Comput. Lang. Vol. 16, No. 2, pp. 113–127, 1991
Printed in Great Britain

0096-0551/91 $3.00 + 0.00
Pergamon Press plc

BLACK-BOX SPECIFICATION IN SPEC

VALDIS BERZINS
Computer Science Department, Naval Postgraduate School, Monterey, CA 93943, U.S.A.

(*Received* 22 *January* 1990)

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.

Black-box specifications Abstractions Specification language Computer aided software engineering Distributed systems Real-time systems

1. INTRODUCTION

Spec is a formal language for writing black-box specifications for software systems and their subsystems. Black-box specifications are essential for realizing the benefits of abstractions in the software development process [3]. 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 can be used in the initial requirements analysis for recording conceptual models of the problem domain, in the functional specification stage for defining the external interfaces of the proposed system, in the architectural design stage for defining the internal interfaces of the proposed system, and in the implementation stage for recording concrete invariants and other design information. The use of formal specifications at these stages can control the conceptual complexity of very large systems by factoring out independent aspects of the system into locally understandable units, and providing the precision needed for computer-aided verification and validation processes. Problems in software maintenance have also been linked to lack of specification and design information [30]. The Spec language has been designed primarily for supporting software development and evolution on a large scale. The language includes facilities suitable for specifying parallel, distributed, and real-time systems.

A black-box approach has the advantage of emphasizing behavior while suppressing internal mechanisms for the systems at any given level of detail. This contrasts with the popular functional decomposition approach to requirements analysis and functional specification, in which high level processes are explained by breaking them into combinations of lower level processes, often using data flow diagrams. Functional decomposition is effective for describing the behavior of systems on a moderate scale, but for very large systems it fails to provide a conceptually manageable picture of the system as a whole, because the behavior of the subsystems is described directly only at the lowest levels of the decomposition, which can be many levels removed from the global view and may contain thousands of subsystems. For very large systems it is useful to restrict functional decomposition to the design of the software architecture, and to take a black-box approach to specifying user-level functions of the system.

A formal specification language such as Spec is needed for effectively defining the desired behavior of a proposed system at a simple abstract level, 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 [6]. 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 [11, 14, 19, 26, 29]. Previous work on formal specifications has focused mostly on the problem of proving the correctness of programs [13, 16, 21, 33, 35]. Spec is intended primarily for supporting the use of abstractions in the design of software systems. Surveys of related work can be found in [10, 31]. Spec has evolved from an earlier specification language [2] and a rapid prototyping language for the design of large real-time systems [27], guided by extensive classroom experience in using formal specifications in multi-person projects [3]. The most important advances over the earlier language are the integration of time into the underlying model, the development of an inheritance mechanism [4], 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 [17, 18] because it is based on models rather than theories. While it is possible to write Spec axioms in the conditional equation form commonly used in algebraic approaches, the use of models and axioms of other forms can often 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 is based on the premise that interfaces involving state changes are inherently dependent on the implementation language. 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. 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 [9] 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 [10]. A model based approach for specifying distributed systems based on Petri nets is described in [23, 24].

The GIST language is based on a global state model approach that describes behavior independently of interfaces [22], and is intended for use in the early stages of requirements analysis where properties of the entire application are being determined without assigning boundaries or allocating functions to either the proposed software system or its environment. GIST treats system behavior as a sequence of global system states. Localization of information, treatment of distributed systems, and treatment of real-time constraints are explicit design goals of Spec. Spec is object-oriented, in the sense that it avoids global states and defines behavior interms of events at system boundaries. While Spec and GIST can be used for similar purposes, they emphasize different stages of the process. GIST focuses on the earliest problem formulation stages, while Spec focuses on the formulation and refinement of the proposed system.

Spec does not emphasize clear box descriptions, although interconnections between modules can be specified. The notation is most effective at the architectural design stage when coupled with graphical support for describing module interconnections [27, 32].

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, atomic transactions, and defined concepts. Events can be used for defining timing constraints, while localized states and atomic transactions are important for specifying distributed or concurrent systems. Spec supports reuse of abstractions via inheritance and generic modules. Spec also has features important for specifying large conventional systems, such as import/export controls for defined concepts, and view and inheritance mechanisms.

2. SPECIFYING SOFTWARE IN TERMS OF EVENTS

The Spec language provides a means for specifying the behavior of three different types of modules: functions, state machines, and abstract data types. These types of modules span the basic building blocks of software systems. Special types of messages can also be used to model generators or iterators [25, 28]. The properties of these kinds of modules and message are described below, with examples of each.

