ADL/C++ — A Specification Language for C++ and its Implementation

Sreenivasa Viswanadha*
Department of Computer Science
State University of New York
1400 Washington Ave.
Albany, NY 12222, USA
sreeni@cs.albany.edu

Deepak Kapur
Department of Computer Science
State University of New York
1400 Washington Ave.
Albany, NY 1222, USA
kapur@cs.albany.edu

Abstract

A design and implementation of ADL/C++, a specification language for associating behavior specifications for C++ (function) declarations, are developed. Based on the ADL, Assertion Definition Language, framework developed at Sun Microsystems Laboratories, a side-effect free subset of the C++ expression language is used for writing post-conditions as boolean expressions. The specification language also includes additional primitives to facilitate the specification of exceptional as well as normal behavior of functions. Most of the important features of C++ including member functions, virtual members, constructors of classes, inheritance and exceptions can be specified with relative ease. Behavior specifications are given a first-class status allowing their reuse. The syntax of ADL/C++ closely resembles the syntax of C++. The semantics of ADL/C++ is deliberately kept to be informal, pretty much in the style of the C++ reference manual. It is hoped that these design decisions would encourage programmers to use ADL/C++ for writing specifications as they develop C++ programs.

Specifications written in ADL/C++ can be used as semi-formal documentation for C++ programs. These specifications can also be used for testing C++ implementations, thus making the testing process to be automatic and less adhoc. A method for validating tests of C++ implementations using ADL/C++ specifications is discussed. A complete scheme for translating ADL/C++ specifications into C++ has been developed, which makes the automation of test validation process based on these specifications, possible.

1 Introduction

Traditionally specification languages have been designed to give a specification of a computation independent of the implementation language(s) in which the computation is realized. While this approach may have the advantage of giving an independent characterization of algorithms and data structures as well as system behavior, it imposes the burden on the users to learn two different languages using, possibly two different formalisms and two different paradigms (since a specification emphasizes *what* and an implementation emphasizes *how*). For example, the specification language Z [11] uses a set-theoretic semantics for specifications; Larch [4] and Tecton [5] are languages based on algebraic specification emphasizing algebraic equational axioms. Most industrial-strength programming languages including C and C++ are imperative, very different from such specification languages, and have, however, a state-based operational semantics. Because of this big gap between a specification language and a programming language, programmers perhaps do not feel inclined to develop specifications as they write programs.

A specification language ADL/C++ is proposed in this paper, for associating behavior specifications for C++ function declarations. The language is based on the ADL, Assertion Definition Language framework developed at Sun Microsystems Laboratories [8]. A family of languages have been designed using this framework, to associate semantic information with declarations of variety of languages including C and IDL. We have been particularly encouraged with the success ADL/C has had so far in popularizing such a specification methodology in the industry. ADL/C++ builds on the basic ideas of ADL/C to facilitate specifications for the more complex features of C++.

*This work is supported by funding from Sun Microsystems Laboratories.

The main objective in designing a specification language using the ADL framework is to keep its syntax and semantic as close as possible to those of a programming language. A subset of the programming language itself (with some extensions) is used to write specifications. This has the advantage that developers do not have to learn a different language possibly supporting a different paradigm for writing specifications. The main distinction emphasized between a program and a specification is that while writing a specification, the programmer is encouraged to specify *what* the program does without worrying about implementation details, and for this, a small "*side-effect-free*" subset of expressions supported in the language can be used.

Just like C and C++ programmers almost always write header files for declarations, it is hoped that such an approach would encourage programmers to write specification files along with header files before working on the implementations. Even though developing a complete specification that emphasizes what instead of how is a desirable goal, we believe that a methodology that facilitates writing specifications (even if they are partial) would be useful to practitioners in the industry. These specifications can serve as a semi-formal documentation, much beyond just capturing syntactic and typing requirements, as is the practice with header files,

An ADL/C++ specification is a collection of **semantic** clauses stating conditions for normal and exceptional termination of functions; boolean expressions are used for writing post-conditions, specifying state change, if any, as well as relationship between exceptional values as return values to the input parameters and state. Most of the important features of C++ including member functions, virtual members, constructors of classes, inheritance and exceptions can be specified. Behavior specifications are given a first-class status using the **behavior** primitive, thus encouraging their reuse. The syntax of ADL/C++ closely resembles the syntax of C++. The semantics of ADL/C++ is deliberately kept informal, pretty much in the style of the C++ reference manual.

Another important feature of ADL/C++ is its executability. Specifications are translated into code, which will make them useful for testing purposes. An implementation can be executed with an input data and the output generated along with the test input can be plugged into the specification to see whether the specification is satisfied for that particular test data. This makes the testing process to be automatic and less adhoc. A method for validating tests of C++ implementations using ADL/C++ specifications is discussed. A complete scheme for translating ADL/C++ specifications into C++ has been developed, which makes the automation of test validation process based on these specifications, possible.

The proposed approach of requiring a specification language to be, by and large, a subset of a programming language is related to those in high-level functional and logic programming languages where a single language framework is used for both specification and implementation.

Limitations

There are some inherent limitations with this approach. Particularly, the specification language will inherit all the pitfalls and ambiguities of the programming language depending upon the restricted subset of the language being used for specification purposes. Proper choice of a subset of the underlying programming language that is well-understood with minimal ambiguities becomes crucial. Also, some programs/implementation may serve as specifications also. We do not consider this as a limitation insofar as such programs emphasize the behavioral description (*what*) instead of getting into nitty-gritty implementation details (*how*).

Specifying class behaviors also poses problems with this approach. The model of ADL has been unit-testing of functions. This is inadequate for specifying abstract properties of C++ classes, as often this involves capturing interactions between.

This paper expands on our preliminary report [14] on this with more details and a full translation scheme.

The rest of the paper is organized as follows. In Section II, we present some background information on the ADL framework and the use of ADL/C. Section III presents the main ideas behind of ADL/C++ In Section IV, we discuss specifications for the important features of C++. Section V discusses the semantics for virtual functions. In Section VI, we describe how to reuse ADL/C++ specifications. In Section VII, a detailed description of the language syntax and informal semantics is given. Section VIII describes a scheme for translating ADL/C++ to C++. In Section IX, a method for validation of C++ implementations using ADL/C++ specifications is presented. Section X gives a comparison with other related work. We conclude in Section XI with a brief discussion about future work.

2 Assertion Definition Language (ADL)

ADL, Assertion Definition Language, is a specification language framework used to describe input-output behavior of software systems. ADL originated as part of a conformance testing system ADLT [13] for ANSI C API's. In this system, the runtime behaviors of implementations of API's are compared to their specification written in a C-like syntax as extensions to C header files. This particular syntax has been called ADL/C (or ADL for C) [8]. The ADLT project commenced in early 1993 and has culminated in a robust industry quality environment for conformance testing [8].

ADLT has a collection of tools developed for ADL/C and a language called TDD for describing test data. The tools include ACF, the assertion checking function generator, that translates post-conditions given in ADL/C into C functions, a "natural" language document generator, and a test coverage analysis tool. Some of these tools are being successfully used by X/Open and NIST as a part of their conformance testing efforts [10].

ADL/C++ builds on the experience gained from the ADLT project. It is an extension of ADL/C for C++, but it also represents a significant improvement in expressiveness and ease of use. A subset of ADL/C++ itself may be used to specify C programs, and this subset, it is anticipated, will eventually replace the old ADL/C[1] . Other projects currently being undertaken are ADL/IDL (OMG IDL [2]) and ADL/Java™.[2] While this paper focuses on ADL/C++, the general concepts presented here apply equally well to the other language specializations of ADL.

Systems can be developed either top-down or bottom-up style using ADL/C++ (and C++). In the top-down style, typically the interface (declarations of classes) is first defined and then the specification and implementation are both developed simultaneously from that. Thus, one can have semi-formal documentation and also validation tool ready as the system is being built. This is likely to help in finding problems early in the development stages without much overhead on the developers since this process can be automated.

ADL/C++ can also be used to specify legacy systems like software libraries that are in use for a long time. There is hardly any formal documentation about these systems other than the API. ADL specifications can be developed using these API's along with other informal documentation like manuals or man pages. One of the early uses of ADL/C was for specifying system I/O libraries. The advantage of using ADL/C for this purpose became clear when a bug was discovered in an implementation of an I/O library, when tested using the ADL/C specifications [9]. The *write* system function updated the modification time when writing 0 bytes to a local file, but if the file was a remote file, the modification time did not change.

3 An Overview of ADL/C++

ADL/C++ supports behavior specifications for global functions, member functions and constructors in the form of post-conditions. It also supports specification for inheritance and virtual functions based on the specifications for functions. Specifications for exceptions can be given to state conditions for an exception to be thrown and relating the exception value to parameter values.

