Using Annotated C++

Marshall P. Cline

Department of Electrical and Computer Engineering, Clarkson University

Doug Lea

Department of Computer Science,
SUNY Oswego
and
CASE Center, Syracuse NY

ABSTRACT

A++ (''Annotated C++'') is both a formalism and a proposed CASE tool for annotating C++ code with object-oriented specifications, assertions, and related semantic information. Annotations provide programmers with a useful means for approaching class design, exceptions, correctness, standardization, software reusability and related issues in software engineering with C++. This paper shows how A++ provides arbitrarily fine granularity to the C++ type system, how it automates and streamlines exception testing, how it can aid in standardization of software components, and how it can safely remove redundant exception tests.

1. Introduction

A++ (''Annotated C++'') is presented in [1] as both a formalism and a *proposed* CASE tool for extending C++ with object-oriented annotations. In the present paper, we briefly describe A++ (mainly by example), and then discuss how A++ can be of practical utility in the construction of well-specified, correct, robust, efficient, standardized, and reusable C++ classes.

As a formalism, A++ provides a syntax for expressing a variety of semantic constraints and properties on C++ classes. It is roughly similar to the Ada annotation tool ANNA [2], as well as Eiffel [3] in that it extends the base type system to support several forms each of preconditions, postconditions, assertions, and invariants, along with a richer set of ''primitive'' types and constructs that are useful in expressing such constraints. It differs from non-object-oriented systems like ANNA in explicitly addressing issues like classes and inheritance. It differs from Eiffel in addressing C++-specific issues and constructs, and in maintaining the spirit of C++ in terms of efficiency, flexibility and terseness. All three differ from specification languages like VDM [4] in that they attempt to integrate specifications with the implementation language, thus providing programmers with a more familiar and natural syntax.

As a proposed CASE tool, A++ is intended to be used as a front end to a C++ compiler. It will translate annotated programs into C++ proper while also statically verifying, to the extent possible, the conformity of code to annotations. As described in more detail below, static analysis is a powerful tool for helping to ensure correctness, efficiency, and robustness of C++ classes.

2. A++ as a Type System

Types may be construed as sets of constraints on objects [5]. In C++, it is easy to specify ''the type of an object constrained to take on integral values between 0 and 255,'' via the built-in type `unsigned char`. However, C++ provides no direct support for expressing things like ''the type of an object with two components, the first taking integral values between 1 and 12, and the second between 1 and the results of applying function `days_in_month` to the first component.'' Similarly, while it is possible in C++ to

declare a member function `reset` that takes an `int m` and returns `void`, it is difficult to state that `m` must obey a given predicate, and that calling `reset` will have a particular visible effect. A++ provides such support, as in

```
class Calendar {
  int mon, date;                        //month and date
  int days_in(int month) { /*...*/ }  //returns #days in month
legal:
   mon >= 1 && mon <= 12;              //mon in 1..12
   date >= 1 && date <= days_in(mon); //date in 1..#days(mon)
public:
   int  month() const { return mon; }
   Calendar() : mon(1), date(31) { /*...*/ }
   void reset(int m);    // Reset Calendar to month 'm'
axioms:
   [int m; require m>=1 && m<=12; promise month() == m] reset(m);
   //...
};
```

As described in more detail in [1], the `legal:` keyword introduces invariants for data members of a class, `require` indicates preconditions, `promise` indicates postconditions, and `axioms:` introduce pre-condition/postcondition sets that may simultaneously constrain several member functions.