2.1. Functions

Functions are modules without internal memory. From a black-box viewpoint this means the response of a function module to a stimulus cannot be influenced by earlier stimuli. If a function module has been completely specified, then there is exactly one legal response to each stimulus, and the module computes a single-valued function in the mathematical sense. The response of a function module need not be completely determined by the most recently received stimulus. Incompletely specified modules admit non-deterministic behavior unless designer explicitly constrains them to be deterministic. Function modules are emphasized in functional programming [1], but they can be implemented in any programming language via appropriate coding conventions.

An example of a completely specified function is shown below.

```
FUNCTION next_token

  MESSAGE(s: string)
    REPLY(next rest: string)
      --Extracts the next non-blank substring from the front of s.
      --Any leading spaces are discarded.
    WHERE no_spaces(next), delimiter(rest),
      SOME(b: string::all_spaces(b) & s = b || next || rest),
      next = "" < = > all_spaces(s)

    CONCEPT space: char WHERE space = ' '

    CONCEPT all_spaces(s: string)
      VALUE(b: boolean)
      WHERE b < = > ALL(c: char SUCH THAT c IN s :: c = space)

    CONCEPT no_spaces(s: string)
      VALUE(b: boolean)
      WHERE b < = > ALL(c: char SUCH THAT c IN s :: c~ = space)

    CONCEPT delimiter(s: string)
      VALUE(b: boolean)
      WHERE b < = > (s = "" | s[1] = space)
    END
```

This function performs a simple lexical scanning function, with two outputs, the next token, and the rest of the string. Spec CONCEPTS have been used to simplify the definition and to factor out independent concerns. Introducing an explicitly defined concept modularizes the specification. This helps simplify the postcondition and supports stepwise refinement and localization of information. The definition of the concept can be delayed or left as an informal comment when the concept is identified and used to express a compound precondition or postcondition. Since the example has been written to be self-contained, it is not typical of specifications for large systems. If we were specifying a fragment of a large system, definitions of standard reusable concepts such as *space*, *all_spaces*, and *no_spaces* would be imported or inherited from a specification library, and would not be re-invented at each use. Such a library should contain a generalized version of the *delimiter* concept, defined in terms of a generic parameter representing a set of delimiter characters, rather than the specific constant *space*.

Messages are distinguished from concepts to provide a clear and mechanically recognizable distinction between the required parts of a proposed system and definitions that are needed for explanation, specification, verification, and assessment of test results, but which need not correspond to parts of the implementation of the proposed system.

The Spec language gains its expressive power from logical quantifiers such as SOME ("there exists") and ALL ("for all"). The punctuation mark "::" separates the range declarations for the bound variables of a quantifier from the statement forming the body of the quantifier. A range declaration specifies the types of the bound variables and can include an optional subtype restriction introduced by the keywords "SUCH THAT". In many practical applications, such as in the example given above, quantifiers range over finite sets. In these special cases, the quantifiers can be directly implemented using loops that are guaranteed to terminate. The Spec language also includes unbounded quantifiers. This facility can support brief descriptions of the requirements for a module which can be very useful for proving properties of the module. However there can be no general automatic procedure for mapping specifications containing unbounded quantifiers into programs that are guaranteed to both terminate and correctly evaluate the quantifier in all cases.

Spec quantifiers are general operators on sets of values defined by range declarations. Spec allows users to define additional quantifiers, and provides a set of pre-defined generalized quantifiers including SUM, PRODUCT, MAXIMUM, MINIMUM, NUMBER, UNION, and INTERSECTION in addition to the usual logical quantifiers SOME and ALL. This facility can be used to represent the requirements for a software system using standard mathematical constructs such as limits, integrals, and infinite series. Spec allows variables ranging over types and functions, and includes second order quantifiers, which can be useful for defining general properties of generic modules.

In general the response of a module to a message can be defined with several cases introduced by WHEN clauses. Each WHEN clause expresses a precondition, i.e. a predicate 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. Different cases are usually distinguished by qualitatively different kinds of responses, such as normal outputs vs exceptions, or messages sent to different destinations under different conditions.

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.

OTHERWISE is an abbreviation for the case where none of the other WHEN statements apply. 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. In Spec, there are two explicit representations for "undefined". The first, "?", indicates the design has not been finished, and can be read as "to be determined". The other representation, "!", indicates that a case is deliberately left undefined, and can be read as "this case should never arise". Designs containing "!"s are dangerous because they may lead to brittle systems. Such designs usually are justified on grounds of efficiency, and involve tradeoffs that need careful review.

