- If the current lexical scope doesn't have privileged access to the type T of the quantified variable, the annotation is a client of object v, so v is automatically constrained to be coherent (and therefore legal as well). If the current lexical scope does have privileged access to the type T, v is merely constrained to be legal.

- The scope of a quantified variable extends up to the closing "]" of an assertion, or through the end of the annotated statement in an axiom. Thus quantified variables may not appear within an assertion's statement (ie: when the statement is translated into run-time code), but they may (and often do) appear with an axiom's statement.

6 Conclusion

- *Optimization:* The very existence of data encapsulation, essential to the object oriented paradigm, uncouples clients from objects. Although "beautiful" from the perspective of localization, this makes it difficult for objects to know which calls require dynamic argument testing. A++ identifies, on a call-by-call basis, which exception tests are really necessary, removing the others.

- *A More Natural Programming Style:* Safe, exception-test laden code is often hard both to write and to read. In most cases A++ should allow code that is at least as safe to be written in ways that do not sacrifice, and will usually enhance readability, clarity, and reusability.

- *Future Work:* The "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.

References

[1] Joseph Goguen, Principles of parameterized Programming, in T. Biggerstaff and A. Perlis (Eds.) *Software Reusability*, ACM Press (1989).

[2] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Bruckner and Olaf Owe, ANNA — A Language for Annotating Ada Programs, Springer-Verlag, *Lecture Notes in Computer Science* 260 (1987).

[3] Bertrand Meyer, *Object Oriented Software Construction*, Prentice Hall, (1988).

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

[5] Peter Wegner, Dimensions of Object-based Language Design, *OOPSLA'87 Proceedings* (1987).

5.4 Basic Annotation Forms

The basic form for both axioms and assertions is: `[`*annotations*`]`*stmt* where *stmt* is a (possibly compound) C++ statement. The following are the allowed annotations:

Named States: `[State` *s*`]` creates a state named *s* which may be referred to at a later point. This must appear immediately after the opening "`[`". `[State init]` is automatically created for all function axioms (but creating a named state in a function axiom overrides this).

Quantified Variables: `[`T v`]` creates a universally quantified compile time "variable" v of type T (as in $\forall v \in T$). Existentially quantified variables ($\exists v \in T$) are declared using the A++ keyword "`exists`": `[exists` T v`]`. Variables must be declared (quantified) before used.

Purity: After execution of a `pure` function, clients of a type can't detect any change of state. For example, after pushing anything onto a stack and popping it off again, clients can't detect any change of state, although the element just beyond the top of the stack might have changed if the implementation is vector based.

Preconditions: `[require` *expr*`]` expresses the preconditions of the annotated statement. In a function axiom, the *expr* is recorded as the function's precondition. In an assertion (an annotation in executable code), *expr* is required to be true at the point it appears. `[require` *e1*`; require` *e2*`]` is equivalent to `[require` *e1* `&&` *e2*`]`. If `[State` *state*`]` is in scope, the *expr* may contain "*state.x*" which is the value of *x* at the moment that *state* was declared.

Postconditions: `[promise` *expr*`]` expresses the postconditions of the annotated statement. In a function axiom, *expr* is recorded as the function's postcondition, and `return` may be used to indicate promises about the function's return value. In an assertion (an annotation in executable code), *expr* is required to be true at the end of the annotated statement. `[promise` *e1*`; promise` *e2*`]` is equivalent to `[promise` *e1* `&&` *e2*`]`. If `[State` *state*`]` is in scope, the *expr* may contain "*state.x*" which is the value of *x* at the moment that *state* was declared.

Invariants: `[across` *expr*`]` is equivalent to `[require` *expr*`; promise` *expr*`]`.

Continuously Obeyed Invariants: `[always` *expr*`]` is equivalent to `[across` *expr*`]`, with the addition that `[require` *expr*`]` appears at each sequence point throughout the annotated statement.

