Preliminary Design of ADL/C++ -- A Specification Language for C++

Sreenivasa Rao Viswanadha
SUNY Albany

Preliminary Design of ADL/C++ — A Specification language for C++

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

Sriram Sankar
Sun Microsystems Laboratories
2550 Garcia Avenue
Mountain View, CA 94043, USA
sankar@eng.sun.com

Abstract

ADL/C++ is a specification language for associating behavior specifications as post-conditions with C++ (function) declarations. The language includes a clean subset of the C++ expression language and the semantics is relatively simple and informal. Most of the important features of C++ like member functions, virtual members, constructors, inheritance and exceptions can be specified in ADL/C++. In addition to being semi-formal documentation, ADL/C++ specifications can also be used for testing C++ implementations. We developed a method for validating tests of C++ implementations using ADL/C++ specifications.

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, 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 [9] uses set-theoretic semantics for specifications and Larch [3] and Tecton [4] are languages based on algebraic specification. But, most imperative programming languages have state-based operational semantics. Perhaps this is the reason why there does not seem to be much use of these specification methodologies in the industry.

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

A somewhat different approach has been taken in high-level functional and logic programming languages where a single language framework is used for specification and implementation. Our approach has similar goals but is still quite different from these. 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 different languages, possibly supporting different paradigms. While writing specifications the programmer is encouraged to specify *what* the program does without worrying about implementation details, and for this, a small "clean, *side-effect-free*" subset of the language can be used.

We believe that such an approach will encourage developers to write specifications before working on implementations. Such specifications can be easily used for testing purposes also, since 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.

ADL (Assertion Definition Language) adopts this philosophy. It is a family of languages designed to associate semantic information with declarations of variety of languages (C, C++ *etc.*). One of the basic goals of the language is to remain as close to the implementation language as possible, at the same time allowing high-level specification independently of implementations. Another feature of ADL is that specifications written in ADL are checkable, *i.e.*, implementations can be tested against ADL specifications.

There are some inherent limitations with this approach, particularly, that the specification will inherit all the pitfalls and ambiguities of the programming language depending upon the restricted subset of the language being used for specification purposes. But, we have taken care to select a subset of the underlying programming language which is well-understood with minimal ambiguities.

The rest of the paper is organized as follows. In Section 2, we present some background on ADL. Section 3 presents an overview of of ADL/C++ using an example. In Section 4, a somewhat detailed description of the language syntax and informal semantics is given. Section 5 describes how reuse of specifications is supported in ADL/C++. In Section 6, a method for validation of C++ implementations using ADL/C++ specifications is presented. Section 7 gives a comparison with other related work. We conclude in Section 8 with a brief discussion about future work.

2 Assertion Definition Language

ADL is a specification language framework used to describe input-output behavior of software systems. ADL originated as part of a conformance testing system (ADLT) [11] for ANSI C API's. In this system, the runtime behaviors of implementations of API's are compared to their specification written in the ADL framework. These specifications are written in a C like syntax as extensions to C header files. This particular syntax is termed ADL/C (or ADL for C). The ADLT project commenced in early 1993 and has culminated in a robust industry quality environment for conformance testing.

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.

We are developing a similar specification capability for C++, namely ADL/C++. ADL/C++ builds on the experience gained from the ADLT project. It is a natural 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 we anticipate that this subset will eventually replace the old ADL/C[1] . Other projects currently being undertaken are ADL/IDL (OMG IDL [2]) and ADL/Java™. While this paper focuses on ADL/C++, the general concepts presented here apply equally well to the other language specializations of ADL.

3 Overview of ADL/C++

ADL/C++ supports behavior specifications for methods (member functions) and constructors in the form of post-conditions. It also supports specification of inheritance, virtual functions, and exceptions. For this, it uses a *side-effect-free* subset of C++ expression language as described in [10] and extends it with the usual logical operators, bounded quantification, call-state operator (for evaluating expressions prior to a function call), return expressions (to denote return values of functions), and some structuring constructs.

A function behavior is typically partitioned into normal and abnormal behavior by giving the corresponding predicates in terms of the values of the parameters, return values, and exceptions thrown. Then, predicates that describe the state transformations corresponding to each of these behaviors are given. Additional predicates that describe the state transformations and invariants independent of normal or abnormal behavior may also be given.

