Implementation*, Addison Wesley, Reading, MA, 1983.
7. J. V. Guttag, E. Horowitz and D. R. Musser, "Abstract Data Types and Software Validation", *Comm. of the ACM* 21, 12 (1978).
8. M. Herlihy and B. H. Liskov, "A Value Transmission Method for Abstract Data Types", *Trans. Prog. Lang and Systems* 4, 4 (Oct. 1982), 527-551.
9. C. E. Hewitt and H. Baker, "Actors and Continuous Functionals", in *Formal Description of Programming Concepts*, North-Holland, New York, 1978, 367-387.
10. C. A. R. Hoare, "Proof of Correctness of Data Representations", *Acta Informatica* 1, 4 (1972), 271-281.
11. B. Liskov and S. Zilles, "Programming with Abstract Data Types", *Proc. of the ACM SIGPLAN Notices Conference on Very High Level Languages* 9, 4 (Apr. 1974), 50-59.
12. B. H. Liskov, A. Snyder, R. Atkinson and J. C. Schaffert, "Abstraction Mechanisms in CLU", *Comm. of the ACM* 20, 8 (Aug. 1977), 564-576.
13. B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. Schaffert and R. Scheifler, *CLU Reference Manual*, Springer Verlag, 1981.
14. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *to appear in IEEE TSE*, 1987.
15. R. Nakajima, "IOTA", *IEEE Trans. on Software Eng.*, Feb. 1985.
16. D. L. Parnas, "A Technique for Software Module Specification with Examples", *Comm. of the ACM* 15, 5 (May 1972), 330-336.
17. M. Shaw, W. A. Wulf and R. L. London, "Abstraction and Verification in ALPHARD: Defining and Specifying Iteration and Generators", *Comm. of the ACM* 20, 8 (Aug. 1977), 553-564.
18. M. Shaw, *Alphard: Form and Content*, Springer Verlag, 1981.
19. D. M. Volpano and R. B. Kieburtz, "Software Templates", CS/E 85-011, Department of Computer Science and Engineering, Oregon Graduate Center, 1985.
20. W. A. Wulf, R. L. London and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs", *IEEE Trans. on Software Eng. SE-2*, 4 (Dec. 1976), 253-265.

21. A. Yonezawa,, "Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics", Ph.D. Thesis, MIT, 1977.

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 Analyses 2000 N. Beauregard Street Alexandria, VA 22311	1
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
Valdis Berzins Code 52Be Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	100

