The following paper was originally published in the
Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)
Santa Fe, New Mexico, April 27-30, 1998

IBDL: A Language for Interface Behavior Specification and Testing

Sreenivasa Viswanadha and Deepak Kapur
State University of New York, Albany

IBDL: A Language for Interface Behavior Specification and Testing

Sreenivasa Viswanadha
Department of Computer Science
State University of New York
Albany, NY, 12222
sreeni@cs.albany.edu, http://www.cs.albany.edu/ sreeni
Deepak Kapur
Department of Computer Science
State University of New York
Albany, NY, 12222
kapur@cs.albany.edu, http://www.cs.albany.edu/ kapur

Abstract

A methodology and language for specifying behaviors of interfaces (a la OMG's IDL, Java™, C++, etc.) for object-oriented systems is proposed based on the message-passing paradigm. Signatures of messages are enhanced to include semantic information, expressing behavior clients can expect from a server. Formulas are given to disambiguate normal termination from abnormal termination of a message using the return values and exceptions to reflect whether the pre-condition associated with the message is satisfied or not. State changes caused by a message invocation are specified by explicitly enumerating subsequent messages that a message invocation enables (and/or) disables, by establishing (or violating, respectively) their pre-conditions. Special operators on sequences of messages are defined to specify such semantic information. A specification language IBDL, Interface Behavior Description Language, based on this methodology is developed. As IBDL specifications explicitly capture the interactions between messages, they are ideal for validating implementation behaviors with sequences of messages. We present a scheme for sequence testing by translating IBDL specifications into code.

1 Introduction

An interface is a description of services offered by a system to external clients. Interfaces have been in use for a long time, starting with C standard header files that describe the services (system functions) provided by the C standard library. As object-oriented systems are becoming popular, interface design has become an integral part of system design since every object has an interface. An interface defines a type, and objects of this type can be created and used just like objects of other types. In this sense, interfaces have the same status as types, especially classes in systems such as JAVA and C++ [4, 15]. This view of interfaces allows hiding implementations of objects from their use, providing restricted hooks to clients for their use. For example, abstract classes of C++ can be considered as interfaces. Even though interfaces have been closely tied to their respective implementation languages, in recent years, interface languages have been developed [3, 6] independent of programming languages, to facilitate language interoperability.

The basic structure of an interface has remained largely unchanged: an interface is simply a set of function (method or message) declarations or signatures. A signature specification consists of specifying the argument types, return types and exception types for each message. (In more modern languages such as ISL [6], additional attributes for messages like *asynchronous* can also be specified to indicate that clients need not wait for those messages to finish and return.) A signature specification thus gives the typing rules for the messages with very little semantic information. Because of this, interfaces are used mainly to facilitate compiler type-checking in implementation languages.

A main advantage of interfaces is that they can facilitate language interoperability. For this purpose, it is inadequate for interfaces to merely support message declarations, which specifies types of arguments and results. Clients and servers also need to understand the behavior of the underlying system (or object) being interfaced. At the same time, it is desirable that interfaces *not* expose state information to the clients. Because of these restrictions, it becomes difficult for clients to infer anything about the behaviors of interfaces.

This paper proposes a methodology and a language IBDL, *Interface Behavior Description Language* for extending interface signatures to include semantic information using the message passing framework. Client programs send messages to objects via their interfaces and expect results back. Any message of the interface can be sent at any time if a handle to the object is available. But sending a message sometimes may not be appropriate because of the messages previously received by the server (in the case of multiple client/single server). For this purpose, we identify two distinct types of behavior of a message - *normal* and *abnormal*. Abnormal behavior is further distinguished using exceptions.

One way for clients to know the behavior of the underlying object is to understand the interactions between various messages. In the proposed methodology, this interaction is specified in terms of normal termination of subsequent invocations of messages by enumerating messages *enabled* and/or *disabled* by a given message invocation. The next step in associating the semantic information is also to restrict return values from a client by specifying the relationship between the return values, and input parameters as well as with input of the previous relevant messages. Special primitives on sequences of messages are supported in IBDL to extract information about earlier messages relevant to the behavior of the message under consideration. Information about the state of the object is thus accessed through these constructs in a disciplined and restricted manner.

A main advantage of the proposed methodology is that semantic information can be included in an incremental fashion based on need and resources available for developing behavior specification.

- **enables/disables** specifications that define interactions between messages,
- **interpretations** specification of return values for value-returning messages along with **enables/disables** clauses.

As evidenced by examples described in the next section, it is often easier to specify the interactions among messages in terms of **enables/disables** than to develop complete specifications including **interpretations**. We believe this flexibility will encourage practitioners to write specifications along with signatures as they design interfaces. We have deliberately kept the language simple (and perhaps less expressive) to facilitate this. In particular, our methodology is very well-suited for *shallow* objects like GUI components. We are aware of its limitations, particularly that the simplicity of the language makes difficult and awkward to specify arbitrary collection type of objects like stacks and queues.

One of the important benefits writing IBDL specifications is that implementations can be tested against these, especially using sequence-testing. In IBDL one explicitly specifies how the completion of a message affects the behavior of the subsequent messages. In a sequence test, messages can be sent one after the other and checks can be made to ensure that the behavior of the implementation satisfies the specification after the completion of every message in the sequence. This provides a sound basis for validating sequence tests, unlike the traditional way of manually inspecting test results for determining success or failure of a test. In fact, we came up with the proposed methodology while working on extending ADL/C++[17] for specifications for sequence-testing; ADL/C++ is a a specification language developed at Sun Microsystems Laboratories for unit-testing of C++ programs.

Another benefit of this methodology is that specifications can be developed in cases where clients do not have access to state information, especially in a multi-client environment. For example, a check clearing house usually does not have access to the balance in a bank account. In these cases the main concern for the clients is to understand the result of sending a message to the server, and have an idea about what messages can be sent next that will exhibit their expected behavior.