As another example, while C++ supports subtyping via inheritance, it is difficult to express in a base class those constraints which must be met in all subclasses. This especially important in the declaration of annotated "Abstract Base Classes" (ABCs), supported in A++ via additional constructs:

```
class Stack { // An annotated ABC; Assumes element type T.
public:
  virtual void push(T)          = 0;
  virtual T    pop()            = 0;
  virtual int  len()      const = 0;
  virtual void clear()          = 0;
  virtual int  capacity() const = 0;
  virtual     ~Stack()          { }
               Stack()          { }
          int  full()     const { return len() == capacity(); }
          int  empty()    const { return len() == 0; }
axioms:
  // An incomplete axiom set addressing only empty/full issues
  [ require !full(); promise !empty(); ] push(x);
  [ require !empty(); promise !full()  ] pop();
  [ promise return >= 0                ] len();
  [ promise empty()                    ] clear();
  [ promise return > 0                 ] capacity();
  [ promise empty()                    ] Stack();
};
```

```
class Vector {        // A standard Vector class
  int cap;            // Capacity of this Vector
  T*  data;           // Actual data elements
coherent:             // Means 'required when in a public state'
  cap == data.nelems; // nelems is a pseudo-member of arrays
public:
  //...
};

class VStack : public Stack {   // 'Vector-based Stack'
protected:
  Vector v;        // A VStack USES-A Vector
  int sp;
legal: sp >= 0 && sp <= v.size();
public:
       VStack(int cap=10) : v(cap), sp(0) { }
      ~VStack()   { }
  void push(T x)  { v[sp++] = x;     } // INHERIT Stack's axioms
  T    pop()      { return v[--sp];  } // Thus a VStack must
  void clear()    { sp = 0;          } // 'behave like a Stack'
  int  len()      { return sp;       }
  int  capacity() { return v.size(); }
};
```

Again, more syntactic details and discussion may be found in [1].

The fact that such type extensions are not typically found inside languages like C++ is almost always a pragmatic issue. While it is a routine matter for a compiler to statically analyze a C++ program to ensure that functions expecting arguments of type `char` actually receive them, determining whether objects of class `Calendar` always remain within their `legal` constraints, or that all uses of `VStack::push` obey the given preconditions and postconditions is very difficult indeed. It requires the use of inference engines that can possess high overhead without any guarantee of success in statically deciding conformance.

In fact, C++ is among the most difficult languages to statically verify, mainly due to insecurities in the underlying C type system (pointer coercions, ambiguities between arrays and pointers, aliasing, etc.). While A++ includes constructs encouraging programmers to avoid most of these problems, the prospects of *complete* static C++ program verification, under any system and/or theorem prover methodology are dim.

Despite this, annotations are extremely valuable constructs for increasing the expressiveness and semantic content of C++ declarations and enhancing the integration of class specification with class implementation. For example, A++ permits essentially direct translation of common Abstract Data Type (ADT) specifications [6] into ABCs, thus increasing the likelihood that such classes are written correctly *by construction*.

Moreover, even with modest inferencing capabilities, an annotation tool *will* be able to verify many common constructs, and can postpone others via translation of annotations into exceptions (or merely compile-time warnings), as discussed below. In this fashion, the advantages of the A++ formalism may be obtained while remaining within the reaches of implementability and practical usability.

A++ may be viewed as an enhanced type system for C++, designed with similar features and goals as the C++ type system itself, but permitting programmers to supply arbitrarily detailed semantic information, thus serving as an integrated specification language. Until the A++ tool is developed, programmers may find the use of A++ constructs as ‘‘structured comments’’ to be a useful stylistic aid in the design and implementation of C++ classes.

3. A++ and Exceptions

Pragmatic software engineering concerns demand a disciplined and liberal use of exception tests and handlers in order to ensure robustness and correctness. While exception handling features will certainly become incorporated into C++ in the near future, their precise form has not yet been established [7].

However, the mere existence of exception constructs is not a cure-all. Programs laden with exception checks can become more complex and harder to read, and can possess higher time and space overhead. A++ offers an easier, more structured, and potentially more efficient means for programmers to utilize exceptions, that is nearly independent of the exact C++ mechanisms chosen to implement exception processing.

The vast majority of exception tests inside class member functions revolve around the trapping of failed preconditions. However, this usage is completely subsumed under A++, where preconditions become ''part of the type'' of member functions.