A REPLY describes the message sent back in response to a stimulus. The reply is sent to the module originating the message that arrived in the stimulus, which is identified by the implicit 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. EXCEPTION can also appear after MESSAGE in the specification of an exception handler, indicating that the stimulus must be an exception condition. Thus responses to exception conditions are specified in the same way as responses to normal messages.

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 declaration in the specification of the 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 declaration 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 in the state where the formal arguments of the visible message declarations are bound to the actual data values in the message. Only the incoming message is visible in a precondition, while the incoming message and all associated outgoing messages are visible in a postcondition.

To summarize, the basic building blocks for black-box specifications are precondition/postcondition pairs expressed in predicate logic, which provide a means for recognizing the required responses of a module. These preconditions and postconditions provide exactly the information needed for formal verification or automatic classification of test results. Message names, exception names, and the guarded response structure are included to help separate logically distinct concerns, thus helping designers and analysts find the parts of a formal specification addressing any given question concerning the intended behavior of the proposed system. This structure, together with intermediate definitions represented as Spec CONCEPTs allows analysts to keep individual formal assertions brief, which is necessary for effective human comprehension.

2.2. Machines

A machine is a module with an internal state, which means the responses of a machine can depend on previous stimuli. Systems providing long term memory, such as databases, are classified as abstract machines in the Spec language. An example of a machine is shown below.

```
MACHINE ticket_system
  INHERIT time_unit
  STATE(outstanding: set{ticket_id})
  INVARIANT true
  INITIALLY outstanding = { }

  MESSAGE ticket(violator: person, ticket_id: integer)
    SEND check_on_payment(violator: person, ticket_id: integer) TO ticket_system
      WHERE (30 days) < = DELAY < (31 days)
    TRANSITION outstanding = *outstanding U {ticket_id}

  MESSAGE payment(violator: person, ticket_id: integer)
    TRANSITION outstanding = *outstanding - {ticket_id}

  MESSAGE check_on_payment(violator: person, ticket_id: integer)
    WHEN ticket_id IN outstanding
      SEND letter(s: string) TO violator WHERE warning(s)
      SEND check_on_payment(violator: person, ticket_id: integer) TO ticket_system
        WHERE (30 days) < = DELAY < (31 days)
    OTHERWISE- - do nothing

  CONCEPT warning(s: string) VALUE(b: boolean)
    - -True if s is a letter requesting immediate payment of the fine.
END
```

This example shows a simplified system for keeping track of payments of fines for traffic violations. The state of the machine consists of the set of tickets that have been issued and not yet paid. The example illustrates the treatment of message delays in Spec and the description of activities with time-out conditions.

The DELAY keyword denotes the length of the time interval between the instant a module accepts a stimulus message and the instant an associated response message is accepted by its destination. The stimulus message and the response are identified by the context in which the DELAY expression appears. The time it takes to prepare a message is not distinguished from the time it takes to transmit the message from one module to another or time spent in a queue waiting for the destination module to accept the message—the DELAY includes all of these. The DELAY can be used to define arbitrary timing constraints, including upper or lower bounds, or both as in the example. If no explicit constraints on the DELAY are given, then the response must arrive after some finite but unbounded delay.

Spec DELAYs represent time-valued expressions, which are used to express declarative constraints on the real-time behavior of a system. The DELAY expression in Spec differs substantially from the DELAY statement in Ada, which is an imperative executable statement. Ada DELAY statements can be used to realize lower bounds on time delays, and can be used to realize upper bounds only in some very restricted contexts.

A time-out is represented as a message sent from the machine to itself with a specified delay. While this message is in transit, the machine is free to respond to other requests. When the time-out message, *check_on_payment* in the example, is received, the machine must check whether the expected responses to the original request have arrived. If they have, then the time-out message is ignored, and otherwise the actions associated with the time-out conditions are carried out. The time-out processing is specified as a separate event rather than as part of the response to the original stimulus because it depends on a future state, and the machine should be free to respond to other stimuli while it is waiting for the time-out. Similar structures occur in specifying robust distributed systems such as communications protocols.