The paper is organized as follows. Section 2 informally introduces the methodology and the language by three examples of increasing complexity. Section 3 is a detailed discussion of the methodology and IBDL with intuitive semantics. In section 4, we describe the scheme for validating sequence tests. In section 5, we discuss some related work. Section 6 concludes with a brief discussion of ideas for future work.

2 Motivating Key Ideas using Examples

In interface declarations, the intention usually is not to expose the state. This can make it difficult to write specifications using pre- and post-conditions (see the bank account example below). However, one can get a fair idea about the state of the object looking at the return values of observers. The main idea is to partition the messages of an interface into *update* and *observer*[1] operations, and specify how an update message sent will change the behavior of other update and observer operations without explicitly using the state. This includes specifying if subsequent messages will behave normally or abnormally and what the return values, if any, would be.

Below by a series of examples, we will illustrate our methodology for writing interfaces and their behavior specifications. We also introduce (most of) the new constructs of the language one by one. The language IBDL is introduced fully in the next section with informal se-

[1]Some operations can be both updates as well as observers.

mantics.

We first introduce the *enables* and *disables* constructs along with simple interface declarations.

Consider the following example of a simple read-write object interface :

```
interface ReadWrite {
   void Write(int x);
   int Read();
   ReadWrite();
};
```

Here `Write` is the update message and `Read` is the only observer. `ReadWrite` is the constructor that creates objects of this type. If the state was exposed as a part of the interface, then it is easy to write the post-conditions for these two methods. Without access to state, however, a post-condition cannot be given.

Our approach to specification in these situations is to define sequences of messages, classify terminations of messages and specify the return values of the observers based on these sequences.

For the above example, this can be done as follows : The `Read` message should not be invoked until something is written using the `Write` operation, assuming initially the value stored is undefined. Similarly once a message `Write(k)` is sent for some integer `k`, subsequently a `Read()` message will succeed and its return value will be `k` until another *Write* message is sent. Here there is no explicit use of state in the specification, even though it can be observed using the observer operation `Read`.

This can be written formally in IBDL as follows. The complete syntax for IBDL is in the appendix.

```
specification ReadWrite {
   interface ReadWrite semantics {
      ReadWrite() { normal enables Write(x); }
      void Write(int i) {
         normal enables Read();
                interpretations   Read() = i;
      }
   }
}
```

Sometimes it is not possible to give complete specifications just by relating an update operation and its parameter values to subsequent messages. It is required to look at the behavior of the messages before the call to the update is made. We will introduce the "`@`" and the *enabled* operators here that can be used for this purpose. The "`@`"operator is used to access the state prior to the message and the *enabled* operator is used to check the enabledness of a particular message without actually sending it.

Consider the following bank account interface :

```
interface Account {
   void    Deposit(int x);
   boolean ClearCheck(int x)
         raises InvalidAmount, NotEnoughFunds;
   Account(int acNum, int initialBalance);
};
```

The objective here is not to allow clients sending the `ClearCheck` message to access the balance. In practice also, it is typical for clearing houses not to have access to the balance when they try to clear a check. At the same time, we want to be able to specify that if there is not enough balance in the account, the `ClearCheck` operation is going to fail. The methodology is particularly geared towards these situations.

An IBDL specification for this interface is :

```
specification Account {
   interface Account semantics {
      Account(int acNum, int initBalance) {
         normal enables
            Deposit(x) if (x >= 0);
            ClearCheck(x) if (x >= 0 and
                                x <= initBalance);
      }

      void Deposit(int amount) {
         normal enables
            ClearCheck(amount + y)
                      if @enabled(ClearCheck(y));
      }

      boolean ClearCheck(int amount) {
         normal disables ClearCheck(z)
                    if @(not enabled(
                         ClearCheck(z + amount)));
         abnormal defined by
                      raised(InvalidAmount) or
                           raised(NotEnoughFunds)
                     (amount < 0)
                          if raised(InvalidAmount)
      }
   }
}
```

The informal semantics of this interface can be given by the following rules :

1. After creating a new object using the message `Account(a, ib)`, a message `Deposit(k)` for any integer $k \geq 0$ or `ClearCheck(l)` for any integer l such that $l \geq 0$ and $l \leq ib$ can be sent to that object.

2. A message `Deposit(k)` with `k` ≥ 0 will always succeed because it is enabled immediately after the constructor and does not appear in a *disables* clause of any message.

3. After a `Deposit(k)` message to an `Account` object returns normally, a message `ClearCheck(l + k)` would succeed by returning `true` if a `ClearCheck(l)` would have succeeded prior to the `Deposit(k)` message. Also, any message that would terminate normally prior to the `Deposit(k)`, would remain so.

4. If a message `ClearCheck(k)` terminates normally, a subsequent `ClearCheck(l)` would fail if `ClearCheck(l + k)` would have failed prior to it and all other messages would succeed only if they would do so prior to the `ClearCheck(k)` message. Otherwise, it should raise one of the two listed exceptions and if the `InvalidAmount` exception is raised, then the value of *amount* should be less than 0.

In rules 2 and 3, the specification of the behavior of `ClearCheck` depends on its behavior prior to a `Deposit` or a `ClearCheck` respectively using the "`@`" operator to denote the state before the message.

A state-based specification for this interface would require the use of balance explicitly to specify the post-condition. One such specification in ADL/C++ using the state variable `long balance` of a C++ implementation of this interface is given in[17]. The reader will notice the dependence of the post-condition specification on the state variable `balance`, whereas our methodology uses only the messages of the interface and their parameters; it does not require the state information for writing specifications.

The above specification defines all possible sequences by specifying all the extensions of a given prefix that will guarantee normal behavior, assuming initially only the constructor is enabled. In that sense this is a constructive definition of all the normal traces (sequences of messages) for an object. As motioned earlier in the related work section, this is not the case with other trace specification methodologies such as in [8]. In these, one can only check for the validity of a given sequence of messages and value(s) returned, but cannot generate (in a straightforward manner) legal sequences.