For this purpose, it uses a *side-effect-free* subset of C++ expression language as described in [12] and extends it with the usual logical operators, bounded quantification, *call-state* operator (for evaluating expressions prior to a function call), *return* to denote return values of functions, and some structuring constructs. The purpose of an ADL/C++ specification is to describe *what* a function does and not *how* it is done.

The core of an ADL/C++ specification is the set of **semantics** clauses that describe the post-conditions for functions in the form of annotations for declarations. A post-condition is a set of boolean formulas to specify state transformations and the relationship between input parameters, return values and exception values (in a given state). This is done in 3 parts.

- Classification of return values and exceptions into *normal* and *abnormal*, in relation to the parameter values and the system state. By default, the presence of an exception after a call is considered abnormal behavior and the absence of any exception along with the required properties of the return value is considered normal behavior.

[1] It is one of the main goals of the design of ADL/C++ to maintain upward compatibility so that the existing ADL/C specifications can work with the new set of tools.

[2] Java™ is a trademark of Sun Microsystems Inc.

- Predicates for relating parameter values to return and exception values.
- Specification of state transitions. Some of these are specific to each of the normal and abnormal behaviors and others are common to both.

A C++ function declaration allows a function to be called with any parameter value(s) as long as the type rules are obeyed. But in practice, many functions are partial and the type rules are not strong enough to specify that. For example, a simple function such as *int Sqrt(int x)* that computes the integer square root of a given integer (see next subsection) cannot calculate the square root if the actual parameter value is negative. Similarly, a function to withdraw money from a bank account cannot do so properly if there is not enough money in the account. So, we want to specify that certain parameter values are not appropriate for a function (in a particular state).

In most traditional specification languages, this is done using pre-conditions on the parameter values. For example, a pre-condition like $x >= 0$ will be associated with the *Sqrt* function given above. But a pre-condition is not very useful in practice because it cannot be checked automatically, unless a compiler inserts it at each call. Since we don't expect a C++ compiler to do this, the assumption in ADL/C++ is that every function is total. Whenever inappropriate parameter values are supplied, a function indicates that fact by throwing an exception (or returning a special value).

The behavior classification is used to specify this by partitioning the set of possible valid return values and exception values into two sets, **normal** and **abnormal**. When a function returns a value that satisfies the predicate for **normal**, it is considered that the parameter values are appropriate or the pre-condition is satisfied. Similarly, when a function returns value in the **abnormal**, it corresponds to a call with inappropriate parameter values (for that system state). At present there is no check to ensure that the predicates defining **normal** and **abnormal** are disjoint in a **semantics** clause.

We emphasize here that both **normal** and **abnormal** correspond to valid behaviors of the function. If a function implementation ever returns a value that is in neither the **normal** nor the **abnormal** set, then it means that the implementation does not satisfy the specification.

State transformations are specified using the call-state operator "@" to relate values of state variables before and after a call. The details are given in the later sections.

3.1 Specification Vs. Implementation

Even though ADL/C++ specifications look procedural, they are not implementations and implementations cannot be generated from those. Their purpose is to specify *what* a program does as input/output relationships.

Consider an example ADL/C++ specification of a function that computes the integer square root of a given number.

```
specification {
   int Sqrt(int x)
   semantics {
      x >= return * return;
      x < (return + 1) * (return + 1);
   };
};
```

The above specification states that *Sqrt* is a function that takes an integer and returns a (positive) integer which is the floor of the square root of the input parameter value. That is, if a function call *Sqrt(k)* returns a value j, then the following two predicates should evaluate to *true* :

$$k >= j * j$$
$$k < (j + 1) * (j + 1).$$

The keyword **return** denotes the return value of a call to *Sqrt*.

Albeit using C++-like expressions, the above specification just states *what* the function *Sqrt* should do, but not *how* it should be done.

The above specification is partial because no return value can satisfy the behavior specification if the input value of the parameter x is a negative number. This can be easily fixed using exceptions as follows.

```
specification {
   int Sqrt(int x)
           throw(int negativeValue)
   semantics {
     (x < 0) <:> thrown(negativeValue);
     if (normal) {
       x >= return * return;
       x < (return + 1) * (return + 1);
     }
     else {
       x == negativeValue;
     }
   };
};
```

The new declaration of *Sqrt* has an *int* type exception. The new **semantics** clause states that whenever a call to *Sqrt* throws an *int* type exception, it indicates that the actual parameter supplied was a negative value. It also states, by default that the presence of this exception corresponds to **abnormal** and in that case, the thrown exception value is same as the value of the parameter x.

This is an example of a pure function that does not cause any state changes. We give more complicated examples of functions that cause state transformations in the next section.

The complete syntax and informal semantics of the language are presented in Section VII.

4 Specifications for the Important Features of C++

We recall, for the sake of completeness, some of the important features of C++. A C++ program consists of a set of classes and global functions using which instances of these classes interact.

A class declaration contains

- data members that constitute the state of an object of that class,
- constructors that are functions to construct newly created objects by initializing the data members, and
- member functions that operate on the data members to implement the services provided by objects of that class.

Classes of a system are organized in a hierarchy using class derivation. In a class hierarchy, a derived class inherits all the data and function members of a base class. Every instance (object) of a derived class is also an instance of the base class. Member functions declared **virtual** in a base class can be reimplemented in a derived class. Some of the classes can also be abstract, *i.e.*, they define the interface without implementing the member functions.

A C++ function call can throw an exception when it cannot return normally. Values of any type (including primitive types) can be thrown as exceptions.

4.1 An Example: A Bank Account Class

Figures 1 and 2 give the class declaration and ADL/C++ specification of a bank account class *Account*. It has methods *Deposit* and *Withdraw* to deposit and withdraw amounts from accounts, which are instances of this *Account* class. It also has other classes that are used to denote exceptions. The member functions throw these exceptions under certain conditions.

```
#ifndef ACCOUNT_HH
#define ACCOUNT_HH 1

class Account
{
   public :
      enum { MAX_ACCOUNT_NUM = 100 };
      enum AccountTypes { CHECKING = 1, SAVINGS };

   private :
      const AccountTypes accountType;
      const int accountNum;
      long balance;

   public:
      // Exception classes.
      class InvalidAccountNum {
         const int num;
         InvalidAccountNum(int i) : num(i) { };
      };

      class InvalidAccountType {
         const int type;
         InvalidAccountType(int i) : type(i) { };
      };

      class InsufficientFunds {
         const long bal;
         InsufficientFunds(long i) : bal(i) { };
      };

      class NegativeAmount {
         const long amt;
         NegativeAmount(long i) : amt(i) { };
      };

      // Interface
      virtual void Deposit(long amount)
         throw(NegativeAmount);
      virtual void Withdraw(long amount)
         throw(NegativeAmount, InsufficientFunds );
      Account(AccountTypes type, int num, long bal = 0)
         throw(InvalidAccountNum, InvalidAccountType);
};

#endif /* ACCOUNT_HH */
```

Figure 1: A Bank Account Class Declaration in `Account.hh`

```
specification AccountSpec
{
#include "account.hh"

   void Account::Deposit(long amount) throw(Account::NegativeAmount na)
   semantics
   [ abnormal := thrown(na); ]
   {
      (amount < 0) <:> thrown(na);
      normal ==> balance == @balance + amount;
      abnormal ==> unchanged(balance);
   };

   void Account::Withdraw(long amount)
      throw(Account::NegativeAmount na, Account::InsufficientFunds isf)
   semantics
   [ abnormal := thrown(na) || thrown(isf); ]
   {
      @(amount > balance)  <:> thrown(isf);
      amount < 0           <:> thrown(na);

      thrown(isf) ==> (isf.bal == balance);
      thrown(na)  ==> (na.amt == amount);

      if (abnormal) { unchanged(balance); }
      else { balance == @balance - amount; };
   };

   Account::Account(AccountTypes type, int num, long bal)
      throw(Account::InvalidAccountNum ian,
            Account::InvalidAccountType iat)
   semantics
   {
      (num < 1 || num) > MAX_ACCOUNT_NUM    <:> thrown(ian);
      (type != CHECKING && type != SAVINGS) <:> thrown(iat);

      thrown(ian) ==> (ian.num == num);
      thrown(iat) ==> (iat.type == type);

      if (normal) {
         accountNum == num; accountType == type; balance == bal;
      };
   };
};
```

Figure 2: A Specification of a Bank Account Class

We will develop this specification in a top-down fashion. The declaration of the *Account* class is first developed as a header file `account.hh`. It is then included in the specification file to give semantics for the various (member) functions.