The example also illustrates the use of dimensioned quantities such as "(30 days)" and the use of an INHERIT clause to refer to concepts defined in another module. The example refers to a module called "time_unit", which defines time units such as "days". Formally a time unit is a constant of type "duration", which is defined to be a subtype of real (every duration is a real number, but not every real number is a duration). The module "time_unit" is part of the specification library defined in [7].

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 arrive 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, plain variables refer to their values in the new state (just after the arrival of the stimulus), while variables prefixed with a * refer to their values in the previous state (just before the arrival of the stimulus). In the event model which forms the basis for Spec state changes occur at events where the triggering stimuli arrive and take effect instantaneously. Formally "*" is a temporal operator. The temporal logic underlying Spec includes other temporal operators as well [7], but these are not needed for most conventional applications.

The transitions in the example are specified using equations. Equations can describe a transition either forwards or backwards in time, whichever is simpler. 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. Spec follows the convention that *x = ! in the initial state for all variables x, which means that the predecessor of the initial state is undefined. The Spec language also follows the convention that components of the state of a machine or the model of an abstract data type do not change unless the component is explicitly mentioned in a TRANSITION clause.

A TRANSITION predicate provides a declarative description of the required relation between the initial state and the final state which can be used to recognize acceptable state changes after the fact. A TRANSITION predicate differs substantially from an assignment statement, which is an imperative executable statement that can be used to realize a state transition. The subset of TRANSITION predicates which have the form of equations defining the final state in terms of the initial state, including those shown in the example, can be transformed into assignment statements directly. However, it is also legal to write TRANSITION predicates which define the initial state in terms of the final state, or which implicitly define a constraint on the state change that can be used to decide whether a transition is correct if the initial state and the final state are both given, but may not provide any means for deriving one of these states from the other. In such cases there may not be any simple uniform procedure for constructing the final state from the initial state and the TRANSITION predicate other than an exhaustive enumerate-and-test loop. Currently the construction of an efficient implementation in these cases depends on the knowledge and skill of a software engineer, although such transformations may be eventually carried out by knowledge-based software systems.

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. SEND statements are useful for describing distributed systems with a pipeline structure. A MESSAGE can have only one REPLY, but it can have any number of SENDs. In the example, the first case of the *check_on_payment* message has multiple responses, one that sends a letter and another that sets up a new time-out condition. 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. Message transmission is assumed to be asynchronous in Spec. Synchronization is specified by including explicit acknowledge messages when it is part of the system requirements.

2.3. Types

A type module defines an abstract data type. A realization of 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 known as 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.

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. 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. Functions are immutable and machines are mutable modules. A type can be either mutable or immutable.

The distinction between mutable and immutable modules is important for supporting optimization and software evolution. For example transformations that eliminate common subexpressions are valid only for functions, since an operation on a mutable module could return a different result each time it is invoked, even if all of the input values are the same. Similarly knowledge of which operations can cause state transitions and which operations can be affected by which components of the state can help to assess the impact of a proposed evolutionary change, especially if the change influences the meaning or form of a data structure representing state information.

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

```
TYPE date
  INHERIT equality{date}

  MODEL(day month year: nat)
  INVARIANT ALL(d: date :: 1 < = d.day < = 31 & 1 < = d.month < = 12 &
    0 < = d.year < = 99)

  MESSAGE create(d m y: nat)
    WHEN 1 < = d < = 31 & 1 < = m < = 12 & 0 < = d.year < = 99
      REPLY(d: date) WHERE d1.day = d, d1.month = m, d1.year = y
    OTHERWISE REPLY EXCEPTION illegal_date
    MESSAGE "<"(d1 d2: date) REPLY(b: boolean)
      WHERE b <=> 0 < (d2.year-d1.year) MOD 100 < 50
                        |d1.year = d2.year & d1.month < d2.month
                        |d1.year = d2.year & d1.month = d2.month & d1.day < d2.day
      --note 12/31/99 < 01/01/00
      --< is a total ordering on any time interval less than 50 years long
      --but it is not transitive on longer intervals

    MESSAGE "<="(d1 d1: date) REPLY(b: boolean)
      WHERE b <=> d1 < d2|d1 = d2
  END
```

The example shows a type for representing calendar dates. This data type corresponds to the short form of dates commonly used in business, which have only two digits for the year. Such dates represent relative rather than absolute points in time, with a cycle long enough relative to human experience to make references unambiguous in most practical contexts. The abstract type has been defined so that it is free from "overflow" conditions and will be usable indefinitely. The interesting part is the definition of the "<" ordering, which is defined in a relative fashion, depending only on the two dates to be compared and not on any assumptions about the relation of dates to absolute time. While this relation is not a global ordering on the entire data type, it provides a consistent and natural local total ordering on any subinterval short enough not to "wrap around" (i.e. less than 50 years). The example is a simplification of a realistic application since it does not account for the fact that different months can have different lengths.