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);
   };
};
```

Intuitively, the above specification says 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. Albeit using C++-like expressions, the above specification just says *what* the function *sqrt* should do, but not *how* it should be done[2].

The operational meaning of the above specification is that if *j* is the return value of a call *sqrt(k)*

[1]Upward compatibility will be maintained so that the existing ADL/C specifications can work with the new set of tools.

Java™ is a trademark of Sun Microsystems Inc.

for some (positive) integers j and k, then both the formulas listed in the semantics clause should evaluate to *true* by setting the value of the variable x to k and that of **return** to j. More precisely, the following two predicates should evaluate to *true* :

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

The "@" operator says that the expression following it should be evaluated before a call to *sqrt*. The keyword **return** denotes the return value of a call to *sqrt*.

The complete syntax and semantics of the language are presented in Section 4.

We illustrate the specification of various important features of C++ using the more realistic example presented in Figures 1 and 2 of a specification of a bank account class. The account class has methods to deposit and withdraw amounts. The declaration of the *Account* class can be first developed in a file *account.hh* and then, it can be included in the specification file to give behavior specifications for the various (member) functions.

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

The *Account::Withdraw* method semantics says that the operation terminates abnormally if either of the exceptions corresponding to insufficient funds in the account or a request to withdraw a negative amount is thrown. Otherwise it terminates normally and the value of *balance* decreases by the amount withdrawn. Then it also says what the values of a thrown exception should be.

The constructor specifies what the initial values of various data members should be in case of normal termination. It also specifies the exceptions thrown in case of illegal initial values for some of the data members.

[2] This is a partial specification because no return value can satisfy the behavior specification if the input value of the parameter x is zero or a negative number. This can be easily fixed using exceptions.

3.1 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 typically relates the before and after values of the part of the state that it can modify in a particular call. This includes all the global variables, data members (for member functions) and any reference parameters. In addition, it can also specify what exceptions, if any, it can throw and the properties of the values of the exceptions thrown.

A behavior specification for a function can be given inline at the time of declaration, or separately, just like a definition.

Constructors

In semantics for constructors, typically one can specify the initial values for various data members which effectively gives 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.

Virtual Member Functions

If a superclass declares a member function to be virtual, then a call to it using a pointer might actually invoke its implementation in a subclass. Therefore it is logical that semantic description given in a subclass implies the semantics specified in the superclass for a virtual function.

To achieve this, ADL/C++ imposes the semantic restriction that any 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. This means that the assertions in the derived class are strengthened by those given in the superclass. This, in some sense extends the (mostly syntactic) inheritance mechanism of C++ to inheritance of properties of methods rather than respecifying the properties given in the superclass.

For example, consider the ADL/C++ specification given in Figure 3, of a new bank account class

```
#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.
      struct InvalidAccountNum {
         const int num;
         InvalidAccountNum(int i) : num(i) { };
      };

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

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

      struct 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

```
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

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

Figure 3: A Specification Using Inheritance and Virtual Functions

which inherits from the *Account* class. It allows an overdraft upto a certain amount which can change. Therefore the new *Withdraw* method will say 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.

Another 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++.

This semantics can also be useful to specify the behaviors for specializations if the superclass specification is not overly constrained. For example, the sorting functions specified in Section 5 can be easily specified using C++ inheritance 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 say anything about the relative order of elements with equal key and hence the subclass has the freedom to add the extra constraint.

The idea here is to impose some discipline on inheritance (which C++ doesn't) so that some kind of subtyping is achieved. Our current semantics for virtual functions is inspired by the behavioral notion of subtyping described in [6] and a similar scheme called *weak behavioral subtyping* is supported in Larch/C++ [5].

3.2 Exceptions

ADL/C++ allows the specification of exceptions that can be thrown by a function. In the semantic specifications of functions, exceptions can be explicitly used (see the next Section). C++ does not allow exceptions to be named in the throw clause. However, in ADL/C++, they can be named just like other parameters, and can be used in the postcondition. We feel this will make specifications more readable than the corresponding C++ code because now users can name exceptions in a way as to reflect the nature of the exception. In the bank account example above, instead of using separate exceptions, if the class designer decided to throw an exception of type *long* in case *amount* is negative, it can be named appropriately (say, *negativeAmount*) to reflect this fact in the behavior specification. We also extended the C++ exception declaration to specify things like no exception can be thrown and any exception can be thrown.

The semantics of exception specification is that any implementation can throw *at most* those exceptions listed in the throw clause. It can not throw anything that is not mentioned. Note that in C++, the throw list that follows a function declaration is just informational and the function can throw other types of exceptions not listed there.

In the post-condition, tests can be made for the presence of exceptions using the **thrown** operator. Exception values can be used just like normal data values except that it is illegal to use an exception when it is not thrown. Similarly, it is illegal to use an exception in the call state. It is also illegal to use the return value in case an exception is thrown.

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

Note that 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 used as part of normal behavior also. For example, a read method of a file class may throw an exception to signal an end-of-file. But, usually it is not abnormal behavior. On the other hand, if the file is not open, then the exception thrown usually signals abnormal termination.

3.3 Access Control

We envisage the use of ADL/C++ for giving detailed specifications for implementations also, therefore, ADL/C++ ignores all access restrictions. This, we believe, will give the implementor flexibility to specify and test the implementations without having to worry about access permissions. This gives the ability to look at (but not to modify) private data not accessible to a function to do test validation.

4 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 detail in the following subsections.

4.1 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.

4.2 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 ";") + "]"

As mentioned earlier, 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).

An annotation for a function is like 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 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*) * "};"

4.3 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. 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 reports in case of testing.

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.

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.

This says that if the left operand evaluates to *true*, then it denotes abnormal termination. It also says that in case of abnormal termination, if the right operand evaluates to *true*, then the left operand must also evaluate to *true*.

Bounded Quantification

ADL/C++ supports 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.

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.

4.4 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.

5 Reusing Specifications

Sometimes the behavior (post-condition) of a function might just be a simple refinement of an existing function (behavior). But, ADL does not provide a way to specify this refinement directly. In stead, there is a more general construct to use the semantics (behavior specification) given in ADL/C++ of a function while giving the semantics of other functions.

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

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

The behavior operator requires a function name and values for parameters and a value to denote the return value of a call to the function with those values. The actual parameters might also include (a pointer to) the object in the case of member functions. The syntax is very close to a function call, so for members, the object (or pointer) is supplied using the appropriate member selector operator.

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*.

Consider the following specification of a *Sort* function :