This strategy has several advantages over the raw use of exception checks. Under A++, constraints are *automatically* translated into validity checks (unless statically shown to be superfluous), ensuring that all declared preconditions are consistently and completely applied wherever applicable. For example, there is no explicit emptiness test at the head of `pop` in the `VStack` example above. Declaring preconditions *per se* rather than burying them inside member functions via exception tests tends to increase the clarity of class declarations and definitions.

Moreover, direct incorporation of preconditions allows for the possibility of static analysis of validity checks, which can result in significantly faster executable code. In this sense A++ can serve as an ''exception optimizer.'' If static analysis shows that a particular instance of a validity check will *never* fail in a particular program, then the check, and the corresponding exception raising support, need not appear in the generated code. In the best case, A++ will insert exception checks only where they really matter, i.e., where there is sufficient indeterminacy of usage to indicate that an exception check could actually fail during program execution. While doing so, A++ will provide programmers with feedback about which preconditions sometimes, or even always, fail, in the same fashion and spirit (i.e., to aid in the early detection of programming errors) that C++ compilers provide warning and error diagnostics for potential and actual type mismatches.

In these ways, A++ may become an important means for assisting programmers in minimizing classic safety versus efficiency trade-offs. The need for efficient mapping of clean, safe designs is an increasingly vital issue as the use of exceptions becomes more prevalent. A good exception optimizer can remove the motivation for questionable C++ techniques involving excessive numbers of `friend` functions for the sole purpose of providing unrestricted access in ways that are known to be safe by the programmer but not by a C++ compiler, the use of designs that break encapsulation, and the omission of validity checks, which are all commonly, and often understandably, invoked for the sake of efficiency, but which are all unnecessary under a good static analyzer.

This issue is important enough that careful consideration, evaluation, and experimentation to determine best available techniques for integrating, analyzing, and translating precondition annotations has become one of the central objectives in the development of the A++ tool. In principle, prospects for generating highly efficient code are bright. Optimizability and strong typing are often just two different views of programmer-provided semantic information inside programs. Constructs that enhance one can only help the other.

4. A++ and Standardization

As C++ becomes increasingly widespread, people have expressed the need for standardizing common components like `Strings`, `Vectors` and `Sets`.

The notions of ''standards'' and of ''annotations'' are closely interwoven. Standards most often take the form of specifications that do not make reference to matters of representation or implementation. This kind of specification technique is directly supported in A++.

Annotated ABCs (like the elided `Stack` example, above) are object-oriented analogs of ADTs, containing representation-independent specifications of the behavior of base classes that can be implemented

via subclasses using any of a number of representation and/or coding strategies. As such, they are appropriate bases for standardization efforts.

While it is sometimes more difficult to spell out high-level specifications using A++ rather than natural language or unintegrated specification tools (e.g., VDM), there are some clear advantages for using A++ in standardizing C++ classes. Annotated ABCs may be standardized upon and specified within the language itself (as extended via A++), rather than via auxiliary documentation. This tactic both formalizes and simplifies creation of standard base classes, helps automate compliance monitoring, and helps avoid questions of interpretation, while still allowing component providers the freedom to supply implementation subclasses that employ arbitrarily diverse coding strategies. The fact that all implementation subclasses share the same ABC parent guarantees interoperability, while the fact that they are subclasses allows providers to include additional capabilities, beyond those mandated, within their implementations.

The only drawback to this solution is the efficiency issue. The use of ABCs can generate significant performance degradations compared to representation-specific classes. However, the A++ tool, along with the customization-based compiler strategies and language support described in [8] have the potential for eliminating these degradations. Pending the development of such support, the use of the A++ syntax as a notational device may still be an attractive vehicle for standardization efforts.

5. A++ and the Construction of Reusable Software

Of course, methods useful for standardization are equally, if not more appropriate for the development of component libraries in general. A++ is valuable both in design *for* reuse and design *with* reuse of C++ class libraries. Beyond the obvious fact that well-specified, correct, robust, and efficient classes are almost by definition highly reusable, the use of annotations supports techniques that appear to generally enhance reuse.