Abstract data types are usually specified via conceptual models in the Spec language. The conceptual models are used to visualize and describe the value set of the type, to specify the behavior of the operations, and to provide a mental picture of the type for the programmers who use the operations of the type. The conceptual model is chosen for clarity, and is often different from 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.

An abstract data type can have many different realizations, all of which must provide operations with the specified behavior. The conceptual model provides one possible concrete realization for the type. Spec definitions of abstract data types with non-empty conceptual models are constructive in this sense. It must be possible to find a function which maps the sequence of operations used to construct an arbitrary instance of the type into the corresponding conceptual model to check whether the implementation satisfies the specification. For types that are specified in terms of a unique conceptual model for each instance, a simplified function can be used for this purpose which maps the concrete data structure into the conceptual model directly, without considering the sequence of operations that were used to construct the instance.

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.

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 will often be much more complicated than the conceptual invariants, because implementation invariants often determine 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. 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. Instances of the type are denoted by expressions containing Spec variables. Such variables must be declared in input and output messages, conceptual models, generic parameters, or quantifiers. The notation x.y can be used to refer to the y component of the abstract data value x. 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. This restriction helps to enforce the locality of information principle characteristic of object-oriented design and programming. Spec also supports the concept of object refinement via a multiple inheritance mechanism, as discussed in Section 3.3.

One difference between Spec and many object-oriented programming languages is that a Spec type module defines the behavior of a type manager rather than the behavior of a typical instance. This avoids special treatment for object creation and eliminates the need for concepts such as "self" and meta-classes, which we have found to be a source of confusion for many programmers. Cases where it is natural to send a message to an instance, such as messages from an aircraft control system to an individual airplane, are handled by a convention similar to Simula 67: if x is an instance of type t then

SEND operation($a_1, \ldots, a_n$)TO x

is treated as an abbreviation for

SEND operation(x, $a_1, \ldots, a_n$)TO t.

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 employee
  INHERIT mutable (employee)
    --Inherit definition of the concept "new".

  MODEL(name: string, salary: money)
  INVARIANT ALL(e: employee :: e IN *employee => e.name = *e.name)
    --The name of an employee cannot change.

  MESSAGE hire(name: string, salary: money)
    REPLY(e: employee)
    TRANSITION new(e), e.name = name, e.salary = salary
    --The predicate new(e) means e is newly created.

  MESSAGE raise(e: employee, r: money)
    TRANSITION e.salary = *e.salary + r

  MESSAGE name(e: employee)
    REPLY(s: string) WHERE s = e.name

  MESSAGE salary(e: employee)
    REPLY(m: money) WHERE m = e.salary

  MESSAGE fire(e: employee)
    TRANSITION not(e IN employee)
    --The employee is removed from the type, and ceases to exist.

  CONCEPT money: type
    WHERE subtype(money, nat)
END
```

This example specifies a data type representing the employees of an organization. The identity of an employee remains the same, even though other properties of the employee, such as the salary, may change. To illustrate invariants that restrict the range of legal state transitions, we have specified that the name of an employee cannot be changed, although that is not entirely realistic. The reader is invited to formulate an explicit invariant restricting salaries to be non-decreasing, which is an implicit property of the definition shown. The *hire* operation illustrates the specification of the creation of a new instance of a mutable data type, while the *fire* operation illustrates the specification of the destruction of an existing instance of a mutable data type.

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 instance x). Mutating operations, such as *hire*, *fire*, and *raise* 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. A new object is guaranteed to be distinct from all objects defined in the previous state. The concepts *new* and *id* are not part of the Spec language, but they are provided by a predefined generic module *mutable* whose instances can be inherited by any mutable type. A definition of this 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 :: *y IN *t => id(y) = id(*y))
      --Every object has a permanent unique identifier.
  END
```

This is an example of a DEFINITION module, which defines only concepts. Such modules do not directly correspond to any components of a software system, and are used to group together concepts that are shared by many different parts of the system. The definitions in such a module can be included as a group via an INHERIT clause, as was done in the specification of the type *employee*, or they can be selectively included one at a time via IMPORT clauses. DEFINITION modules are also useful for recording the results of a conceptual modeling effort during requirements analysis in a form that can be used directly in the later functional specification and architectural design stages.