5.5 Some Details

Some of the more detailed semantics for these annotations are as follows:

- `[pure]` is presently allowed only in class axioms (in a `class` declaration applied to a member or `friend` function). However purity could be meaningful to top-level functions (e.g. `cos()` is pure), an extension which is under consideration.
- A class's legality constraint is automatically included as a precondition of every member function and `friend`. Furthermore objects must also be coherent before a public member or a public `friend`. Thus the actual precondition for a public member or `friend` is the conjunction of its `require`, `across` and `always`, `legal` and `coherent` expressions. Postconditions are analogous, with `promise` substituted for `require` in the above.
- `always` may only appear in assertions (executable code) since it distinguishes a granularity of implementation detail which is too fine for clients to discern (function calls are atomic, so `[always` *expr*`]` and `[across` *expr*`]` are indistinguishable from the client's perspective). In concurrent environments, however, an even stronger (recursive) version of this might be appropriate, which is the subject of ongoing investigation by the authors.
- A quantified variable (either `[`T v`]` or `[exists` T v`]`) is often constrained, accomplished by appending a constraint expression such as `[int i {i` $\geq$ `0 &&` i $<$ `size()}]`. This is equivalent to including the constraint expression as an antecedent in each place the quantified variable is used (usually in `require` and `promise` clauses).

The most common axioms annotate a function ("function axioms"). However axioms can also handle ordinary statements ("statement axioms"). For example, saying that `{push(x); pop();}` is `pure` indicates that pushing anything onto a Stack then popping it off again won't change the Stack:

```
class Stack {
  // . . .
axioms:
  [T x; pure]  { push(x); pop(); }
  // . . .
};
```

Although presented as a minor extension, this is actually a significant step in A++, as it provides A++ behavioral specifications with the full power of Algebraic Specification languages like OBJ [1].

5.2 Assertions

In contrast to Axioms, Assertions appear in the context of executable statements, such as within a function body. In this case, one can imagine that A++ replaces the original assertion:

```
[require x > y; promise z > 0]  z = x - y;
```

with `assert()` macros: [3]

```
assert(x > y);   // Assert the 'requirements'
z = x - y;
assert(z > 0);   // Assert the 'promises'
```

Note that this example is trivial, not demonstrating much of the power of A++. Furthermore A++ does most of its analysis at compile time whereas `assert()` has a non-zero run-time penalty.

Axioms and assertions are similar in many ways since the verifier must prove the postconditions follow from the preconditions in both cases. However an axiom's statement only exists within the verifier at compile time, so it may (and often does) contain references to quantified variables. Other than relatively minor distinctives such as this, an axiom is an assertion whose statement is never compiled.

5.3 Function Axioms

Function axioms (axioms when the statement is a simple function call) can be thought of as an extension of the function's signature: they give information about how the function behaves when it is eventually called, but don't generally indicate how the function is defined.

In function axioms, `require` indicates the required preconditions of the function, as if the function were compiled with an exception test at its head, and `promise` analogously expresses the function's postconditions. Additional annotations such as "*if* p happens to be true at the beginning of `f()`, then q will be true afterwards" can be accomplished by an implication:

```
[ promise !init.p || q ]  f();
```

where `init.`p refers to the value of p at the beginning of `f()`. Note that a simple inline function could make this look better:

```
//Uses: inline int implies(int p, int q) { return !p || q; }
[ promise implies(init.p, q) ]  f();
```

[3] `assert(`x`)` is a preprocessor macro which could be defined to raise an exception should x evaluate to false (zero) at run-time.

array" might be performed within `operator[]`. 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 run-time penalty for the bounds test.

```
Vector operator+ (Vector& a, 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;
}
```

Figure 8: At present, efficiency would dictate making this a `friend`. However it is not always possible for clients to add new friends, and even when it is, an excessive number of friends increases the effective size of the public interface, thus diluting locality.

Although the bounds tests will provably never fail in figure 8, `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 [4]. Neither approach is ideal: the former breaks abstraction and the latter increases the number of functions which can directly manipulate instances.

However, the use of A++ will remove the motivation for such work-arounds. We are carefully exploring implementation strategies that achieve this, while still staying within the reaches of implementability. Such strategies include translation of annotations into "caller-raises" (rather than "callee-raises") exception conventions, the use of multiple function entry points, and reliance on classic back-end optimization strategies for removing dead code.

5 Annotation Syntax

The usual form for formal assertions is:

```
quantifiers { pre } stmt { post }
```

where "quantifiers" refers to a list of $\forall x$ and $\exists y$, "pre" and "post" are boolean expressions, and "stmt" is the statement on which the axioms apply.

Since A++ allows arbitrary statements to be annotated, including an entire function body, it is profitable to avoid the physical separation between preconditions and postconditions. This is the reason behind A++ putting all the annotations associated with *stmt* together:

```
[quantifiers; require pre; promise post] stmt
```

Whether or not the "stmt" is executable (a statement definition or a declaration) distinguishes between two slightly different cases: axioms and assertions.