Even the ability to look at the behavior of observers prior to an update message is not enough to describe the behavior fully in some cases. It may also be necessary to look at a message that was sent much earlier and its parameter values. For this, two new operators on sequences - *param* - to access the parameters of a particular message sent earlier and "#" - to count the number of times a particular kind of message sent, are introduced.

For example, consider an IBDL specification of an unbounded queue of integers :

```
specification Queue {
   interface Queue semantics {
      Queue() {
         normal enables Enqueue(x);
      }

      void Enqueue(int elem) {
         normal
           enables Dequeue();
           interpretations
             Dequeue() = param(#(Dequeue)+1,
                                          Enqueue, elem);
      }

      int Dequeue() {
         normal
           disables
             Dequeue() if (#(Dequeue) =
                           #(Enqueue));
           interpretations
             Dequeue() = param(#(Dequeue) + 1,
                                       Enqueue, elem);
      }
   }
}
```

All the operations have their usual meanings and `Queue` is the constructor for queue objects. Following is the intuitive meaning of the above specification :

1. When a queue is created using the `Queue()` message, only a message `Enqueue(k)`, for any integer `k`, can be sent to the created Queue object.

2. After a message `Enqueue(k)`, a message `Dequeue()` will terminate normally. Its return value (the next element in the Queue) is the parameter to the `i` th occurrence of the `Enqueue` message if there are *i - 1* occurrences of normal-terminating calls to `Dequeue()` so far (before this `Enqueue` message).

3. Similarly, after the message `Dequeue()` is sent, a subsequent message to `Dequeue()` will succeed only if there are more `Enqueue` messages than `Dequeue` messages prior to that. Once again, the return value of an enabled `Dequeue` message is going to be determined same way as in the previous step.

The "`#`" operator counts the number of times a particular message is sent and terminated normally since the time of creation of the object (including this message). The **param** operator returns the value of a certain parameter to a message given a message name and the number of occurrence in the sequence of messages prior to this one. This all abnormal messages are ignored for counting purposes.

As the above example shows, sometimes it is also necessary to know how many times a particular message is sent and its corresponding parameters for giving a complete specification.

This particular example can be much simplified if we allow trace simplification rules, *i.e.*, equations to specify that a `dequeue` message will cancel the first `Enqueue` message, immediately following the constructor, in the current sequence giving a new trace without either being present. This will be similar to incorporating some algebraic axioms as simplification rules into the language. We are currently investigating such an extension to the language.

3 Specification Methodology and IBDL

We have informally attempted to give the flavor of the language and the methodology using a series of examples in the previous section. We now give more details of the language in a systematic manner. All the features of the language are explained with informal semantics. A formal denotational semantics will be presented in a separate paper.

In order to use an interface effectively and correctly, it is necessary to understand interactions among different messages supported by an interface. As examples in the previous section demonstrate, message signatures (or declarations) of an interface need to be enhanced with behavior specifications. We discuss the methodology and a specification language - *Interface Behavior Description Language* (IBDL), designed to support this methodology.

3.1 Hierarchical Specifications

This methodology supports hierarchical specification of interfaces and their behaviors as :

- Simple interfaces declarations (signatures),
- Interface declarations with enables/disables clauses that define interactions between messages,
- Specification of return values for value-returning messages along with enables/disables clauses.

Each higher level in this hierarchy allows more powerful specifications of behavior. Depending on the need, a particular level of specification can be chosen. For example, if the goal of specifications is to do sequence testing, then enables/disables specification (possibly along with some other state-based specification) can be used. Similarly, if the goal is to do reasoning, the return value specifications and/or trace simplification rules may also be needed.

Specifications can be developed in a top-down fashion for new systems as well as in a bottom-up manner for existing systems using this methodology.

3.2 The Model

In our model, every object has an associated sequence of messages sent to it since its creation. A handle to an object can be obtained by clients using the constructors specified in the interface and the only interactions between a client and an object are through the messages provided in the interface. The first message in every sequence is a successful constructor message that defines (a handle to) the object to which all the subsequent messages are sent.

Every message has a name and a value for each of its parameters. We also assume that given a sequence of messages (to an object), one can count the number of occurrences of messages with a given message name and can access the parameter values of a message in a sequence.

Throughout this section we assume that T is the sequence of messages sent so far to the object, with the first message being a constructor message that uniquely defines the object.

3.3 Modules and Declarations

The first step in the system design is to come up with the desired interfaces, what services need to be offered to clients and the message names for those services.

In IBDL, a system can be organized as a set of named modules. A module consists of a set of named interface and exception declarations. The messages of these interfaces can optionally be annotated with behavior specifications. An IBDL module can import other modules using the **with** clause. Declarations in a module can use all the interfaces and exceptions declared in the imported modules.

3.4 Interfaces

An interface has a name and consists of a set of message declarations and one or more constructor declarations. We do not allow (state) variable declarations[2] at present because the idea is to support specifications in the absence of state information.

An interface defines a type, a set of messages (services) that the object can handle and constructors for creating new objects of the type. Message declarations can be overloaded.

The `ReadWrite` interface described in section 2 has two messages `Read()`, `Write(int i)` and a constructor `ReadWrite()`. An object created using the constructor `ReadWrite()` is of `ReadWrite` type.

3.5 Exceptions

Exceptions are named records of basic values. They can be used only by messages to signal exceptional conditions. The record fields (members) can be used to give more information about the error that caused it. It is desirable to name an exception in such a way as it reflects the kind of error that it is supposed to indicate.

In specifications, checks can be made to see the presence of exceptions using the *raised* expression. A *raised(e)*

[2]These can be easily emulated using *set/get* methods, but we don't encourage that.

expression evaluates to true after a message m iff m terminates raising the exception e. Expressions can use values of the data members of an exception for writing specifications.

3.6 Types

The primitive types available are the usual `int`, `boolean`, `char` and `String` types. A message that does not return a value has `void` as the return type.