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.

2.4. Generators

A generator is a message that generates a sequence of values one at a time. Generators are sometimes also called "iterators" or "streams". An example of a specification for a generator is shown below.

```
FUNCTION Vendors
  IMPORT price FROM ic
  IMPORT sells FROM vendor

  MESSAGE(component: ic)
    GENERATE(s: sequence{vendor})
    WHERE ALL(v: vendor :: v IN s <=> sells(v, component)),
      increasing_price(s)
    CONCEPT increasing_price(s: sequence{nat})
      VALUE(b: boolean)
      WHERE b <=> ALL(i j: nat
        SUCH THAT 1 <= i < j <= length(s) :: price(s[i]) <= price(s[j]))
  END
```

This example shows a module that generates a sequence of suppliers for a specified integrated circuit, with the cheapest sources first. Such a module is a useful component for a decision support system for the managers of a computer manufacturing plant. The example also illustrates the use of IMPORT clauses for including concepts defined in other modules. Definitions of the types *ic* and *vendor* are not shown. The concept "increasing_price" has been defined explicitly to make the example self-contained. In a production context, this concept would be specified as an instance of the generic concept "sorted", which is part of the pre-defined generic type "sequence" in the specification library [7].

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 often 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.

Any message with a GENERATE is a generator, so that generators can be defined as operations of an abstract data type or a machine. Such operations support sequential scanning of the elements of an abstract collection without exposing the data structure used to implement the collection.

3. FEATURES FOR SPECIFYING LARGE SYSTEMS

The Spec language contains several features that are needed mostly for specifying large systems. Some of these features include generic modules, defined concepts, and an inheritance mechanism. An example illustrating the development of a complete system using Spec and a more detailed description of the language can be found in [7].

3.1. Generic modules

A parametrized module specifies a family of modules rather than an individual module. Generic modules are important for achieving re-use of specifications and designs because they can be adapted to a wider variety of applications than their more specific instances. 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 DEFINITION module *mutable{t}* given in the previous section is an example of a parametrized module. Such a definition defines one module for each legal set of values for the parameters of the module. The parameters can range over data values, functions, or types. Spec allows generic modules to have a variable number of parameters, and provides a means to define restrictions on legal values for the parameters.

3.2. Concepts

Concepts are used for explaining and testing the behavior of modules, and should be reflected in reference manuals and test oracles. A CONCEPT in the Spec language defines a constant symbol, predicate symbol, or function symbol that can be used in constructing the logical assertions defining the behavior of modules. 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 separating independent concerns.

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 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 FUNCTION module in Spec 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.

3.3. 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. Inheritance also provides a means for recording the boundaries between refinement steps in the design, and can be used to keep track of the distinction between the aspects of the system visible to the users and those visible only to the implementors. The inheritance mechanism and the rules for combining different versions of messages and concepts inherited from multiple parents are described in more detail in [4].

4. THE EVENT MODEL

The Spec language uses the event model to define the black-box behavior of software modules. The event model has been influenced by the actor model [20, 34]. The main differences from the actor model are the treatment of time and temporal events, and the treatment of multi-event transactions [2]. In the actor model, the behavior of a module is defined in terms of stimulus–response pairs, without atomic transactions involving longer chains of events, and all responses are assumed to be triggered by the arrival of messages rather than by points in local absolute time. 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. The event model is described informally below. A more formal definition can be found in [8].

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 stimuli it recognizes 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.

Messages can be used to model user commands and system responses. 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. A message has a condition, a name, and a sequence of zero or more data values. 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 triggering event is an implicit attribute of each message, used for identifying the destination for reply messages. Message transmission is assumed to be reliable, which means every message that is sent eventually arrives at its destination.

Each module has its own local clock and can send messages to itself at times determined by its local clock. The arrival of such a message is called a **temporal event**. The sending of the message corresponds to an **alarm** which enables the temporal event at a specified point in local time. The scheduling delay between the alarm and the corresponding temporal event can be constrained via a Spec DELAY expression. Temporal events allow modules to initiate actions as well as to respond to external stimuli. Temporal events are often periodic, and their phases may be defined relative to the time of day or the date.