The *Account::Deposit* method semantics states that it throws an exception if called with a negative value for *amount*. This is specified using the "`<:>`" operator. This means that a negative value of the *amount* would result in abnormal termination. In the normal case, the value of *balance* increases by the amount that is deposited.

The *Account::Withdraw* terminates abnormally by throwing an exception if either there are insufficient funds in the account or the request was to withdraw a negative amount. Otherwise it terminates normally and the value of *balance* decreases by the amount withdrawn.

The constructor *Account::Account* initializes the values of various data members in case of normal termination. It throws exceptions in case illegal initial values are passed for any of the data members. For example, if the input parameter *num*, which supposed to be the account number, is greater than the largest account number allowed, then an exception *InvalidAccountNum* is thrown.

4.2 ADL/C++ Specifications

We illustrate the ADL/C++ specifications for some of the important features of C++ using the above example in the following subsections. First we briefly discuss the specifications for functions and constructors and then describe specifications for exceptions in detail.

Functions

Behavior specifications for a function can be written using all the (program) variables that can be used in a definition of that function with the same scope and visibility rules. A specification relates the before and after values of that part of the state that the function (being specified) can modify. This includes all the global variables, private as well as public data members in case of member functions and any reference parameters. A specification also states what exceptions, if any, can be thrown and the properties of the values of the exceptions thrown.

As mentioned earlier, a specification for a function gives a classification of its termination (into normal and abnormal), relating return values to parameters and state transformations.

A behavior specification for a function can be given inline along with the declaration or separately, just like a definition.

Constructors

In semantics for constructors, one specifies the initial values for various data members thus giving the initial state of the object. In the example, the semantics for the constructor for the *Account* class specifies that the values of the account number, type and initial balance should be equal to the corresponding parameters.

In a specification for a constructor, the values of the data members of the object (being constructed) cannot be accessed in the call-state ("@" operator) because the object would not have been constructed before a call to a constructor.

Exceptions

C++ allows exception declarations for a function to specify the types of exceptions that can be thrown. An exception can be of any declared type including basic types. Types of exceptions, unlike parameter and return types, are not enforced, but are just informative. Thus a function can throw an exception of a type that is not (convertible to) any of its declared exception types. Unlike parameters, exceptions cannot be named, thus making it difficult to understand in some cases what an exception is supposed to indicate.

ADL/C++ extends the C++ exception declaration syntax to support named exceptions and to declare that a function never throws any exception as well as to specify that a function can throw any exception (using the ellipsis notation).

For every declared exception, the user is encouraged to specify conditions that will cause the exception to be thrown and to relate the value of a thrown exception to parameters state and values.

The semantics of an exception declaration in ADL/C++ is that any implementations of the function can only throw an exception of a type convertible to one of the types listed in the **throw** clause. This enforces a type discipline on exceptions similar to return type. Exception names are treated like output parameter names that are set (by an implementation) only in certain cases. In the post-condition, tests can be made for the presence of exceptions using the **thrown** operator. Exceptions can be used in **semantics** clauses in the same way as formal parameters.

In the example, the semantics for *Account::Withdraw* specifies that it can throw *InsufficientFunds* and *NegativeAmount* exceptions to signal abnormal termination. It also states that if the *InsufficientFunds* exception is thrown (using the **thrown** operator), then the value of its data member *bal* should be equal to *balance*.

It is illegal to use an exception when it is not thrown, or in the call state.

Exception conditions need not always be part of the abnormal section of function semantics. Since exceptions can also be used for communication, they can be a part of normal behavior. For example, a read method of a file class may throw an exception to signal an end-of-file. But, it is not abnormal behavior. On the other hand, if the file is not open, then the exception thrown signals an abnormal termination.

```
with [AccountSpec] // Inherit the previous specification
specification NewAccountSpec
{
   class OverdraftAccount : public Account {
      long maxOverDraft;  // Max overdraft allowed (can change)
      long overDraft;     // How much overdrawn

     public :
      OverdraftAccount(AccountTypes type, int num, long bal)
        : Account(type, num, bal + maxOverDraft) {
         overDraft = 0;
      };

      void Withdraw(long amount) throw(Account::NegativeAmount na,
                                       Account::InsufficientFunds isf)
      semantics {

         /* Semantics of Account::Withdraw are automatically
            inherited as per the ADL/C++ rule for virtual
            functions.
          */

         if (normal && bal < maxOverDraft) {
            overDraft == maxOverDraft - bal;
         };
      };
   };
};
```

Figure 3: A Specification Using Inheritance and Virtual Functions

5 Virtual Member Functions

Virtual member functions of a class allow redefinitions of these in subclasses of that class. The operational meaning of a virtual member function is that a call to it using a pointer (or reference) to a superclass object might actually invoke its implementation in a subclass. This feature is commonly used for implementing abstract classes, extension subclasses and for refining superclass implementations. All these forms of use result in subtyping via inheritance making a subclass of a superclass, also a subtype of the type denoted by the superclass.

In C++, there is no language support for relating the behaviors of virtual functions in a hierarchy. For example, it is possible for a class to implement a queue by inheriting from a super class (with virtual functions) implementing a stack. This example is rather contrived, but a more common error is to implement a function in a subclass overlooking some assumptions made in the superclass. Since one can specify behaviors of functions in ADL/C++, it is possible to include sanity checks for virtual functions in a subclass.

One such simple check can be easily derived from the operational meaning of virtual functions. Since a virtual function called using a pointer to a superclass can actually invoke its implementation in a subclass, the behavior as specified in the subclass should logically imply the behavior in the superclass. This guarantees that as long as the virtual function works correctly according to its specification, it will not violate the corresponding specification in a superclass, thus producing the same observable behavior. This guarantees the users of superclasses won't have surprises when new subclasses are added.

ADL/C++ adopts a stronger version of the above check. It requires that the semantics for a virtual member function redefined in a derived class should obey the semantics given for it in the base class as interpreted in the context of the base class. The idea here is to impose this behavioral constraint automatically. Thus the assertions in the derived class are strengthened automatically by those given in the superclass. So, whenever a **semantics** clause for a virtual function evaluates to true, one can safely conclude that it also satisfies the assertions given in the super class. This is analogous to the generation of proof obligations in other formal specification languages, *e.g.*, [5]. Note that this does not have any effect on pure code inheritance when there are no virtual functions.

The main idea here is to impose some discipline on inheritance so that behavioral subtyping is achieved. Our current semantics for virtual functions is inspired by the behavioral notion of subtyping described in [7] and a similar scheme called *weak behavioral subtyping* is supported in Larch/C++ [6].

We illustrate different uses of specifications for virtual functions using examples in the following subsections.

5.1 Extension Subclasses

A subclass can extend a superclass by introducing new data and/or function members. It may also redefine some member functions of the superclass using the newly introduced members. If any virtual function of the super class is redefined in the subclass, ADL/C++ ensures that the new definition does not violate the semantics given in the superclass.

Consider the ADL/C++ specification given in Figure 3, of a new bank account class which inherits from the *Account* class. It allows an overdraft upto a certain amount which can change. Therefore the specification for the new *Withdraw* method specifies how the value of the variable *overdraft* changes if the account is being overdrawn. Since it is virtual, by our semantics it automatically inherits the specification given in the base class and therefore it is not necessary to specify that part. In this case, the variable *balance* merely denotes the amount available for withdrawal, not the real balance.

5.2 Abstract Classes

A common use of virtual functions in C++ is the specification of abstract classes. The abstract properties can be specified for the virtual functions in the abstract class and in the implementation classes, these functions would have to obey all those properties automatically because of the semantic constraints imposed by ADL/C++. In this case, ADL/C++ partially guarantees implementation correctness.

As an example, consider the specification of an abstract class of a stack of integers and its implementation using an array of integers given in Fig. 4.

The above specification is partial in many respects. For example, it does not specify what happens if an attempt is made to push more than 1000 elements or a call to *Pop* is made on a stack of zero size. These can be easily specified. The point to note here is that the semantics for the member functions given in *StackByArray* should imply those in the class *Stack*. To ensure that, ADL/C++ implicitly strengthens the semantics of the functions in

```
specification StackOfIntegers
{
   class Stack {
      public :
         virtual void Push(int ) = 0;
         virtual int Pop(int ) = 0;
         virtual int Top() = 0;
         virtual int Size() = 0;
   };

   void Stack::Push(int i) semantics {
      if (normal) {
         Top() == i;
         Size() = @Size() + 1;
      };
   }
   int Stack::Pop() semantics {
      if (normal) {
         @Top() == return;
         @Size() = Size() + 1;
      };
   }

   class StackByArray : public Stack {
      int size;
      int index;
      int array[1000];

      public :
         void Push(int i) semantics {
            array[index - 1] == i; index == @index + 1;
         };
         int Pop() semantics {
            @array[index + 1] == return; @index == index + 1;
         };
         int Top(int) const semantics {
            return == array[index - 1];
         };
         int Size(int) const semantics {
            return == index;
         };
   };
};
```

Figure 4: A Specification of a Stack Interface and Implementation

StackByArray by the corresponding semantics of the same function from the *Stack* class. For example, the complete semantic specification of *StackByArray::Push* would also include *Stack::Push* semantics interpreted in the context of declaration of class Stack. This makes sure that the implementation does not violate the specifications given for the interface so that clients using the interface can have some guarantees.