5.1 Axioms

As has already been shown, axioms are an extension to the signature declaration of a function: they indicate what the function does, but they don't call it. Top-level functions can also be annotated:

```
// A 'top-level axiom' (for a top-level function):
int strlen(const char* s);
axioms [char* s; require s!=0; promise return>=0] strlen(s);
```

are inherited and are thus not repeated. Naturally `full()` and `empty()` are not implemented, having already been specified in `Stack` itself.

```
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;     }
  T    pop()      { return v[--sp];  }
  void clear()    { sp = 0;          }
  int  length()   { return sp;       }
  int  capacity() { return v.size(); }
};
```

Figure 7: A `VStack` IS-A (conforms to the behavior of) `Stack`.

This example represents a significant improvement in clarity over unadorned C++ code. There is no need for explicit error checking (for example, there is no emptiness test before `pop`) since these are part of the behavior of *any* Stack.

As with Eiffel, member functions of derived classes are not allowed to require stronger preconditions or promise weaker postconditions than those in their base classes.

4 A++ and Exceptions

While exception handling features will certainly become incorporated into C++ in the near future, their precise form has not yet been established. It is clearly desirable that annotations be integrated with exceptions, as they are in Eiffel and ANNA. A failed precondition discovered at run-time should trigger an exception. However, the mechanics of A++ are not particularly dependent on the specific nature of C++ exception constructs. In fact, programmers using A++ should require few explicit exception checks:

- Most exception handling revolves around trapping failed preconditions. Since A++ itself will automatically convert preconditions into exception tests (whatever their form might be), programmers need not make much contact with the underlying exception raising mechanisms.
- Relying on A++ to automatically convert legality expressions into exception traps simplifies code, and also ensures that such checks are consistently and completely incorporated.
- When fully developed, A++ will be able to statically remove exception tests that can be proven to never be raised.

The fact that A++ can serve as an "exception optimizer" is a very important consideration for C++ programmers. C++ is often chosen as an implementation language on the basis of its efficiency. Generation of code full of exception checks can easily nullify this advantage. For this reason, we see A++ development as a natural, even vital, part of the evolution of C++. If exception checking is seen by practicing programmers as adding time and space overhead only in those cases where it really matters, there is a greater chance that the resulting better design and implementation techniques will gain wider acceptance.

A case in point is the common C++ programming practice of declaring `friend` functions solely for the purpose of allowing a non-member function to access private data in a way that is known to be safe, thus evading public interface functions that check validity of access. For example, index testing in a "safe

3.2 Abstract Data Types

A class is an implementation of a type, so "class" and "type" are often used synonymously. However an *abstract* data type (ADT) is defined here to be "abstract" in the sense of lacking specific representation. Thus an ADT is a template to create new types, being declared in C++ via an abstract base class. Abstract base classes (ABCs) allow vastly different concrete implementations of an abstract concept to be alternately installed in client code in a "plug compatible" manner.

Typically an ABC contains mostly pure virtual members that define the protocol, but not the implementation, of a set of subclasses. The ABC serves as the root of the resultant inheritance graph. Presently, however, subtypes are guaranteed to be conformant in only a syntactic, and not semantic, sense.

For example, consider the `push` pure virtual member function in an Stack ABC. Subtypes (publically derived subclasses) of Stack are guaranteed to contain a signature compatible member function called "`push`", but no guarantee whatsoever is made on this member function's behavior. Supplementing signature conformance with behavior conformance strengthens the contract associated with abstract data types.

In the Stack ADT of figure 6, all the members are pure virtual except `full()` and `empty()` which are defined in terms of virtual members `length()` and `capacity()`. Note that ADTs (annotated ABCs) typically have no legality or coherence constraint since these annotate a particular concrete representation.

```
class Stack {
public:
  virtual void push(T)    = 0;
  virtual T    pop()      = 0;
  virtual int  length()   = 0;
  virtual void clear()    = 0;
  virtual int  capacity() = 0;
  virtual     ~Stack() { }
              Stack() { }
          int  full()  { return length() == capacity(); }
          int  empty() { return length() == 0; }
axioms:
  [ require !full(); promise !empty(); T x ] push(x);
  [ require !empty(); promise !full()      ] pop();
  [ promise return >= 0                    ] length();
  [ promise empty()                        ] clear();
  [ promise return > 0                     ] capacity();
  [ promise empty()                        ] Stack();
};
```

Figure 6: Abstract data types (annotated abstract base classes) can be rich in concept even though they are starved of implementation details.