Events at the same module happen one at a time, in a well-defined order. This order can be observed as a computation proceeds, and corresponds to the ordering of the local times at which those events occur. Events at different places need not have a well-defined order because the local clocks of different modules are not guaranteed to be synchronized with each other. The clock associated with a module measures local physical time, and in distributed systems simultaneous readings from the clocks of different modules have significantly different readings if the modules are at physical locations in different time zones. Thus time readings must be transformed to a standard frame of reference if they are to be used to determine orderings between physically separated events. Spec is based on a Newtonian model of time, since relativistic effects are negligibly small for current practical applications. However, practical measurements of message delays must account for clock synchronization errors if the source and destination of the message are at different locations, since such clocks can be synchronized only by sending messages with non-zero delays that are not completely predictable.

Orderings between events are not subject to clock synchronization errors if they are derivable from discrete sequences of the following types of steps:

(1) Two events at the same module are ordered by their local times.
(2) The event which triggered the sending of a message comes before the event in which that same message is received.

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 is a formalization of the requirement that each module must correspond to a coherent abstraction.

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. 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. Atomic transactions can be used to specify constraints on the order in which a module can accept events. Atomic transactions can be used to specify 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 familiar examples of such situations. Examples of atomic transactions defined in Spec can be found in [7].

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.

5. CONCLUSIONS

Spec is a specification language with a broad range of applications. The language is primarily intended for representing black box interface specifications in the early stages of design and in the maintenance phase. The language has a precise semantics and a simple underlying model. We have found the language to be sufficiently powerful for specifying 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 has been used by computer science students to develop moderately large programs using team projects and thesis work [12, 15]. The application of the language to the analysis, specification, design, construction, and evolution of an airline reservation system is described in [7]. We have found that black-box specifications are natural for complex systems and that they help the analyst in creating order out of chaos. The inheritance features were very useful for recording the refinement structure that lead to the current version for the system interface.

Students have learned the language using materials similar to those for introducing a new programming language [5], and with a comparable amount of effort. With some practice it is possible to produce short and clear specifications for even relatively complex software components. The language does force relatively complete analysis of the intended behavior of a system, and tends to expose incomplete understanding via long and complicated predicates. Such predicates are indications that the analysis is not yet complete, and that factoring and concept formation steps remain to be done. We have found effort for clarifying a specification to be well spent. After factoring and transforming specifications until they appear as collections of simple and independent constraints, we have found that implementations can be created quickly and with few errors, even for sizable pieces of code.

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, consistency checkers, design completion tools, test case generators, and prototype generators. A substantial subset of the language is executable in principle and work on the implementation of such a subset is in progress. However, the full language includes very powerful constructs which provide practical ease of expression at the expense of allowing the specification of some functions that are not algorithmically computable. The best that can be done is to provide partially correct implementations for the entire language which may fail to terminate in some cases where the specification is actually well defined.

REFERENCES

1. Backus, J. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Commun. ACM* **21**(8): 613–641; August 1978.
2. Berzins, V. and Gray, M. Analysis and design in MSG.84: Formalizing functional specifications. *IEEE Trans. Software Engng* **SE-11**(8): 657–670; August 1985.
3. Berzins, V., Gray, M. and Naumann, D. Abstraction-based software development. *Commun. ACM* **29**(5): 402–415; May 1986.