```
specification SortSpec
{
   typedef Record RecordArray[100];

   RecordArray Sort(RecordArray records)
   semantics {
      // First say it is a permutation
      IsPermutation(return, records);

      // Then say that the return value
      //  has keys in nondescending order
      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 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 postcondition 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];
            };
         };
      };
   };
};
```

As the expression language of ADL/C++ includes C++ expression language, a function call in a specification refers to its implementation, not the specification. Since some function implementations can have side-effects, it may not be correct to invoke function implementations. In those cases, behavior expression can be very useful.

Another situation where this will be useful is when one class contains objects of some other class and uses the contained class methods to implement its own methods. Then the specifications cab also be written using 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 {
         assign Account *acPtr with
              acPtr = GetAccount(acNum);
         if (normal) {
            behavior(
               acPtr->Deposit)<amt>;
         };
      };
   }
};
```

This specification just says that the bank's deposit method merely locates the account corresponding to the given account number and then the amount is deposited into the located account. It also throws an exception if it can not find an account with a 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 it is properly implemented).

6 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.

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 part of the function's behavior specification.
- 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 behavior specifications.
- 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 normal or abnormal behavior, if not there is something wrong with the implementation, that is, it is either returning a value or throwing an exception that it is not supposed to. In addition, a test might fail because it did not cause the appropriate state change that it is supposed to cause.

For example, consider the following (wrong) 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;
}
```

As it can be seen, the *Account::Withdraw* method is wrong 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 supposed to throw and exception.

Now, any test for the *Withdraw* method with a nonnegative amount will make its post-condition false, and thus can be classified as a failure. Even though this may be a simple bug to catch, there might be other examples where there might be 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).

Thus, an ADL/C++ specification gives some kind of formal basis for test validation rather than the tester deciding whether the test failed or succeeded based on some 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.

Since ADL/C++ is very close to C++, it is relatively easy to generate code from ADL/C++ specifications that can used for automating the test validation process. 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 which can help in quickly locating the problem in the implementation that caused the test failures.

When testing using ADL/C++ specifications, there could be calls to constructors, assignment operators and copy constructors. Hence, users should keep these relatively simple and clean, otherwise, the testing might become inefficient and imprecise.

At present, the test data has to be supplied by the user. We are currently working on coming up with a language like TDD [8] 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*.

7 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++ [5]. 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.

8 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 :

- *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 real software engineer. While this has reduced the expressivity of the language to some extent, 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 verify 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. In general executable specifications impose undue restrictions on specification constructs to simplify the process of generating executable code. 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 design of the language is not yet complete. We merely extended the ADL/C-style postcondition specifications for functions to specify class method behaviors with some obvious semantic constraints for virtual member functions.

In C++, classes can also be considered as the basic programming units in addition to global functions like in C. So, it would be nice if class behaviors can be specified and *tested* just like function behaviors.

Specification of (methods of) abstract classes is difficult at present. This is because, usually abstract classes do not have any implementation (or state) information that can be used for the post-conditions. So, the only way to do this at present in ADL is to come up with some crude implementation (using the auxiliary declarations) and writing the post-conditions using that. This makes specifying abstract classes complicated.

One way to solve these problems is to adopt an algebraic specification framework, but it is not clear how testing can be supported in that framework.

Considering these issues, it appears some kind of trace-based specifications on top of the current ADL/C++ specifications would be a nice way to capture class behaviors, at the same time preserving the two defining features of ADL - testability and closeness to the base implementation language. We have some preliminary ideas in this respect and we will be incorporating them into ADL/C++ soon.

Availability

The development of the tool is currently in progress. The initial version will be available in about a year. Contact the authors for further information.

Acknowledgments

We would 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. We would also like to thank Deepak Kapur for his invaluable suggestions for improving the overall quality of the paper.

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] 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.

[4] 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.

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

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

[7] 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.

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

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

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

[11] 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.