5.3 Refinements

Another use of subtyping is for specialization (or refinement), *i.e.*, a superclass has more general behavior which will be specialized (or refined) in the subclass. The semantic constraints on virtual functions are useful to ensure that the redefinition is indeed a specialization of the behavior specified in the superclass.

For example, the sorting functions specified in the next section can be easily specified using the C++ inheritance mechanism by appropriately making the first sort function a virtual member of a class and then redefining it as a stable sort function in a subclass. This is possible because the original sort function does not constrain the relative order of elements with equal keys, thus allowing the subclass to add the extra constraint.

6 Specification Reuse

Whenever the intended behavior of a function is a refinement or composition of behavior(s) of other function(s) in the system, it is possible to reuse the already developed specifications. ADL/C++ provides the **behavior** construct for this purpose. Without this construct, specifications used have to be duplicated in such cases, which is not desirable from the maintenance point of view.

Most formal specification languages provide reuse mechanisms such as refinements and extensions to support this. The units of refinement are usually theories, concepts or abstract datatypes. Semantics of these mechanisms can be complex and difficult to understand for the untrained person.

ADL/C++, on the other hand, has function specifications as units of specification. The reuse of specification is supported at the level of **semantics** clauses, using **behavior** expressions. The semantics of this construct has been kept deliberately simple in line with the overall ADL philosophy.

behavior_expression ::=
    behavior "(" { *selector_expr_prefix* }
        IDENTIFIER < { *param_list* }
        { : *expression* } >

selector_expr_prefix ::=
    expression ("." | "->")

The **behavior** operator requires a function name, values for parameters and a value to denote the return value of a call to the function with those parameter values. The actual parameters might also include (a pointer to) the object in the case of member functions. The syntax is similar to a function call syntax, so for members, the object (or pointer) is supplied using the appropriate member selector operator. There should be a **semantics** clause associated with the function name used, otherwise it is an error.

In "$behavior(a.f)<arg_1, ..., arg_n : retVal>$", the types of the expressions $arg_1, ..., arg_n$ obey all the rules that the types of actual parameters to f do and the type of *retVal* should be the return type of f. The type of a behavior expression is boolean.

"$behavior(a.f)<arg_1, ..., arg_n : retVal>$" evaluates to *true* iff the **semantics** clause corresponding to the (member) function f (of a) does, when evaluated in the scope in which it is specified, by setting the values of its n parameters to $arg_1, ..., arg_n$ respectively, and setting the value of **return** to *retVal*.

Exception values need not and cannot be passed to a **behavior** expression because they have a "global" scope. Thus the exceptions of the function using the **behavior** expression will be used to evaluate the semantics clause of the function used.

The **behavior** construct is quite general in the sense that a single **semantics** clause can use any number of

behavior expressions of any visible function and a single function behavior can be used any number of times in a single **semantics** clause. At the same time, the semantics of this construct is straightforward, because intuitively, it is just the truth value of the **semantics** clause evaluated with the given parameter values (and return value).

Function Call Vs. behavior Expression

We emphasize here that a **behavior** expression is different from a function call expression (which is allowed, but not encouraged in ADL/C++). In general, these two expressions are not interchangeable. A function call invokes an implementation and can return a value, whereas it is not possible to get a return value using a **behavior** expression. It merely evaluates the boolean formulae given in the semantics clause. A set of parameter values and the expected return value for those have to be supplied to the **behavior** expression to evaluate the **semantics** clause.

For example, the call to *GetAccount* in the *Bank* specification, given in the next subsection, cannot be replaced by a **behavior** expression. This is because it returns a pointer whose value can only be determined at runtime and the *Bank::Deposit* method cannot give an expression for that pointer value for a **behavior** expression using *GetAccount*.

This distinction is very important in ADL/C++ because specifications and C++ implementations of functions can coexist in an ADL/C++ specification module.

We can give an alternative (testing) semantics to a function call expression appearing in **semantics** clauses using the **behavior** construct. With this semantics, an expression $f(e_1, e_2, ..., e_n)$ in ADL/C++ can be considered equivalent to

$$behavior(f) < e_1, e_2, ..., e_n : f(e_1, e_2, ..., e_n) >,$$

where the expression $f(e_1, e_2, ..., e_n)$ (denoting the return value for the **behavior** expression) is the usual C++ function call.

This guarantees that all the uses of f satisfy its specification within an ADL/C++ specification. This will make testing a bottom-up process, thus ensuring that every function call used for testing is itself tested for validity. We encourage users to do this explicitly whenever they need to invoke a function implementation.

The **behavior** construct can be used for specification in a variety of situations as described in the following subsections. Some of these are similar to the virtual function mechanism described in the previous section. In fact, the semantic constraints for virtual functions are implemented using the **behavior** expression. The distinction here is that ADL/C++ inherits the virtual function mechanism from C++, whereas the **behavior** expression is a reuse mechanism introduced by ADL/C++ itself.

6.1 Reuse for Refinement

Sometimes, the behavior of a function is a simple refinement of the behavior of another function. In that case, if the semantics of the function being refined is already developed, then the semantics for the new function can be developed using those available specifications. For example, consider the following specification of a *Sort* function :

```
specification SortSpec
{
   typedef Record *RecordArray;

   RecordArray Sort(RecordArray records)
   semantics {

      IsPermutation(return, records);

      forall (int i : int_range(0,
                 size(records) - 2)) {
         return[i].key <=
                  return[i + 1].key;
      };
```

```
   };
};
```

Now consider a stable sort function, *i.e.*, a sort function which preserves the relative order of elements with equal keys after sorting. A specification for this function can be written as a refinement of the sort function to say that it sorts and also preserves the relative order of elements with equal keys. In this case, the specification *StableSort* will be concise and intuitive if it can be written using the existing post-condition for *Sort* in the spirit of reuse supported by C++ by way of inheritance. This can be done as follows :

```
with [SortSpec]
specification NewSortSpec
{
   RecordArray StableSort(RecordArray
                              records)
   semantics {
      // It behaves like Sort.
      behavior(Sort) <records : return>;

      // Relative order of elements
      // with equal key is preserved.
      forall (int i, j : int_range(0,
                size(records) - 1)) {
         if (i < j && records[i].key ==
                      records[j].key) {
            exists (int m, n : int_range(
                0, size(records) - 1)) {
               m < n &&
               records[i] == return[m] &&
               records[j] == return[n];
            };
         };
      };
   };
};
```

6.2 Reuse for Contained Objects

Another situation where the **behavior** construct is useful, is when an object of a class contains objects of another class and uses the contained class methods to implement its own methods. The specifications for methods of the container can be written in terms of the specifications of the contained objects using the **behavior** expression. This gives modularity to specifications similar to the modularity of implementations as supported by C++.

For example, consider a bank class which has methods to deposit to and withdraw from an account given the account number. Using the previously specified *Account* class, a specification of the bank's deposit method can be given as :

```
with [AccountSpec]
specification Bank {
   class Bank {
      Deposit(int acNum, long amt)
         throw(int acNotFound)
      semantics {
         // Get the account object
         assign Account *acPtr with
              acPtr = GetAccount(acNum);
```

```
          if (normal) {
             behavior(
                acPtr->Deposit)<amt>;
          };
       };
   };
};
```

This specification states that the bank's deposit method locates the account object (using *GetAccount*) corresponding to the given account number and then the amount is deposited into the located account. It also throws an exception if it cannot find an account with the given number.

An advantage with this specification is that if later on some subtypes are added to the *Account* class, say something like an account with overdraft protection, the bank specification need not be rewritten, much like C++ implementation of the *Bank* class need not be rewritten (if properly implemented).

6.3 Reuse for Special Case Functions

It is common for many classes to provide functions that are shorthands for other functions called with particular values(s) for parameter(s). The behavior of such a shorthand function can easily be specified using that of the general function.