3.7 Specifications for Messages

A message scheme $m(x_1, ..., x_n)$ where $x_1, \cdots, x_n, n \geq 0$ are formal parameters of a message name m associated with an interface is specified in the following steps :

Type Signature

Firstly, the name of the message, the types and names of parameters and the return value, if any, are specified. Parameters can be of any type except **void**. Each parameter in a message declaration has a distinct name.

Exception Specifications

Names of exceptions possibly raised by the message in case of errors without returning any values, are specified. An exception is raised to indicate to the client that the (intended) pre-condition for that message is violated. Different exceptions signal different ways a pre-condition is violated. We do not specify other exceptions that may be thrown by an implementation, *e.g.*, errors due to running out of memory etc. We also assume that after a message, *at most* one exception can be raised.

Termination Classification

Classify the termination of a message by defining the predicates

- *normal* which is true, if the message terminates without raising any exception and possibly returning a value.
- *abnormal* , which is false, if the message terminates raising an exception.

normal should always be the negation of *abnormal.* If a message scheme does not raise any exception, then *abnormal* defaults to *false* and *normal* defaults to *true.*
The IBDL syntax for specifying this is :

(abnormal | normal) defined by *expression*

The **defined by** clause is optional. If it is not given, then *abnormal* defaults to

$$raised(e_1) \ \ or \ \ raised(e_2) \ \ or \ \ ... \ \ or \ \ raised(e_k)$$

where $e_1, ..., e_k$ are different exceptions specified in the message scheme. In that case, *normal* defaults to *not(abnormal).*
The message can affect the behavior of different sets of messages depending on whether it terminates normally or abnormally. The subsequent specification (of effects of the message) have to be labeled as *normal* or *abnormal.* In the current language we don't allow updates in case of abnormal termination, but we might relax this later if we find the need for it.

Effect of a Message

It is specified how the given message affects the behavior of subsequent messages. Firstly, subsequent messages enabled, *i.e.*, will terminate normally, and disabled disabled, *i.e.*, will terminate abnormally, if invoked, are specified.
This is specified in IBDL using the *enables* and *disables* clauses with formulas of the form

$$(\textbf{enables} \mid \textbf{disables}) \quad g(y_1, ..., y_k) \qquad \text{if} \quad p(x_{i_1}, ..., x_{i_l}, y_1, ..., y_k)$$

where g is a message of the same interface[3], x_{i_j} is in $\{x_1, ..., x_n\}$, $0 \leq j \leq i_l$, $y_1, ..., y_k$, $k \geq 0$ are new variables, p is a predicate.
The above formula appearing in a *enables* (or *disables*) clause defines the set of all messages $g(d_1, ..., d_k)$ which are enabled (or disabled), after a particular invocation $m(c_1, ..., c_n)$, if the condition $p(c_{i_1}, ..., c_{i_l}, d_1, ..., d_k)$ evaluates to true.
A message that is neither in the *enables* nor in the *disables* set will continue to exhibit the same behavior as it did before the message $m(c_1, ..., c_n)$.
A specification for a message is not *well-formed* if it enables and disables a message at the same time.

Effect of a Message on Observers

The behavior of a given message is further captured by specifying the return values of observers affected by the updates caused by it.
These are given using the *interpretations* clause with formulas of the form

$$\textbf{interpretations} \ \ g(y_1, ..., y_k) = h(x_{i_1}, ..., x_{i_l}, y_1, ..., y_k) \quad \text{if} \quad p(x_{i_1}, ..., x_{i_l}, y_1, ..., y_k)$$

[3]This restriction may be relaxed in the future.

where h can be another observer of the same interface, an arithmetic, relational or logical operator or one of the operators on sequences of messages discussed below.

The meaning of the above formula is that, after an invocation $m(c_1, ..., c_n)$, the return value of the message $g(d_1, ..., d_k)$ will be equal to the value of the expression $h(c_{i_1}, ..., c_{i_l}, d_1, ..., d_k)$ evaluated after the message $m(c_1, ..., c_n)$ terminates normally and $p(c_{i_1}, ..., c_{i_l}, d_1, ..., d_k)$ evaluates to true.

If the return value of an observer is not in the set of *interpretations* defined, it returns the same value as it before the invocation of $m(c_1, ..., c_n)$.

A specification for a message is not *well-formed* if there is an enabled message whose interpretation is not defined.

Observers

Specifications for observers have the *defined by* clause, as they don't affect the behavior of any subsequent message. For example, in the `ReadWrite` interface, the *Read* operation can be called any number of times without affecting the behavior of either the `Read` or the `Write` message. Return value specifications of an observer are given in the *interpretations* clauses of the updates that can affect it.

Constructors

Borrowing the notation from C++ [15], constructors are messages with the same name as the interface, and they create new objects. For simplicity, we restrict the parameters to constructors to be of basic types.

Each constructor returns a (handle to a) unique new object that is different from all other objects in the system. A new object is assumed to exist *only if* the constructor terminates normally, *i.e.*, without raising any exception. The abnormal clause of a specification for a constructor cannot enable, disable or change the interpretation of any message. All the other components can be specified just like any other message.

Predicates and return values in the above are specified using the usual arithmetic, relational and logical operators. In addition, there are special IBDL operators on sequences. We define these below.

3.8 Special Operators

In the following, let m be the name of the message being specified, T be the sequence of messages sent to the object so far including m and S_1 be the sequence just before m is sent.

[3]Throughout these specifications, the (same) object handle is assumed to be implicit to all the messages in the interface.

- Message expressions - of the form $m(arg_1, ..., arg_n)$ denote values that the message m will return when called with actual parameter values of $arg_1, ..., arg_n$.
- The "`@`" operator - this specifies that the expression following the "`@`" should be evaluated using S_1 as the sequence for any operator(s) requiring a sequence.