3.3 Inheritance

C++ supports both public and private subclasses. Private subclasses are used either for partial code reuse or to express a HAS-A relationship. With private derivation, no mechanism is provided for automatic inheritance either by C++ (inheritance of the public interface) or by A++ (inheritance of behavioral constraints).

Public derivation express the IS-A or subtype relation: if `VStack` is publically derived from `Stack`, a `VStack` IS-A `Stack`. The implication, lacking currently in C++, is that a `VStack` should behave like a `Stack`. This is accomplished in A++ by providing `VStack` with all the behavioral specifications of `Stack`. Since the two types may use inherited `protected` members differently, the `legal` and `coherent` class-wide specifications are not *automtically* inherited.

The implementation of `VStack` in figure 7 HAS-A `Vector` and IS-A `Stack`. The behavioral specifications

```
class Stack {
  int sp;   // stack pointer
  // . . .
public:
  int  empty() { return sp == 0; }
  int  full()  { return sp == capacity(); }
  T    pop();
  void push(T x);
axioms:
  [ require !empty(); promise !full()      ] pop();
  [ require !full(); promise !empty(); T x ] push(x);
  // other Stack axioms . . .
};
```

Figure 4: A highly elided example showing some "stack axioms" for a Stack of `T`. These are valid for *any* "Stack of `T`" implementation, whereas the definitions of the various member functions are implementation dependent.

all of these concepts, with object-based and class-based languages supporting only a subset. The following sections show the role of A++ with respect to each.

3.1 Encapsulation

Encapsulation is the strategy of causing the underlying implementation of a class to be hidden from its users, while also presenting a controlled interface of its functionality. Objects are presented as complete entities rather than just data or just code.

Figure 5 adds only slightly to the earlier examples: the form `init.sz` is the initial value of parameter `sz`; `resize()` promises `size()` will be what formal parameter `sz` was initially. In the axiom for `size()`, `return` is an expression referring to the member's return value. Instances are legal if their capacity, `cap`, is non-negative, and they are coherent if the number of elements pointed to by `data` is equal to `cap`.

```
class Vector {
  int cap;     // Capacity of the Vector
  T*  data;    // Pointer to the actual data elements
legal:    cap >= 0;
coherent: data.nelems == cap;
public:
  int  size() { return cap; }
  T&   operator[](int x) { return data[x]; }
  void resize(int sz);
       Vector(int sz=10) : data(new T[sz]), cap(sz) { }
      ~Vector() { delete [cap]data; }
axioms:
  [ promise return >= 0                            ] size();
  [ int x;  require x >= 0 && x < size()           ] (*this)[x];
  [ int sz; require sz>=0; promise size()==init.sz ] resize(sz);
  [ int sz; require sz>=0; promise size()==init.sz ] Vector(sz);
};
```

Figure 5: Example of Data Encapsulation

2.2 Coherent Objects

The coherent (self consistent) states of an object form a subset of the legal states. Objects containing redundant information usually become temporarily inconsistent (violate coherence) while changing state. For example, a linked list might become temporarily inconsistent between adding a node and incrementing the node count. All public member functions have the right to expect a coherent object, and have the responsibility to restore coherence before finishing.

As with legality, the programmer simply describes coherence at the class level, as in figure 3. A++ statically verifies (again, to the extent possible) that all constructors construct coherent objects, and that objects remain coherent "across" (ie: at the beginning and end, but not necessarily throughout) each public member function.

```
class Vector {
  int cap;     // Capacity of this Vector
  T*  data;    // Actual data elements
coherent:
  cap == data.nelems;
      // data.nelems denotes number of 'T's pointed to by 'data'
public:
  //...
};
```

Figure 3: A fragment of a Vector of `T`. Coherence may only be violated during the operation of a "privileged" function (a member or a `friend`).

As with the legality constraint, A++ may force certain constructor calls to dynamically check their arguments, but the ideal overhead is small since A++ will attempt to verify continued conformance. Again, constructors and destructors are treated specially: instances aren't required to be coherent until after a constructor, and are only required to be coherent before the destructor. And again, coherence is a contract with the class maintainer, giving rich indication of internal design specification.