As an example, consider the following declaration of a *Vector* class :

```
class Vector {
  public :
   void Add(int i);
   int GetElem(int index);
   int GetFirstElem();
};
```

The function *GetFirstElem* is just a shorthand for *GetElem* with 1 as the value of the argument. The semantics for *GetFirstElem* can simply be *behavior*(*GetElem*)<*1* : *return*> which intuitively states that the function *GetFirstElem* behaves the same way as *GetElem* would when called with 1 as the value of its only argument.

```
struct Account__Withdraw
{
   Account *ACF_AccountPtr;
   long amount;

   int ACF_Pre_Val1;
   long ACF_Pre_Val2;

   Account__Withdraw(Account *AccountPtrIn, long amountIn) :
      ACF_AccountPtr(AccountPtrIn), amount(amountIn) { };

   void Pre();
   void Call();
   int Post();
};
```

Figure 5: Class for Implementing the Specification of *Account::Withdraw*

7 Translation of ADL/C++ Specifications to C++

One of the primary design goals of ADL/C++ is to translate specifications into executable code so that (C++) implementations can be tested against ADL/C++ specifications. Since the expression language includes the C++ expression language, C++ code generation is straightforward for most of the constructs.

In the old ADL/C, the translation was almost trivial, as every **semantics** clause is translated into a C function. However, this is inadequate for the more powerful ADL/C++. In particular, there are two new constructs that cannot be supported using that scheme - one is that of checking the exception specifications and the other one is the **behavior** construct for reusing specifications. So, we came up with a completely new translation scheme for translating ADL/C++ specifications into C++ code.

There are some other applications of ADL apart from testing, like interposing between clients and servers where this translation can be used. In those applications, violation of a pre-condition (a boolean expression in the call-state) reflects a problem with the client, whereas the violation of a post-condition is problem with the server. A simple translation suited only for testing will not work for these applications. Thus, we have identified a few major components of an ADL specification and translation of each of these components is done independent from one another.

The general scheme of translation consists of generating global variables to store exceptions raised, local variables to calculate and store values the of call-state expressions, parameters and return values. These are generated as members of the class generated for the **semantics** clause in which they appear. Assertion checking code is generated as member functions of this class.

Below we will describe how each of the ADL/C++ constructs is translated to C++ code.

7.1 Output Files

An ADL/C++ specification S generates 3 files when translated. They are `S.hh`, `S_ACF.hh` and `S_ACF.cc` The `S.hh` file has all the class declarations that are included in the specification with minor modifications to remove the access restrictions. This can be included in the test-driver so that there are no compilation errors (due to member access rule violation). The suffix `ACF` stands for Assertion Checking Functions. `S_ACF.hh` contains all the declarations generated from the **semantics** clauses. `S_ACF.cc` contains definitions of all the (global and member) functions declared in `S_ACF.hh`. In addition, if there are any auxiliary declarations used, they are kept in another file `S_AUX.hh`. `S_ACF.hh` includes the other 2 `.hh` files.

7.2 Semantics Clause

A **semantics** clause is simply a boolean expression defined using the parameters, return values and other global variables. To support the **behavior** construct however, we need to give it a first class status. We do this by generating a class for the **semantics** clause of a function. The translated class will have the following members :

- Data members to store the parameter values, including the **this** pointer (for non-static member functions).
- Data member to store the return value (if any).
- Data members to store the values of the call-state expressions
- A constructor which takes the same parameter type as the function (whose specification it represents) and stores the values in the corresponding data members.
- A function member *Pre()* to calculate and store the values of the call-state expressions used in the specification.
- A member function *Call()* to call the function under test with the supplied actual parameters and to store the return value.
- A member function *Post()* to evaluate the ADL assertions using the stored values of the actual parameters, call-state expression values and the return value.

For example, Fig. 5 shows the class declaration for implementing the specification of the *Account::Withdraw* method given in the previous sections. The definitions of the methods are given in Fig. 6.

The first datamember corresponds to the **this** parameter to the *Account::Withdraw* function. The second one is the parameter. The next two data members correspond to the two call-state expressions used in the specification respectively. Storing the actual parameters is done using the member initialization syntax so that it is general enough to handle **const** and reference type parameters (which can only be initialized, but cannot be assigned to).

7.3 Exceptions

ADL/C++ enforces exception specifications to be exact. To ensure this, code is generated in the translation, to check that the type of an exception thrown by a function converts to at least one of the declared exception types. The *Call* method catches a valid exception and stores it for use in assertion evaluation. Each distinct type of exception thrown is stored in a global variable. Note that the name of an exception is useful only for the readability specifications and the variable in which an exception is stored is determined solely based on its type.

Translation of Exception Declarations

Each exception type declared for a function gets translated into a *catch* block in the *Call()* method of the code for its **semantics** clause. The body of the *catch* block does the following.

1. Sets a flag to indicate that the particular type of exception is thrown.
2. Stores the value of the thrown exception in the global variable corresponding to that type.
3. It rethrows the exception so that it can be caught again and recorded as an exception of any other convertible types.

For example, in Fig. 6, the *Call* function has one catch block for each of the declared exceptions for *Account::Withdraw*. The nested *try* blocks guarantee that an exception is recorded as an exception of every type to which its type can be converted. The level of nesting is equal to the number of declared exceptions.

The two flags *Account__NegativeAmountThrown* and *Account__InsufficientFundsThrown* are set in their respective catch blocks to indicate that the exception caught is thrown by the called function. The actual exception objects themselves are stored in global variables *Account__InsufficientFundsObj* and *Account__NegativeAmountObj* in the corresponding catch blocks. The global flag *ACF_SomethingThrown* is set in the *catch(...)* block to indicate that an exception is thrown whose type is not convertible to any of the declared exception types.

Translation of the thrown Operator

A *thrown* expression in assertions is translated to checking the flag corresponding to its type. Similarly, the name of an exception is translated to the global variable corresponding to its type.

In the *Account::Withdraw* semantics, the *if (thrown(isf))* clause is translated to *if (Account__InsufficientFundsThrown)* and the name *isf* is translated to *Account__InsufficientFundsObj*.

Validation of Exceptions

The actual checking for legality of exceptions is done in the *Post* method as follows :

- If the declaration of this function has no **throw** clause, then the specification is violated if there is any exception thrown, as indicated by the corresponding flag.
- If some exception is thrown, but its type is not one of the declared exception types, then it is an illegal exception.

In Fig. 5, the *Post* function initializes *result* to *!ACF_SomethingThrown* that will ensure that no exception other than the declared ones is thrown.

```
void Pre() {
   ACF_Pre_Val1 = (amount > ACF_AccountPtr->balance);
   ACF_Pre_Val2 = ACF_AccountPtr->balance;
}

void Account__Withdraw::Call() {
   try {
      try {
         ACF_AccountPtr->Withdraw(amount);
      } catch(Account::NegativeAmount na) {
         Account__NegativeAmountThrown = 1;
         Account__NegativeAmountObj = na;
         throw;
      }
   } catch(Account::InsufficientFunds isf) {
      Account__InsufficientFundsThrown = 1;
      Account__InsufficientFundsObj = isf;
   } catch(...) {
      ACF_SomethingThrown = !Account__NegativeAmountThrown &&
         !Account__InsufficientFundsThrown;
   }
}

int Account__Withdraw::Post() {
   int result = !ACF_SomethingThrown;
   int ACF__abnormal = Account__NegativeAmountThrown ||
                       Account__InsufficientFundsThrown;
   int ACF__normal = !ACF__abnormal;

   result = result && (!ACF_Pre_Val1 || ACF__abnormal) && (!(ACF__abnormal &&
              Account__InsufficientFundsThrown) || ACF_Pre_Val1);
   result = result && EXCEPTION_OP(amount < 0, Account__NegativeAmountThrown);

   if (Account__InsufficientFundsThrown)
      result = result && ((Account__InsufficientFunds)
              Account__InsufficientFundsObj).bal == ACF_AccountPtr->balance;
   if (Account__NegativeAmountThrown)
      result = result && ((Account__NegativeAmount)
                       Account__NegativeAmountObj).amt == amount;
   if (ACF__abnormal)
      result = result && ACF_AccountPtr->balance == ACF_Pre_Val2;
   else
      result = result && ACF_AccountPtr->balance == ACF_Pre_Val2 - amount;

   return result;
};
```

Figure 6: Translation of the Specification of *Account::Withdraw*

```
struct __Sort {
   RecordArray records;
   RecordArray retVal;

   __Sort(RecordArray recordsIn) : records(recordsIn) { };
   void Pre();
   RecordArray Call();
   int Post();
};

struct __StableSort {
   RecordArray records;
   RecordArray retVal;
   __Sort __SortObj;

   __StableSort((RecordArray recordsIn) :
      records(recordIn), __SortObj(records) { };

   void Pre() {
      __SortObj.Pre();
      // Other call-state expressions of the semantics clause of StableSort
   }

   void Call() {
      retVal = StableSort(records);
      __SortObj.retVal = retVal;
   }

   void Post() {
      int result = __SortObj.Post();
      // Code for evaluating other assertions.
   }
};
```

Figure 7: Translation of the Specification of *Sort* and *StableSort*

7.4 Translation of the behavior construct

The meaning of a **behavior** expression is the value of the **semantics** clause associated with the function with the supplied parameter values. Due to the presence of call-state expressions, the translation of the **behavior** construct goes partly into the *Pre* and partly into the *Post* functions. The basic idea is to first evaluate the **semantics** clause by plugging in the values of the expressions used to denote the parameter and return values.