By using annotated ABCs that separate specification from implementation (while still, of course, remaining within C++) component writers may provide additional opportunities for reuse: (1) Direct reuse of an implementation subclass; (2) Further subclassing of the implementation subclass to add or modify implementation code; (3) Reuse of the ABC, but implemented via a new implementation class; and (4) Subclassing of the ABC to create a new abstract subclass with extended behavioral specifications. In other words, this strategy allows design reuse to occur nearly independently of code reuse, as advocated by researchers (e.g., [9]) studying software reuse in general.

Integrated semantics also increases the ability of potential class users to locate, understand, and use library classes. Via A++, most of the important information associated with a class resides in its declaration, simplifying search, retrieval, browsing, and use.

Of course, specifications are supplements, *not* replacements for testing library components. Specifications can be wrong, fail to conform to requirements, and so on. Additionally, as discussed above, even the most sophisticated tools may fail to verify correct code. However, additional tools and methods based on the A++ notation appear to be promising approaches to C++ component testing. For example, Frankl's [10] tool for generating test programs by analyzing axiomatic class specifications in order to generate two different paths (sequences of method calls) that place objects in the same alleged state, could be adapted to read A++-based specifications.

6. Migrating Exception Tests to the Caller

The term ''exception'' is used here to denote the occurrence of an unanticipated event rather than an intentional non-local jump. In general ''exceptions'' actually occur very infrequently. Indeed, the logic and sanity checks in a correct program are designed to intelligently handle errant user input so that (ideally) an unexpected failure will *never* occur. In contrast to this low frequency of actual exceptions, exception *testing* is ubiquitous.

Callers of a function can't ''see'' what tests are just inside the called function, so the compiler is unable to jump around the test in a subcall even if it is superfluous. For example, bounds testing in a ''safe array'' subscripting operation might be performed at the head of `operator[]`, but unless `operator[]` is expanded inline (and unless the C++ compiler has an optimizer that can exploit the resulting information), even accesses such as `array[0]` will incur an unnecessary runtime penalty for the bounds check.

```
Vector operator+ (const Vector& a, const Vector& b)
{
   if (a.size() != b.size()) error("sizes not same");
   int sz = a.size();
   Vector ans(sz);
   for (int i = 0; i < sz; ++i)
      ans[i] = a[i] + b[i];
   return ans;
}
```

Although the bounds tests will *provably* never fail in the above function, `i` is still checked three times every time around the loop! Unfortunately this kind of thing is by no means unusual: Stroustrup describes it as ‘‘typical’’ and proposes either an ‘‘unchecked access’’ function or else using the `friend` construct [11]. Neither approach is ideal: the former breaks abstraction and the latter dilutes localization by increasing the number of functions which can directly manipulate instances.

A++ provides a different solution: since the preconditions for `operator[]` *are* visible to the caller, all the above runtime checks can be eliminated. The key concept is to migrate exception testing from the head of the subcall out to the caller. For example, if `pop()` requires `!empty()`, conventional C++ wisdom says that `pop()` might begin: `if (empty()) throw StackError()` (assuming some such exception syntax). As will be shown, migrating the test out to the caller is merely conceptual, all the details being taken care of by the A++ preprocessor.

Suppose `main()` calls `pop()`, the latter requiring `!empty()`. Under A++, `main` will wrap the call in a test: `if (empty()) throw StackError(); else pop()`. Since the `Stack` will *never* be `empty()` at once inside `pop()`, A++, in conjunction with a back-end optimizing compiler, will be able to discard as redundant any conventional `if (empty())` tests found there. As discussed above, few (ideally *no*) subcalls can generate exceptions, so few tests will in practice be performed.