  Consider the following sequence of messages to a `Queue` object

  ```
  Queue() Enqueue(10) Dequeue() Enqueue(5)
  ```

  The value of the expression `@`$enabled(Dequeue)$ is $false$ for this sequence, whereas the value of $enabled(Dequeue)$ is $true$.
- The "enabled" operator - takes a message and returns true iff extending the current sequence of messages by that message will return normally. A recursive definition of this operator can be given as follows.

  An expression *enabled(m)* , where m is a message, evaluates to true at T iff

  - m is in the *enables* clause of m, or
  - m is not in the *disables* clause of m and `@`*enabled(m)* is true.

  Consider the `Queue` example and the sequence

  ```
  Queue() Enqueue(10) Dequeue() Enqueue(5)
  ```

  Here, *enabled*$(Dequeue)$ is $true$, whereas `@`*enabled*$(Dequeue)$ is $false$.
- The "`#`" operator - takes a message name. It counts the number of messages in T with the given name that terminated normally.

  Consider the following sequence of messages to a `Queue` object.

  ```
  Queue() Dequeue() Enqueue(10)
  ```

  The value of the expression `#(Dequeue)` is 0 because the first `Dequeue` would terminate abnormally.

  On the other hand, consider the sequence

  ```
  Queue() Dequeue() Enqueue(10) Dequeue()
  ```

  Here `#(Dequeue)` evaluates to 1 and so does `#(Enqueue)`.
- The **param** operator - takes 3 arguments - the number of occurrence of the message name in the sequence T, a message name and a parameter name.

An expression *param(k, m, n)* returns the value of the parameter named n of the message with name m, in the shortest subsequence T' of T containing m, such that at T', the value of $\#(m)$ is k. Value of this expression is undefined if no such (nonempty) T' exists.

Consider the `Queue` example and the sequence

```
Queue() Dequeue() Enqueue(10) Enqueue(5)
```

At the end of the sequence, the expression *param(2, Enqueue, elem)* evaluates to 5.

The sequence parameter in the above discussed operators on sequences of messages is implicit. It is always either the current sequence, or the subsequence excluding the last message (when using the "`@`" operator).

4 Sequence Testing and Validation

One of the main design goals of IBDL is to provide formal support and automation to testing. Specifications written in IBDL can be compiled into code that can be used to validate tests. IBDL can support sequence testing very well, as a specification explicitly expresses the behaviors of subsequent messages after a message completion. One can send a sequence of messages one after the other to the implementation, and check for every message with respect to all the previous messages, if it is behaving as defined by the specification.

The sequence testing and validation system (Fig. 1) consist of

1. a test driver,
2. a validation class generated from an IBDL specification, and
3. a user-written implementation.

These components are discussed in the following subsections.

4.1 Test Driver

The test driver in this framework is somewhat different from a conventional test driver in that it not only performs the tests, but it also validates tests either as success or failure. This module has a handle to a validation object of the validation class type. It takes a sequence of messages as input and iterates over the sequence sending messages to the validation object (see below) for invocation and validation of the message using the implementation.

The top level loop of the test driver works as follows :

- Create a validation object with the required parameters to the constructor.
- Send the messages one by one to the validation objects. If there is any inconsistent behavior, the validation throws an error indicating message behavior not satisfying the specification. Stop testing this sequence any further, indicating test failure.
- If all the messages in the sequence get executed without throwing any error, then the sequence test succeeds otherwise, it fails.

4.2 Validation Class

This is a class generated from the IBDL specification of an interface. This class interfaces to the test driver as well as the implementation. Every object of a validation class has the generic part which implements the "`#`" and `param` operators of IBDL, and a specific part that has a handle to an actual implementation object and provides interface to the test driver so that messages can be received. In this sense, the validation class is a wrapper class for the implementation, which intercepts messages sent to and return values/exceptions from the implementation object for validation purposes.

The specific part of the validation class has code to receive messages, store the parameters and return values and/or exceptions and invoke the message validation code based on these. This class also includes the code generated from the behavior specification of each of the messages, as described in the following subsections.

The validation class has a constructor with the same signature (parameter types and exceptions) as the constructor specified in the interface. This constructor creates the object under test and stores a handle to it so that messages can be sent to the object using the handle. This is done to ensure that the first message received is a constructor as the IBDL semantics dictates.

`IBDLObject` Class

The IBDL `IBDLObject` class is a superclass of all validation classes (see below) that provides the implementation of the IBDL operators "`#`" and `param` which are common to the translation of any IBDL specification. The implementation of these methods is fairly straightforward and follows the informal semantics described in section 3.

It includes the `record` method that records a message in the history only if the message terminated normally.

Validation Code for a Message Declaration

Every message declaration with a behavior description is translated into a code which has a generic part that does

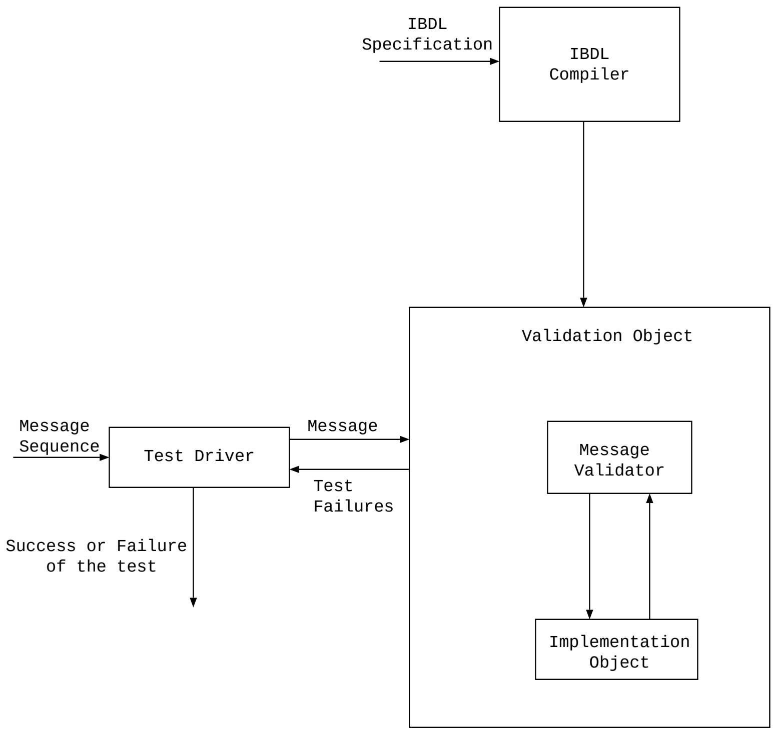

Figure 1: Sequence Testing and Validation System

the general book-keeping of storing parameters, return values and/or exceptions and a specific part that implements the behavior specification. Below, we discuss the specific part related to implementing the behavior specification.