4. Berzins, V. and Luqi. The semantics of inheritance in Spec. Computer Science, Department Naval Postgraduate School. NPS 52-87-032; 1987.
5. Berzins, V. and Kopas, R. A student's guide to Spec. Computer Science Department, Naval Postgraduate School. NPS Tech. Rep. 052-89-028; 1989.
6. Berzins, V. and Luqi. Languages for specification, design and prototyping. In *Handbook of Computer-Aided Software Engineering*. Amsterdam: Van Nostrand Reinhold; 1989.
7. Berzins, V. and Luqi. *Software Engineering with Abstractions: An Integrated Approach to Software Development Using Ada*. Reading, MA: Addison-Wesley; 1990.
8. Berzins, V. and Luqi. An introduction to the specification language Spec. *IEEE Software*. To appear. Also Computer Science Department, Naval Postgraduate School. NPS 52-87-033.
9. Bjoerner, D. and Jones, C. *Formal Specification and Software Development*. Englewood Cliffs, NJ: Prentice-Hall; 1982.
10. Cohen, B., Harwood, W. T. and Jackson, M. I. *The Specification of Complex Systems*. Reading, MA: Addison-Wesley; 1986.
11. *Ada Programming Language*. American National Standards Institute/Mil-STD-1815A, DoD; 1983.
12. Douglas, B. A conceptual design of a design database for the computer aided prototyping system. M.S. thesis, Computer Science Department, Naval Postgraduate School; 1989.
13. Goguen, J. A., Thatcher, J. W., Wagner, E. G. and Wright, J. B. Abstract data types as initial algebras and the correctness of data representations. In *Proc. Conf. on Computer Graphics, Pattern Recognition, and Data Structures*, pp. 89–93; 1975.
14. Goldberg, A. and Robinson, D. *Smalltalk-80: The Language and its Implementation*. Reading, MA: Addison-Wesley; 1983.
15. Guenterberg, H. Case study on rapid software prototyping and automatic software generation: an inertial navigation system. M.S. thesis, Computer Science Department, Naval Postgraduate School; 1989.
16. Guttag, J. V., Horowitz, E. and Musser, D. R. Abstract data types and software validation. *Commun. ACM* **21**(12): 1048–1064; 1978.
17. Guttag, J. V. and Horning, J. J. A larch shared language handbook. *Sci. Comput. Prog.* **6**: 135–157; 1986.
18. Guttag, J. V. and Horning, J. J. Report on the Larch shared language. *Sci. Comput. Prog.* **6**: 103–134; 1986.
19. Herlihy, M. and Liskov, B. H. A value transmission method for abstract data types. *Trans. Prog. Lang. Syst.* **4**(4): 527–551; October 1982.
20. Hewitt, C. and Baker, H. Actors and continuous functionals. In *Formal Description of Programming Concepts*, pp. 367–387. New York: North-Holland; 1978.
21. Hoare, C. A. R. Proof of Correctness of data representations. *Acta Informatica* **1**(4): 271–281; 1972.
22. Johnson, L. Overview of the knowledge-based specification assistant. In *Proc. 2nd Annual RADC Knowledge-Based Assistant Conference*, RADC(COES), Grifiss AFB, New York; 1987.
23. Kraemer, B. SEGRAS–A formal language combining Petri nets and abstract data types for specifying distributed systems. In *Proc. 9th International Conference on Software Engineering*, pp. 116–125; March 1987.
24. Kraemer, B. and Schmidt, H. Types and modules for net specifications. In *Concurrency and Nets*, pp. 269–286. Berlin: Springer; 1987.
25. Liskov, B. H., Snyder, A., Atkinson, R. and Schaffert, J. C. *Abstraction Mechanisms in CLU*, Vol. 20; August 1977.
26. Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, J. and Scheifler, R. *CLU Reference Manual*. Berlin: Springer, 1981.
27. Luqi, Berzins, V. and Yeh, R. A prototyping language for real-time software. *IEEE Trans. Software Engng* **October**: 1409–1423; 1988.
28. Shaw, M., Wulf, W. A. and London, R. L. Abstraction and verification in ALPHARD: Defining and specifying iteration and generators. *Commun. ACM* **20**(8): 553–564; August 1977.
29. Shaw, M. *Alphard: Form and Content*. Berlin: Springer; 1981.
30. Tanik, M. In search of silver bullet. In *Proc. IEEE/ACM Fall Joint Computer Conference*, Dallas, TX, p. 686; November 1987.
31. Turski, W. and Maibaum, T. *The Specification of Computer Programs*. Reading, MA: Addison-Wesley; 1987.
32. Tyszberowicz, S. and Yehudai, A. OBSERV: Object-oriented specification, execution and rapid verification system. In *3rd Israeli Conf. on Computer Systems and Software Engineering*, Tel-Aviv, Israel; June 1988.
33. Wulf, W. A., London, R. L. and Shaw, M. An introduction to the construction and verification of Alphard programs. *IEEE Trans. Software Engng* SE-**2**(4): 253–265; December 1976.
34. Yonezawa, A. Specification and verification techniques for parallel programs based on message passing semantics. Ph.D. thesis, MIT; 1977.
35. Yuasa, T. and Nakajima, R. IOTA: A modular programming system. *IEEE Trans. Software Engng* **SE-11**(2): 179–187, February 1985.

About the Author—VALDIS BERZINS is an Associate Professor of Computer Science at the Naval Postgraduate School. His research interests include software engineering and computer-aided design. His recent work includes papers on software merging, specification languages, VLSI design, and engineering databases. He received B.S., M.S., E.E., and Ph.D degrees from MIT, served as an Assistant Professor at the University of Texas, and as an Associate Professor at the University of Minnesota. He has developed a number of specification languages and software tools.