2.3 Behavioral Specifications

Legality and coherence are invisible to clients of the class. If the class has been implemented correctly, a client can never see an object that is illegal or even incoherent. Indeed an object will *never* be illegal, and will only be incoherent within a privileged function.

However there are numerous conditions which are visible to the client. For example, a stack might be "full," a file might be "readable," etc. All these are strict subsets of coherence (and hence of legality). The behavior of operations (member functions) is annotated as transitions between these conceptual states. For example, part of the behavior of any stack is that `pop` requires a stack that isn't "empty" and promises to leave the stack in a state that isn't "full".

These behavioral "class axioms" appear in an `axioms` section, as in the "Stack axioms" shown in figure 4. `require` indicates a condition which must be obeyed before the operation begins, with `promise` indicating a condition which will hold after the operation finishes. `T x` in `push`'s axioms is a universally quantified variable ("for all `T` x ...").

Unlike legality and coherence, behavioral axioms use only implementation-independent information (public member functions), thus corresponding to concepts rather than raw data values. Furthermore this behavior is inherited by subtypes (publically derived subclasses), thus capturing the essence of the IS-A hierarchy.

3 Encapsulation, ADTs and Inheritance

Object oriented methodology is often presented as consisting of three distinct concepts: data encapsulation, abstract data types, and type inheritance [5]. Object oriented languages are said to be those which support

Figure 1: The "Universe" of possible values, with subsets called Legality, Coherence and various conceptual conditions which help define a type's implementation-independent Behavior.

read from a data file. However A++ minimizes further dynamic run-time checking in such cases, since it will attempt to show that instances remain legal once they are constructed.

Since objects are incomplete during constructors and destructors, instances aren't required to obey the legality constraint during the head of constructors or during the body of destructors.

Even apart from formal verification, legality is a valuable maintenance tool, since it provides a concise documentation of the class designer's intentions. C++ by itself does not provide a fine enough granularity of type qualifiers to accomplish these goals, leaving the class designer with comments in some less rigorous language. Legality is therefore an arbitrarily precise "contract" between the class designer and the class maintainer.

```
class Calendar {
  int month, date;
  int days_in_month(int mo) { /*...*/ }
legal:
  month >= 1 && month <= 12;
  date >= 1 && date <= days_in_month(month);
public:
  Calendar() : month(1), date(31) { /*...*/ }
  //...
};
```

Figure 2: The "legality" constraint is never violated throughout an object's lifetime.

- *Semantic Information:* Most programming languages are very limited in their ability to express semantic information. Indeed one reason descriptive comments, meaningful identifiers and up-to-date documentation are so valuable is that the language proper has little to express the *intent* of the programmer. A++ rectifies this by supporting behavioral specifications of objects.

- *Integration:* In the spirit of ANNA, we view annotation systems as extensions of native type systems. Whenever possible, A++ syntax was chosen to be consistent with that of the underlying C++ type system.

- *More Natural Programming Style:* Because A++ supports the expression of high-level constraint information, C++ programs using A++ may require fewer explicit consistency and error checks, enhancing both readability and naturalness of programming.

- *Inherently Object Oriented:* Unlike many systems (e.g., ANNA [2]) which annotate what "code does to data," A++ annotates objects as a whole, thus avoiding the live-code dead-data notion common in program verification strategies, but antithetical to the object-oriented programming paradigm.

- *Support for C++isms:* Different object oriented languages incorporate the basic ideas of inheritance, encapsulation, etc., in different ways. A++ accommodates, and in some respects extends, C++ specific constructs. Some mechanisms are essentially identical to those in Eiffel (e.g., the ways in which annotations are inherited by publically derived subclasses). Others, supporting C++ private inheritance, protection mechanisms, etc., necessarily differ, leading to a system with a noticeably different emphasis and character than Eiffel.

2 Expressive Power of A++

When programmers declare the data members of a C++ class, they implicitly allow instances of that class to enter states corresponding to all possible bitwise combinations of these data. Of course, it is almost always the case that some of these states are illogical, unreachable, or undesirable. Subsets of this "universe" correspond to restricted ranges of values, or alternatively as constraints on instances of the type; see figure 1. From the perspective of annotating the type as a whole, useful choices for these subsets/constraints must be identified.