The specific part has 3 methods (or functions) - `enabled`, `interpretation`, `isNormal` corresponding to the *enables/disables*, *interpretations* and the *defined by* clauses, respectively, of a specification. These methods are for validating subsequent messages, and they are defined below :

- `isNormal` returns `true` if the predicate for *normal* given in the specification evaluates to `true` and `false` if the predicate for *abnormal* evaluates to `true`. It throws an error if neither *normal* nor *abnormal* predicate evaluates to `true`. Note that this error is different from the exceptions that can be raised by a message sent to the interface.
- `enabled` takes a message name and a vector of parameters and returns true if the message formed using the name and parameter values is enabled subsequently to the current message as per the *enables/disables* clauses.
- `interpretation` takes a message name, a vector of parameters, and a return value and returns `true` only if that combination of message name, parameter values and return value satisfies the predicate given in the *interpretations* clauses of the specification for this message.

The specific part also has a `boolean` method `validate` that first checks if the message, given the history, behaves as expected. It raises an error, indicating a failure of the test if

- either it is `enabled` after the last message before this and `isNormal` returns `false`, or
- it is not `enabled` after the last message before this and the method `isNormal` returns `true`, or
- the value returned does not satisfy the predicate for the `interpretation` when evaluated in the previous history.

The IBDL expressions used in the predicates are translated accordingly so that the above methods can be evaluated. A part of the generated class for the `Account` interface as specified in Section 2 is given in Fig. 2.

Validation of a Message

When the validator class receives a message from the test driver, it validates it as follows :

- Store the parameters.

```
class IBDLAccount extends IBDLObject
                  implements Account
{
   /* The object under test.  */
   final Account accountObj;

   /* The constructor of this class.  */
   IBDLAccount(int acNum, int initialBalance)
   {
      MessageSend ms =
        new AccountMsg(acNum, initialBalance);
      record(ms); ms.Validate();
   }
   /* Class that represents a call to the
      ClearCheck method.  */
   class ClearCheckMsg extends MessageSend {
      ClearCheckMsg(int amount)
          throws InvalidAmount, NotEnoughFunds
      {
         super("ClearCheck", 1);
         AddParam(0, amount);
         prev = lastMessage;
         lastMessage = this;

         try { /* Call the method.  */
            StoreReturnValue(
               accountObj.ClearCheck(amount));
         } catch (InvalidAmount t1) {
            StoreException(t1);
         }
         catch (NotEnoughFunds t2) {
            StoreException(t2);
         }
         catch (Throwable t) {
            StoreException(t);
         }
      }

      public boolean IsNormal() {
         if ((exception instanceof
                           InvalidAmount ||
              exception instanceof
                           NotEnoughFunds)) {
            if (exception instanceof
                           InvalidAmount) {
               if (((Integer)GetParam(0)).
                               intValue() < 0)
                  return false;
               throw new Error("Test failed."+
                " The definition of abnormal" +
                 " is not satisfied");
            }
            return false;
         }
         return true;
      }
      public boolean Enabled(String name,
                             Object[] params)
      {
         if (IsNormal()) {
            switch(messageNames.indexOf(name))
            {
               case 2 :  // ClearCheck
                  int i = ((Integer)params[0]).
                                    intValue();
                  Object[] newParams = {
                      new Integer(i +
                          ((Integer)GetParam(0)).
                                  intValue()), };
                  if (!prev.Enabled(name,
                                  newParams))
                     return false;
                  break;
            }
         }
         return prev.Enabled(name, params);
      }
   }
   /* Method to call the ClearCheck method
      on the Account object.  */
   public boolean ClearCheck(int amount)
      throws InvalidAmount, NotEnoughFunds
   {
      MessageSend ms =
               new ClearCheckMsg(amount);
      record(ms); ms.Validate();
      if (ms.exceptionThrown) {
         throw ms.exception;
      }
      return ((Boolean)ms.retVal).
                       booleanValue();
   }
   /* Code for Deposit and the constructor
      will be similar */
}
```

Figure 2: Class generated for the `Account` Interface Specification

- Send the message to the implementation object and get the return value or exception.

- Using the stored parameters and the return value or exception, evaluate the `validate` predicate. If it throws an error, then return indicating a test failure.

- Evaluate the `isNormal` predicate of this message and if it returns `true`, then append this message to the history.

We are currently implementing the scheme as described above and we expect to have the beta version ready sometime in April 1998.

4.3 Language-Independent Testing

Testing implementation using IBDL can be done in an implementation language-independent fashion. But it will be dependent on the interface language binding. For example, if we choose IDL [3] as the interface language then the validation module can be generated to conform to the binding of a specific implementation language, *e.g. Java*, and the interface can be generated in IDL. Then the implementation can use the generated interface to develop the implementation in any language for which a binding is defined, e.g., C++ and it will be (sequence) testable using IBDL[4].

5 Related Work

The proposed specification method is similar in spirit to the trace based specification approach of [8]. However, there are two crucial differences. Firstly, our approach is more structured in that semantics is associated with each message explicitly rather than merely giving rules to describe legality of traces. One of the problems with the approach in [8] is that exceptions cannot be specified. This is a severe restriction in practice as it is common for servers to raise exceptions when a message is received whose pre-condition is not satisfied in the current state, and subsequently, continue with other messages. Clients that send such a message are in a position then to handle such exceptions raised by the server and continue. In the trace-based specification approach in [8], such behavior will not be allowed because a trace with a message whose pre-condition is not satisfied is considered illegal. In contrast, the proposed methodology captures this by characterizing the result of a message to be abnormal in case the pre-condition for a message is not satisfied, and allows both the client as well as the server to continue their respective operations, probably after taking some corrective measure.