In the following discussion, let us assume that the **semantics** clause of a function f has a **behavior** expression $behavior(g) < s_1, ..., s_m : r >$ and the translated class for the semantics of f is f_C.

Intuitively, in *f_C::Pre* a call to the *Pre* of g's **semantics** clause is made. The return value of *f_C::Post*, is a conjunction of the return value of *Post* of g's **semantics** clause with the values of all the other assertions.

The translation of the **behavior** expression is done in the following steps :

1. The function appropriate to the types of $s_1, ..., s_m$ is selected using the C++ overloaded function call resolution mechanism. If that function has no **semantics** clause, then it is an error. Let the class corresponding to this **semantics** clause be g_C.
2. A data member g_CObj of type g_C is declared in f_Ca.
3. g_CObj is initialized in the member initialization of $f\_C :: f\_C$.
4. A call $g\_CObj.Pre()$ is made in $f\_C :: Pre$.
5. In $f\_C :: Call$, $g\_CObj.retVal$ is set to (the translation of) r.
6. In $f\_C :: Post$, the result is conjuncted with a call $g\_CObj.Post()$.

For the specification of the *StableSort* function given in section 4, the translated code will be as shown in Fig. 7.

As the translation shows, the semantics of the **behavior** expression is straightforward and it is not affected by the presence of exceptions.

7.5 Other ADL Expressions

Translation of most of the other ADL-specific expressions is straightforward. We will briefly describe those below.

- *With* clauses are translated into "*#include*" directives to include the corresponding `_ACF.hh` files.
- *Group Expressions* are translated into blocks with each assertion in the group translated to a (possibly compound) statement.
- *Normal and Abnormal* get translated to local variables inside the *Post* method and are initialized by the specified expressions.
- Binding declarations are generated as declarations with the specified initialization(s).
- *Logical expressions* (non short-circuited) are first decomposed into their components. These components are evaluated and then the final result is computed using the logically equivalent short-circuited versions of the expressions.
- *Implication expressions* are translated to equivalent *if ...* expressions.
- *if ... elsif ... else ...* expressions are translated to C++ *if ... else ...* statements.
- *Quantified Expressions* are translated to C++ *for* loops with body containing the appropriate translation for the expression. If the quantifier is a **forall**, then the enclosed expression is evaluated for all the values in the specified (finite) domain. If the quantifier is an **exists**, then the loop is executed evaluating the enclosing expression until for some value from the domain it evaluates to true. If no such value exists, then the value of the expression is false.

- *Unchanged* expression is translated into the equivalent of $e == @e$ where e is the argument expression.
- *Return* expression is translated into the corresponding data member of the class implementing its **semantics** clause.

8 Testing and Validation

One of the important benefits of writing ADL/C++ specifications is automatic test-validation. A C++ implementation can be tested by giving it a sample input data and then the test can be validated to see if it is a success or failure using the ADL/C++ specification. We first describe the procedure for doing this and then an implementation of this using the translation scheme described in the previous section.

8.1 Test Validation

At present, only unit testing of (member) functions is supported. Given a function declaration to be tested along with a sample data, the validation can be done in the following steps.

- Store the values of all the parameters that are referred to in the **semantics** clause of the function.
- Evaluate all the call-state expressions using the given sample input data and store them in temporary variables.
- Invoke the function with the given sample data to get any return values.
- Evaluate the predicates for normal and abnormal. If neither one evaluates to *true*, then the test *failed*.
- Using the stored parameter values, call-state values and the return value, evaluate each of the expressions that is given in the **semantics** clause.
- If any of the expressions evaluates to *false*, then the test *failed*, that is, the (C++) implementation does not satisfy the ADL/C++ specification for that particular input data. Otherwise, the test *succeeded*.

Intuitively this means that any test should conform to either the normal or the abnormal behavior specified, otherwise the implementation returned or thrown an unexpected value. A test might also fail because the caused state changes do not conform to the specification.

For example, consider the following incorrect implementation of the *Account* class whose specification is given in Figure 2.

```
#include "account.hh"

void Account::Deposit(long amount)
     throw(Account::NegativeAmount na)
{
   balance = balance + i;
}

void Account::Withdraw(long i) throw
     (Account::NegativeAmount na,
      Account::InsufficientFunds isf)
{
   if (i >= balance)
      throw Account::NegativeAmount;

   if (i >= balance)
      throw Account::InsufficientFunds;
```

```
   balance == balance - i;
}


int ACF__Test__Account__Deposit(Account *ACF_ACF_AccountPtr,
                                          long amount)
{
   try
   {
      bank__InitExceptionFlags();
      Account__Deposit ACF__Account__DepositObj(ACF_ACF_AccountPtr, amount);

      ACF__Account__DepositObj.Pre();
      ACF__Account__DepositObj.Call();
      return ACF__Account__DepositObj.Post();
   }
   catch(...)
   {
      fprintf(stderr, "Uncaught Exception during testing\n");
      return 0;
   }
}
```

Figure 8: Test Validation Function for *Account::Withdraw*

As it can be seen, the implementation of the *Account::Withdraw* method is incorrect because it does not update the balance. This is not an unlikely error because of the lexical similarity between the C++ operators `==` and `=`. Similarly the *Account::Deposit* method is wrong because it does not check for the value of *amount* being negative in which case it is specified to throw an exception.

Any test for the *Withdraw* method with a nonnegative amount will make its post-condition false, and thus is classified as a failure. Even though this may be a simple bug to catch, there are other examples with hard-to-catch bugs that can be detected by testing with appropriate test data (and a correct specification). Similarly, a test of *Account::Deposit* with a negative value for *amount* will fail because the first assertion in the semantics fails because *amount* is negative and the function did not terminate abnormally (as the *NegativeAmount* exception is not thrown).

An ADL/C++ specification thus gives a formal basis for test validation rather than the tester deciding whether the test failed or succeeded based on informal specification. It is also possible to give a more fine-grained test report in case of a failure, by looking at the individual predicates that have failed for a particular test data.

As mentioned earlier in section VI, closure-testing can be done using the **behavior** expressions for every function call used in the semantics (or even implementation) of a function. This will guarantee that for the particular test-data every invoked function satisfies its specifications. Another byproduct of this kind of testing is that the test-data for the functions used with the **behavior** expressions can be enriched with the values generated while testing the function using it. One possible problem with this approach could be the resources required for testing, but whenever the overhead is not much, it can enhance the correctness of the testing process.

8.2 Implementation

The test validation procedure described above can be automated using the translation of ADL/C++ to C++ described int the previous section. This has the advantage that a large number of tests can be run automatically whenever an implementation is modified. Detailed reports for failed tests can also be generated. This will help in quickly locating the problem in the implementation that caused the test failures.

For every member or global function with an annotation, a global function is generated that can be used for testing the function. The signature of this function is same as the constructor of the ACF object (described in the previous section). This function first initializes the various flags used to indicate the presence of exceptions, the creates the ACF object for the function to be tested and invokes the *Pre*, *Call* and *Post* methods in that order, on that object.

For example, the global function for testing the member function *Account::Withdraw* is given in Fig. 8, using the translation for the same function described in Fig. 5. These functions are made global so that they can be used for other purposes like sequence testing etc. which we have plans to support in the future.

If there is a test suite available for the function, one can write a driver to call this function repeatedly on each of the test data sets of the test suite.

At present, the test data has to be supplied by the user. We are currently working on coming up with a language like TDD [9] to specify test data for testing C++ member functions. In fact, to a large extent, TDD itself can be used to specify test data for C++ functions as well.

We are also looking at ways to generate "interesting" test data from ADL/C++ specifications. For simple data types, it seems to be very feasible to automatically generate reasonable test data. For example, in the *Account::Deposit* method, a negative value for the parameter is interesting because it can cause an exception to be thrown. So, any reasonable test data should have at least one such value for *amount*.

Caveat: While testing using ADL/C++ specifications, there can be calls to constructors, assignment operators and copy constructors. It is the users responsibility to keep these clean, otherwise, the validation becomes inefficient and imprecise.