Briefly, constraints which instances must always obey identify "Legal" objects. "Coherent" objects correspond to a subset of the legal states and are analogous to self consistency: all public member functions have the right to expect a coherent object, and they have the responsibility to restore coherence before terminating. Of the coherent values, numerous conceptual conditions exist which help define a type's implementation independent "Behavior." For example, the fact that `pop` requires a "non-empty" stack and promises a "non-full" stack is part of the behavioral description of a stack.

These concepts are detailed in the sections that follow.

2.1 Legal Objects

Classes are typically designed to use only a portion of their possible set of values, giving rise to the Legality constraint. This continually obeyed constraint can be thought of as a subset of the "universe" (see figure 1) which instances never leave.

The expressiveness of A++'s legality constraint is rich. It extends the semantic power of C++ to include subranges (by saying `int month` continually abides by the constraint `month` $\geq$ `1 && month` $\leq$ `12`), as well as many more esoteric forms, such as `int odd` always obeys `(odd & 1) == 1`. Even dynamic expressions can be used to delimit the set of legal values, such as the stack constraint `stackPtr` $\leq$ `capacity`.

The arbitrary predicates which describe the legality subset thus give an extremely fine granularity to the base type system. Syntactically this is accomplished by adding a `legal` section to the class declaration, as in figure 2. A++ verifies that all constructors construct legal objects, then it verifies, to the extent possible, that objects remain legal throughout the operation of each member function.

Whenever it is not possible to statically determine that a constructor call will result in a legal object, A++ generates code to dynamically test initializers. This is necessary, for example, when an initializer is

The Behavior of C++Classes

Marshall P. Cline[1] and Doug Lea[2]

Abstract

A++ ("annotated C++") is introduced as an aid in writing clearer, safer and faster obje programs in C++. A++ supports the specification of semantic information associat in a manner which is consistent with the object oriented paradigm. It enhance formal verification, (2) code clarity, by replacing explicit dynamic consisten level annotations, and (3) code performance, by using the annotations as an aid

A++ has two significant side benefits: (1) it encourages a more natural object or style than otherwise possible, again promoting safer code without execution pe the power of the C++ base type system by supporting arbitrarily precise type d extremely fine grained type qualifiers (subranges, arbitrary sets, dynamic cons

In light of the complexity of formal verification, the immediate practical co use as a specification aid, using a syntax intended to be more familiar and natura than are most specification systems. It is commonly held that the first task in specify the public interface. A++ extends this "contract" by empowering the cl not only the signatures of the public member functions, but also their semantic

1 Introduction

1.1 Basic Motivation

A++ ("annotated C++") is both an annotation formalism and a proposed C++ programming tool supporting object oriented annotations for C++. As a formalism, it provides a means for designers to express their intentions in a concise, high level fashion that is consistent with the object oriented paradigm. Code clarity is enhanced by, among other things, replacing explicit exception tests with higher level "behavioral constraints."

As a CASE tool, A++ performs formal verification on the annotations. By showing the consistency of what was *meant* (the annotations) with what was *said* (the code), code safety is increased. Verified libraries of objects will promote code reuse, one of the elusive trademarks of the object oriented paradigm.

The tool is intended to be an integral part of the compilation process. Positioned as a "front end" to the normal C++ compiler, it is able to use the annotations to make a number of critical improvements to the code. For example, many run-time consistency and exception tests are redundant. Formal verification shows this redundancy, and A++ (as a front end to the compiler) removes the extra tests. Thus exception testing overhead is reduced without sacrificing safety. In this sense, A++ is an "exception optimizer."

A++ enjoys the unusual status of improving both code safety and effiiency. The tension between these forces is well known, with maintenance cost invariably pushing design toward safety and clarity at the sacrifice of speed. A++ causes these normally opposing goals to be symbiotic, allowing a simultaneous improvement in each.

A++ contains features inspired by ANNA [2], Eiffel [3], and other specification systems, but with constructs specifically designed around C++.

1.2 Design goals

- *Small:* A++ attempts to remain in the spirit of C++. In addition to being simple to use and terse, the annotation language was kept fairly small. One of the advantages of this decision is that A++ appears to be implementable using existing compiler and program verification technology.

[1] M. Cline is with the Electrical and Computer Engineering Department, Clarkson University, Potsdam, NY. Email: cline@sun.soe.clarkson.edu.

[2] D. Lea is with the Computer Science Department at SUNY Oswego, the NY CASE Center at Syracuse University, and the GNU project. Email: dl@oswego.oswego.edu.