[4]For this paper, we considered Java as the interface as well as the implementation language.

ObjLog [2] is a language based on trace model for object-oriented analysis and design. It does not use states explicitly for specification, but requires selector services that return values from the state to give specifications of updates. It uses transition equations to specify properties of updates. It is unclear how updates whose behavior is affected by other updates much before it (as in the queue example below) can be specified in ObjLog. It might be possible to use transition equations for that purpose, but we believe the resulting specifications will be difficult to develop as well as understand. In the proposed methodology, operators on sequences of messages are explicitly introduced for this purpose.

Borneo [12] is a language designed for low-level specification of message behaviors of OMG's IDL using the ADL framework [13]. The main problem with Borneo is that if the interface does not expose any state, the only way to specify the interface behavior is to use the auxiliary definitions, akin to coming up with a crude implementation. This blurs any distinction between a specification and an implementation, thus making the specification as vulnerable to having bugs as the implementation is. The proposed methodology does not suffer from this limitation. Further, the key ideas of our methodology can be easily adapted into the ADL family of languages and in fact, we are working on incorporating them into extending ADL/C++.

Languages based on Algebraic methodology have long been used for specifications. IBDL can be easily translated into any of those languages. But the purpose of IBDL is to give programmers/designers using interface languages, a simpler specification language with straightforward semantics, unlike algebraic specification languages which are usually based on equational theories. Algebraic languages, like Larch/CORBA [11] are quite powerful and general in that almost any computable function can be specified. IBDL on the other hand, is a less powerful language with a few reasonably easy-to-understand new concepts. As demonstrated by the ADL [13] project, despite the simplicity, these methodologies seem to be adequate for specifying many practical problems. Also, this simplicity and the familiar *look-and-feel* of the specification language seems to encourage practitioners to write specifications at least for some critical portions of their code.

IBDL also provides flexibility to a designer of incrementally specifying behavior at different levels of detail at different stages of interface design. As a start, only normal and abnormal clauses for messages and exception conditions, if any, raised by them can be specified. This can be followed by specifying enable and disable clauses once interaction among messages is analyzed. Subsequently, interpretations can be given once the design and the behavior of messages are finalized.

The other aspect of IBDL that is different from other methodologies is that states need not be explicitly modeled, instead observer methods give the ability to get properties of states. State however comes up in the form of history of messages received, but users never need explicitly specify the state transformation after a message; it is simply defined as extending the previous history by the current message. To see this, consider the LARCH/CORBA specification for a `PrinterQueue` interface as given in [11].

```
interface PrinterQueue {
    const int MAX_QUEUE_SIZE 20
    uses PrinterQueueTrait(PrinterQueue for PQ);
    initially self' = empty;
    void enqueue(in int id) raises (QUEUE_FULL) {
       requires true;
       modifies self;
       ensures if len(self^) = MAX_QUEUE_SIZE then
                  raise(QUEUE_FULL) /\ self' = self^
              else self' = append(self^, id);
    }
    int dequeue() {
       requires ~isEmpty(self^)
       modifies self;
       ensures
         self' = tail(self^) /\ result = head(self^);
    }
    int size() {
       ensures result = len(self^);
    }
}
```

There are two main differences between the above specification and an IBDL specification for a similar interface given in section 2. Firstly, there is (an abstract) representation of the object in terms of `self` (see page 11 of [11]). The specification for `enqueue` (and `dequeue`) is given as their effect on the object to say that the new value of `self` is same as appending (and removing the head of) the old value of `self`.

In contrast, in the IBDL specification, this behavior is done in terms of messages by specifying that a `dequeue` operation can be performed after an `enqueue`; if there have been equal number of `enqueue` and `dequeue` messages, then a `dequeue` cannot be performed. This second requirement is captured above in LARCH/CORBA by the pre-condition (`requires` clause) using the state of the queue object.

In algebraic methods, recursion is used extensively which practitioners often find difficult to understand. In IBDL, we support a very restricted form of recursion using the `@` operator and to some extent, the `#` and `param` operators. We believe this will make it easier to develop and understand specifications. We can certainly extend the language to include more operations on sequences to enhance its power, but we purposely chose not to do it to keep the semantics simple.

The other very important difference between IBDL and other languages mentioned is the capability to test implementations against specifications. We have come up with a scheme for this using IBDL and we are actively building a tool to support this scheme. To the best of our knowledge, there are no specification-based sequence testing/validation tools using any of the other languages discussed above.

6 Concluding Remarks And Future Work

We have described a specification methodology for interface behaviors based on a message-based paradigm. An important feature of the methodology is that specifications can be designed without needing access to state information. A behavior specification for a message is a post-condition that distinguishes between normal and abnormal termination as well as its effect on subsequent messages vis-a-vis their pre-condition being satisfied or not. This supports the explicit (extensional) specification of the normal traces of an interface. A simple specification language, IBDL, is defined embodying the main ideas of the methodology. Specifications at the client level using *normal/abnormal* as well as at the server level using the *enables/disables* constructs can be given. The use of this methodology for validating sequence testing of implementations is also developed.

Even though the specification method is quite powerful and expressive, specifying certain behaviors can be tedious. We are investigating the use of message sequence axioms to simplify message sequences. Consider an example of a `Stack` interface. With the proposed methodology, one has to write a complex formula on the prefix of a sequence just to specify that a `Pop` message is enabled only if the number of `Push` messages is greater than the number of `Pop` operations in the prefix. A simpler specification would be to specify that a `Pop` message will cancel the latest `Push` message in the prefix, thus simplifying a message sequence. Such a rule, combined with a specification for `Push` that it *enables* a `Pop`, can completely define the behavior of the `Stack` interface. We are developing conditions on such rules on message sequences for specifications so as to keep the specification simple.