The disadvantage of migrating exception testing to the call point is code bloat: if a function is called many times, its precondition tests might be elaborated many times. It is very difficult in general to beat the fundamental tradeoff between speed and size, however a *wrapper* function can help in this case. For example, if code size were more important than speed, A++ could create `popwrapper()` which would perform the precondition tests and call the ‘‘raw’’ `pop()`. If the output were assembly language, the ideal situation would be multiple entry points to the function:

```
proc  popwrapper__5StackFv
  ...          ;Do precondition tests
proc  pop__5StackFv
  ...          ;Raw function -- no tests
  ret
```

Special consideration must be made in multiple language environments since only C++ would know about A++. The ‘‘foreign language’’ callers should get the fully-tested version when they call `pop()`, so the names presented above can be swapped:

```
proc  pop__5StackFv
  ...          ;Do precondition tests
proc  popraw__5StackFv
  ...          ;Raw function -- no tests
  ret
```

We have come full circle. The above code fragment is *exactly* what is produced by current C++ technology, the only difference being the additional (`raw`) entry point. But in conventional compilers, only the *fully tested* function is ever called, so *all* calls incur the runtime penalty. Thus A++ safely allows (hopefully many) calls to ‘‘jump around’’ the runtime tests. Furthermore, since the only additional ‘‘op code’’ is the `jmp` around the stack setup, both speed and space overhead is negligible.

Of course, all of these code-generation options are very tentative, and will be further explored during the process of implementing a usable version of A++.

7. Conclusions and Future Work

Although essential for robust software, disciplined and liberal use of exception testing is often seen as pessimistic, increases code size, decreases efficiency and can make code harder to read. The result is often that programmers fail to decorate their code with such checks. We have proposed a CASE tool (''A++'') which infers exception tests from higher level annotations rather than requiring exception tests to be explicitly stated in the code. The programmer can then instruct A++ to either translate all annotations into run-time exception tests or to perform static analysis whereby tests which are shown to be redundant need not be inserted. It is hoped that minimizing superfluous run-time checks will help programmers avoid the temptation to manually remove exception tests in ''tested'' software.

A++ was also discussed as a means of aiding efforts to standardize software components since it provides a formal yet convenient language for expressing the appropriate semantic information.

A ''proof of concept'' implementation will translate A++ syntax into an intermediate form, insert exception tests everywhere an axiom or assertion directs, and translate the result back into C++ for compilation. It is hoped that the intermediate reference form (IRF) will be a significant spinoff, since the IRF could be the basis of an entire suite of C++ analysis tools. Several extensions to A++ are also being considered, including the handling of pointer aliasing and reference binding/lifetime issues, as well as allowing behavioral compatibility to exist apart from inheritance.

8. 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] D. Luckham, F. von Henke, B. Krieg-Bruckner and O. Owe, ''ANNA: A Language for Annotating Ada Programs,'' Springer-Verlag, *Lecture Notes in Computer Science 260*, 1987.

[3] B. Meyer, *Object Oriented Software Construction*, Prentice Hall, 1988.

[4] C. B. Jones. *Systematic Software Development Using VDM*, Prentice Hall, 1986.

[5] P. Wegner. ''The Object-Oriented Classification Paradigm.'' In B. Shriver and P. Wegner *Research Directions in Object-Oriented Programming*, MIT Press, 1987.

[6] B. Liskov and S. Zilles, ''Programming with Abstract Data Types.'' *SIGPLAN Notices*, 1974.

[7] A. Koenig and B. Stroustrup, ''Exception handling for C++.'' *Proceedings of the USENIX C++ Conference*, 1990.

[8] D. Lea, ''Customization in C++,'' *Proceedings of the USENIX C++ Conference*, 1990.

[9] W. Tracz, ''The Three Cons of Software Reuse,'' *Proceedings of the Third Annual Workshop: Methods and Tools for Reuse*, 1990.

[10] P. Frankl and R. K. Doong, ''Tools for Testing Object-Oriented Programs,'' *Proceedings of the Pacific Northwest Quality Assurance Conference*, 1990.

[11] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.