9 Comparison and Related Work

A++ [1] is a language for annotating C++ programs with semantics constraints. This is close to ADL in spirit in that it uses (mostly) C++ syntax for doing this. But, unlike ADL, A++ provides facilities for writing complex specifications, like legal values for datatypes, axioms defining classes i*etc.* Developing any practically usable tools using A++ is ambitious, because specifications could range from high-level axioms for classes to low-level assertions for statements. On the other hand, we have started with the modest goal of associating behavior specifications to functions. This is shown to be achievable and very useful by the success of ADLT using ADL/C. So, have we adopted a similar framework for C++ and are building on top of this to incrementally support specification of other features of C++, like classes and inheritance. It has been demonstrated by our semantic constraints on non-static virtual functions, that it is possible to develop ADL/C++ for specifying the more complex features of C++ with this simple idea as the basis. The A++ annotations are *intrusive*, *i.e.*, embedded within programs, making it difficult to use with pre-compiled libraries. ADL/C++ specifications are *non-intrusive* and hence can be used for specification even when the source code is not available.

The other effort with similar goals that we know of is Larch/C++ [6]. This is a member of the Larch family of languages. This supports two-tiered specifications. First the abstract values are described in the larch shared language LSL. Then the specifications of classes are given in Hoare-style pre and post-conditions for member functions. In our opinion, this is a burdensome process especially for practitioners who want to use formal specifications, because specifications have to be written in a different language with totally different semantics. In ADL also the language is different, but it is built around C++ and we made it a point to introduce as little new notation as possible. We should also point out that Larch has a large collection of LSL traits which can be used to describe most commonly occurring data structures. However, to be able to pick the correct trait, one needs to understand the semantics that describe the behavior of the trait.

To the best of our knowledge, there is no support for testing C++ implementations against Larch/C++ specifications. Larch has a very good collection of tools, most important of which is LP, the larch prover. One can hope to do some reasoning about Larch/C++ specifications using LP. At present there is no support for reasoning about ADL specifications.

Another member of the Larch family, LCLint[3] supports specification based checking of C programs. Static analysis of C code can be done using this tool to infer dynamic properties like memory leaks, incompatible types for parameters, illegal use of global variables etc. For this, it extends the standard *lint* program to use specifications written in the LCL [4]. The basic difference between LCLint and ADL/C++ is that LCLint needs the source code to do its analysis, whereas ADL/C++ doesn't. The primary goal of ADL/C++ is to make testing automatic so that third party testing can be done where the developers are not involved, whereas LCLint is like a high level compilation activity to detect more error-prone areas of a program than a traditional C compiler is capable of. LCLint works on a program at compile time whereas ADL/C++ is a checker at runtime, thus they complement each other. LCLint has a fine-grained checking at the statement/expression level as compared to ADL/C++ which works at the level of functions which is appropriate for testing API's.

10 Conclusions and Future Work

We have designed a language ADL/C++ to write behavioral specifications for C++ programs. The important aspects of this language are :

- *It is easy to learn and use for real software engineers.* We have been careful in designing specification constructs that are simple and easy to use for the practitioner. While this has reduced the expressivity of the language, it is surprising how much can still be expressed. ADL/C has shown itself to be useful and practical at the same time, and ADL/C++ will only improve on this.
- *Specifications can be compiled into checking code.* All ADL/C++ specifications may be compiled into code that can test for correct behavior (or conformance) of API implementations. Note that this is different from "executable specifications" where it is possible to generate executable implementations directly from a specification. ADL/C++ specifications offer a reasonable middle ground where the specifications may be used to efficiently check conformance, while at the same time allows easy development of specifications.
- *Specifications may be developed independently of the program.* ADL/C++ specifications may be written as separate files (that "include" C++ header files). Hence ADL/C++ facilitates the development and use of specifications by organizations that do not have access to C++ source code (*e.g.*, some proprietary API's come with precompiled libraries).
- *Various kinds of tool support is possible.* Checkable specifications may be used in many ways. A system such as ADLT uses ADL/C++ for conformance testing. In such a system, test-data is produced independently of the writing of the specifications. The tool then generates a driver that exercises the API on the test-data, checking the system behavior each time with the specifications. Specifications may also be used to generate self-checking interfaces in cases where security is an issue. Self-checking interfaces are also useful during the development cycle.

The language at this point is useful for specifying a large class of C++ programs. But, there are still some unresolved issues. The most important one is specifications for classes. Invariants and history constraints are useful in specifying class behaviors, but they are not supported.

Another problem is with specifying (methods of) abstract classes. Since they do not have any state information, it is difficult to write post-conditions. The only alternative is to write post-conditions using a crude implementation (in the form of auxiliary declarations). This makes specifying abstract classes complicated and error-prone.

We are investigating this problem in the global context of the ADL-II project, to have a common paradigm for ADL specifications for objects in all the different implementation languages like Java™, C++ etc. We have been proposing a trace-based specification language for class behaviors that can be used on top of ADL/C++ specifications. We are designing this language so that it retains the two defining features of ADL - testability and closeness to the base implementation language. The details will be presented in a different paper.

Availability

A subset of the language described here will be adopted as part of the ADL-II project being developed by the Suntest group (formerly known as the Primavera group) of Sun Microsystems Laboratories. The development of the frontend of the tool consisting of the parser and a (partial) type checker is complete. This initial version will be available in about 6 months. Contact the authors for further information.

Acknowledgments

We would like to thank Sriram Sankar who initiated this project and actively participated in the initial stages of the design. We would also like to thank Rajiv Joshi, Ashvin D'Souza, Roongko Doong and Mark Hefner for their participation in the numerous discussions during the design of the language.

References

[1] M. Cline and D. Lea. The Behavior of C++ Classes. *Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications*, Marist College, 1990.

[2] 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.

[3] David Evans, John Guttag, Jim Horning and Yang Meng Tan. LCLint: A Tool for Using Specifications to Check Code. SIGSOFT Symposium on the Foundations of Software Engineering, December 1994.

[4] 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.

[5] 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.

[6] Gary T. Leavnes. Larch/C++ Reference Manual. Department of Computer Science, Iowa State University, Sept 1995.

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

[8] 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.

[9] Sriram Sankar, Roger Hayes. Specifying and Testing Software Components using ADL. SMLI TR-94-23. *Sun Microsystems Laboratories, Inc.*, April 1994.

[10] Sriram Sankar, Mark Hefner. Formal Methods Applied to Conformance Testing. In *ICSE-17 Workshop on Formal Methods Application in Software Practice*, April 1995.

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

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

[13] 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.

[14] Sreenivasa Rao Viswanadha, 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 Syntax and Semantics

Much of ADL/C++ syntax includes C++ syntax, but it also has some syntactic constructs of its own. We will describe the ADL/C++-specific syntax and semantics in a way similar to the C++ semantics presented in [12] in the following subsections.

A.1 Specification Structure

adl_specification ::=
    { *with_clause* }
    specification *spec_name*
    "{" (*annontated_declaration*) + "};"

Every specification has a name and consists of one or more annotated declarations, that is, C++ declarations optionally annontated by behavior specifications. At present, we require that the name of the file be same as the name of the specification with **.adl++** as the file name extension. ADL/C++ is case-sensitive just like C++ is.

An ADL/C++ specification can also import other ADL/C++ specifications using the **with** clause.

with_clause ::=
    with "[" (*spec_name*)
        ("," *spec_name*) * "]"

A specification can use all the declarations and any annotations from any of the specifications listed in the **with** clause. The tools will not process the imported specifications to generate any output files, unlike specifications that are textually included using the "#include" directive.

A.2 Declarations

All C++ declarations like typedefs, classes, enums, variable declarations *etc.* can be present in an ADL/C++ specification. To facilitate users to include "real" C++ header files, we allow inline function definitions to be present in ADL/C++ specifications. At present, only function declarations can be annotated to give behavior descriptions.

annontated_declaration ::=
    C++_declaration
  | *auxiliary_declaration*
  | *C++_function_declaration annotation*

C++_declaration is any valid C++ declaration and *C++_function_declaration* is a C++ function declaration without the **throw** clause.

Annotations

An annotation consists of ADL/C++ exception specifications, binding definitions for normal and abnormal behavior of the function, and a list of formulas that describe the behavior of the function.

annontation ::=
    { *adl_exception_spec* }
    semantics
    { *behavior_definitions* }
    group_expression ";"

adl_exception_spec ::=

throw { (exception_list) }

exception_list ::=
    param_declaration
        ("," *param_declaration*) *
  | "..."

behavior_definitions ::=
    "[" ((**normal** | **abnormal**)
        ":=" expression ";") + "]"

In ADL/C++, exception specifications can be named. So, the syntax for this is same as that of a C++ formal parameter declaration. If no exception is specified following the throw clause, then the function can not throw any exception. If there is a "..." in the throw list, then the function can throw any exception.