The proposed methodology as well as IBDL do not support explicit specifications of inheritance, an important concept in object-oriented languages as well as in interface languages. Inheritance is also supported in many interface languages including [3]. We would like to ensure that some form of behavioral subtyping [9, 1] is achieved using the specifications as opposed to the syntactic subtyping that interfaces have. The expression language used in IBDL is restrictive in that it only allows specification of fully deterministic systems. But for top-down design, non-deterministic specifications and constructs may have to be supported. This may be useful for behavioral subtyping where the subtype has more specific (deterministic) behavior than the supertype. We have some preliminary ideas on this and are working on incorporating those into IBDL in such a way that it remains testable. We have worked out the formal denotational semantics of IBDL, which will be presented in a separate paper.

References

[1] America, P. Designing an Object-Oriented Programming Language with Behavioral Subtyping. *Foundations of Object-Oriented Languages, REX School/Workshop*, Noordwijkerhout, The Netherlands, Springer-Verlag Lec. Notes in Com. Sci. 489, 1991.

[2] Ted L. Briggs and John Werth. A Specification Language for Object-Oriented Analysis and Design. *Proceedings of the 8th European Conference on Object-Oriented Programming*, Bologna, Italy, July 1994.

[3] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design, Inc., and SunSoft , Inc. The Common Object Request Broker : Architecture and Specification. Pages 45-80. OMG Document Number 91.12.1, Revision 1.1. December 1991.

[4] James Gosling, Bill Joy, and Guy Steele. The Java$^{\mathrm{tm}}$ Language Specification. Addison-Wesley, 1996.

[5] John V. Guttag and James J. Horning with S. J. Garland, K. D. Jones, A. Modet and J. M. Wing. Larch : Languages and Tools for Formal Specification. Springer-Verlag, 1993.

[6] Bill Janssen, Denis Severson and Mike Spreitzer. ILU 1.8 Reference Manual. Xerox Corporation, May 1995.

[7] D. Kapur and D. R. Musser. Tecton: A Framework for Specifying and Verifying Generic System Components. Rensselaer Polytechnic Institute Computer Science Technical Report 92-20, July, 1992.

[8] J. McLean. A Formal Method for the Abstract Specification of Software. J. ACM, vol. 31, no. 3, July 1984.

[9] Barbara Liskov and Jeannette M. Wing. A Behavioral Notion of Subtyping. *ACM TOPLAS* 16(6):1811-1841, Nov. 1994.

[10] OMG, CORBAservices: Common Object Services Specification, OMG Document Number 95-3-31, Object Management Group, Framingham, MA, (1995).

[11] Gowri Sankar Sivaprasad. Larch/CORBA: Specifying the Behavior of CORBA-IDL Interfaces. Department of Computer Science, Iowa State University, TR #95-27a, December 1995, revised December 1995.

[12] Sriram Sankar. Introducing Formal Methods To Software Engineers Through OMG's CORBA Environment And Interface Definition Language. In *Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology*, Munich, Germany, July 1996.

[13] Sriram Sankar and Roger Hayes. ADL: An Interface Language for Specifying and Testing Software. In *Proceedings of the Workshop on Interface Definition Languages*, January 1994.

[14] J. M. Spivey. Understanding Z, A Specification Language and its Formal Semantics. Cambridge University Press, 1988. Tracts in Theoretical Computer Science, Volume 3.

[15] Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 1991.

[16] Sun Microsystems Inc., U.S.A., and Information Technology Promotion Agency, Japan. ADL Translator Design Specification. Document number MITI/0001/D/0.1. August 1993.

[17] Sreenivasa Rao Viswanadha and Sriram Sankar. Preliminary Design of ADL/C++ - A Specification Language for C++. In *2nd USENIX Conference on Object-Oriented Technologies*, Toronto, Canada, June 1996.

A Appendix

Following is BNF syntax for IBDL. Keywords are given boldface. { ... } represents zero or more elements and optional elements are given within [...] .

```
specification_module ::=
     specification module_name “{”
          { interface_declaration | exception_declaration }
     “}”

interface_declaration ::=
     interface interface_name semantics “{”
          { message_declaration }
     “}”

exception_declaration ::=
     exception exception_name “{”
          { variable_declaration “;” }
     “}”

variable_declaration ::= type_name variable_name

type_name ::=
     boolean | char | int | String | void | interface_name

message_declaration ::=
     type_name message_name
     “(” [ param_list ] “)”
     [ exception_specification ]
     { “{” behavior_specification “}” | “;” }

param_list ::=
     variable_declaration { “,” variable_declaration }

exception_specification ::= raises “(” exception_list “)”

exception_list ::=
     exception_name { “,” exception_name }

behavior_specification ::=
     { ( normal | abnormal )
     [ defined by expression ]
     { ( enables_clause | disables_clause ) }
     [ interpretations ]
     }

enables_clause ::=
     enables { message [ if expression ] }

disables_clause ::=
     disables { message [ if expression ] }

interpretations ::=
     interpretations { message “=” expression
                                   [ if expression ] }

message ::=
     [ expression “.”] message_name
                 “(” [ actual_param_list ] “)”

actual_param_list ::= expression { “,” expression }

expression ::= message
                         | binary_expression
                         | unary_expression
                         | sequence_expression
                         | variable_name
                         | raised_expression

binary_expression ::=
        expression binary_op expression

binary_op ::=
         “.” | “+” | “-” | “/” | “<” | “>” | “<=” | “>=”
       | and | or

unary_expression ::= unary_op expression

unary_op ::= “-” | not | “@’’

sequence_expression ::=
      enabled ”(” message ”)”
    | “#” “(” message_name “)”
    | param “(” expression “,” expression “,” expression “)”

raised_expression ::= raised ”(” exception_name ”)”
```