Note that all the variables used in the annotation part are program variables (and not logical variables). This is one of the important differences between ADL and other specification languages. This is done to reduce the introduction of extra variables and concepts and also to facilitate the translation of these into C++ for testing.

An annotation for a function is a post-condition, *i.e.*, all the expressions except call-state expressions (see below) that constitute the group expression will be evaluated using the values of the variables after a call to the function.

The behavior definitions classify the behaviors of the function into normal and abnormal behaviors. If neither normal nor abnormal behavior is explicitly defined, then abnormal defaults to thrown(...) and normal defaults to !thrown(...). If only of normal or abnormal behavior is defined, the undefined one defaults to the negation of the defined one. There can be at most one definition each for normal and abnormal behaviors in an annotation.

Auxiliary Declarations

Sometimes, if the interface or class declaration does not contain enough information about the state, then it becomes necessary to have some auxiliary declarations for specification purposes. Since these are needed only for specification, but not a part of the original declarations, ADL/C++ provides a mechanism to declare them as auxiliaries.

auxiliary_declaration ::=
    auxiliary "{"
        (*C++_declaration*) * "};"

These will be useful in modeling abstract classes which are like interfaces and which typically do not have enough state information. In this case, the auxiliary declarations will provide a way to come up with an implementation, so that specifications can be written in terms of this implementation and testing of actual implementations can be done against this implementation.

A.3 Expressions

The expression language includes the side-effect-free subset of the C++ expression language. All forms of C++ assignment operators, increment and decrement operators and any overloaded versions of these are excluded. However, ADL/C++ does have the function-call operator. We provided this to facilitate testing. One should be very careful in using function calls as there can be side-effects in the invoked functions because the semantics of a function call in ADL/C++ is identical to the same function call in C++. Sometimes, it is possible to replace a function call with the corresponding **behavior** expression, as described in the previous section. In that case, it is advisable for the specifier to use the **behavior** construct, as it is implementation independent.

In addition to the above C++ operators, ADL/C++ also has some special operators of its own.

expression ::=
    C++_expression

| *adl_expression*

adl_expression ::=
group_expression
| *call_state_expression*
| *adl_logical_expression*
| *quantified_expression*
| *thrown_expression*
| *unchanged_expression*
| *behavior_expression*
| **return**
| **normal**
| **abnormal**

We will now describe the various ADL-specific expressions.

Group Expressions

group_expression ::=
"{" (*binding*) *
(*labels* { *tags* }
expression ";") * "}"

labels ::=
(IDENTIFIER ":") *

tags ::=
"[" IDENTIFIER
("," IDENTIFIER) * "]"

A group expression is a semicolon separated list of expressions. Each of these expressions should be a boolean expression and the type of the group expression is also boolean. The semantics of a group expression is the conjunction of the list of expressions that constitutes the group. Note that the expressions can be evaluated in any order and a ";" does not denote sequencing, but conjunction.

Expressions in a group can be given names using labels. The tags are used by tools for generating test reports.

Bindings

Since assignment expressions are not allowed in ADL/C++, a set of variables can be declared and initialized in a group expression using the bindings.

binding ::=
assign (*local_variables*) **with**
[IDENTIFIER ":="] *expression*

Thus, a binding has a list of variable declarations, optionally initializing one of those declared variables using an expression.

Call-State Expressions

call_state_expression ::=
"@" *expression*

A call-state expression in the behavior specification of a function is one which is evaluated before a call to that function. Naturally, it can not be applied to expressions like **return** since they have meaning only after a call to the function and using it with a value parameter has no effect. The type of a call-state expression $@e$ is the same type as that of e.

This can be useful to relate values of some global variables and/or reference and pointer parameters before and after a call to the function.

Logical Expressions

Even though C++ has logical expressions, they are short-circuited. So, ADL has special syntax for logical operations that are not short-circuited.

logical_expression ::=
  expression **and** *expression*
| *expression* **or** *expression*
| *expression* "==>" *expression*
| *expression* "<==" *expression*
| *expression* "<==>" *expression*
| *expression* "<:>" *expression*
| **if** "(" *expression* ")" *group_expression*
  (**elsif** "(" *expression* ")"
    group_expression) *
  { **else** *group_expression* } ";"

A logical expression is of boolean type and requires operands of boolean type.

The **and** and **or** operators have their usual logical meaning. The "==>" operator is the logical implication operator, the "<==" operator is the "implied by" operator and the "<==>" operator is the logical equivalence operator.

An expression like

if (c_1) $\{e_1;\}$
elsif (c_2) $\{e_2;\}$
...
elsif (c_n) $\{e_n;\}$
else $\{e;\}$

means that e_k should evaluate to *true* if k is the smallest integer such that c_k is *true* and if no such k exists then e should evaluate to *true* (if the **else** clause is present).

The "<:>" operator is the exception operator and it is used to relate the abnormal behavior definitions and exceptions thrown. Formally, the meaning of a <:> b is

a ==> **abnormal** &&
abnormal && b ==> a.

The meaning here is that if the left operand evaluates to *true*, then it denotes abnormal termination and in that case if the right operand evaluates to *true*, then the left operand must also evaluate to *true*.

Bounded Quantification

It is often required to give quantified formulae to describe behaviors of functions. The (type) domain of a quantified variable can sometimes be infinite in which case, testing becomes impossible. To facilitate testing, ADL/C++ provides universal and existential quantifiers over finite domains.

quantified_expression ::=
(**forall** | **exists**) "(" *domain_list* ")"
group_expression ";"

domain_list ::=
domain ("," *domain*) *

domain ::=
variable_declaration ":" *expression*

Here, **forall** and **exists** denotes universal quantification and existential quantification respectively. A quantified expression consists of a variable declaration, an expression that defines a finite domain (currently only integer ranges are allowed) and a group expression. The type of a quantified expression is boolean.

One of the advantages of having bounded quantification is that the expression can be evaluated by iterating over the finite domain. If the specification uses a precise range, testing can also be optimized. For example, consider a specification of a function that computes the gcd of two given numbers using unbounded quantification:

```
int gcd(int x, int y) semantics
{
   return % x == 0;
   return % y == 0;
   forall (int i) {
     i > return ==>
       return % x != 0 ||
       return % y != 0);
   };
};
```

This specification states that the gcd of two numbers is that number which divides both of them evenly and any number larger than that would not divide both the numbers. This specification might also be considered testable, that is, test for all allowed integer values MININT to MAXINT of the quantified variable. However, that is crude, imprecise and inefficient. In this particular example, the specification can be made more precise and efficient using bounded quantification :

```
int gcd(int x, int y)
semantics {
   return % x == 0;
   return % y == 0;

   forall (int i : int_range(
        return + 1, min(x, y))) {
     i % x != 0 or i % y != 0;
   };
};
```

This just uses the fact that a number can not be divided by another larger number. So, the bound for the quantification can be narrowed down to a smaller range by eliminating the (infinite) set of integers that are greater than the maximum of the two parameters x and y.

Thrown Expressions

This is used to check if a particular exception is thrown by the function call.

thrown_expression ::=

thrown "(" *exception_names* ")"

exception_names ::=
IDENTIFIER ("," IDENTIFIER) *
| "..."

The type of a **thrown** expression is boolean. Each of the identifiers should be declared as an exception in the exception list for this function.

This expression evaluates to *true* if an exception corresponding to the type of each of the identifiers is thrown. Note that multiple exceptions cannot be thrown by a C++ function. However, because of subclassing, a derived class exception can also be considered as an exception of a superclass type. If "..." is used, then the expression evaluates to *true* if there is some exception thrown.

This expression is commonly used with the exception operator to specify what exception can be thrown under what conditions in case of abnormal behavior.

Unchanged Expressions

This can be used to specify expressions whose values are the same before and after a call to the function.

unchanged_expression ::=
unchanged "(" *expression_list* ")"

expression_list ::=
expression ("," *expression*) *

The type of an **unchanged** expression is boolean. It evaluates to *true* only if the values of all the expressions listed remain unchanged before and after a call to the function in whose **semantics** clause the expression appears.

Other ADL Expressions

The other ADL keywords **return**, **normal** and **abnormal** denote the return value of the function call, and the predicates corresponding to normal and abnormal behavior for that particular function respectively. Since these are properties of the termination of the function, they can not be used in the call-state.

A.4 Preprocessing

A header file consisting of function and class declarations can be included using the standard C++ preprocessor directive ("*#include*") in an ADL/C++ specification file for giving annotations. This is analogous to the way function implementations are developed in C++. To facilitate this, an ADL/C++ specification is preprocessed using some C++ preprocessor before it is processed by any of the ADL tools.