JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5 No. 6, July - August 2006

# ABS++ : Assertion Based Subtyping in C++

**Herbert Toth**, SIEMENS AG Austria, PSE KB B

**Abstract**

For software engineering to meet today's challenges, well designed reuse and composition mechanisms must be established in both theory and practice. Starting roughly ten years ago, theoretical principles and solution possibilities for assertion annotations in daily practice are being discussed in an ever growing number of papers and web pages. We present a further proposal for C++ that is based on the macro technique, conforms to the LSP, enables quantification, and very much resembles the way Design by Contract™ is implemented in Eiffel.

## 1 INTRODUCTION

In this paper I present a C++ macro package together with a technique that - while keeping the additional burden for the software engineer as small as possible - offer him much of the features of Design by Contract™ (henceforth: DbC) as provided by Eiffel (see e.g. [Meyer97]), but avoid its deficiencies (see [Toth05]).

Though there are new ideas included in the macro packages, none of them would ever have been conceived without the inspiration and without the foundations laid by other people. What you find in this paper is what my modest C++ and time resources allowed me to provide using existing concepts, code and visions. However, with some exceptions, I will not undergo the labor to indicate sources explicitly: All of what is listed in the reference section has contributed in one way or the other (and perhaps some other sources too that I do not remember any more).

The structure of the paper is as follows: Section 2 contains an overview of the main concepts and rationales behind assertion based software development and DbC. Section 3 describes the logical and structural properties of ABS++, and in section 4 I present some technical details of the ABS++ and the demonstration example. The final section provides a summary of the main ideas and some conclusions on the technique presented in this paper. At the same location as the paper itself you can also find the two macro packages I have used as well as the example sources. A number of hints to further information is provided by references to papers, books, and web pages.

Cite this article as follows: Herbert Toth: "ABS++: Assertion Based Subtyping in C++", in *Journal of Object Technology*, vol. 5, no. 6, July - August 2006, pp. 83-105,
http://www.jot.fm/issues/issue_2006_07/article3

## 2 ASSERTIONS AND DESIGN BY CONTRACT ™

What has been common in the area of hardware design for approximately twenty-five years under the name design for testability, viz. to enhance the product with means to increase the observability of its behavior, only during the last few years seems to gain some attention also within software industry. This is all the more strange, as software has gained a still rapidly increasing part of responsibility for the well functioning of so many (really or only seemingly) important things of every day life. Due to this pervasive part of software in modern society, its quality is important not only in the context of safety critical systems (e.g. nuclear power plants, cardiac pacemakers), but also for our daily life as a whole, from business to leisure (online banking, mobile phones, internet, cars etc.).

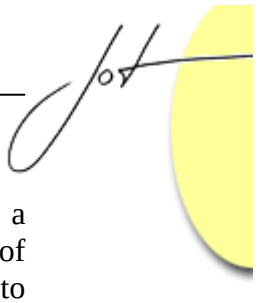
The technical means for achieving design for testability in software engineering is assertion based programming, the logical strategy behind is best known as DbC, a prominent and systematic method introduced by Bertrand Meyer, the creator of the Eiffel language.

What certainly should be regarded as indication of an increasing interest in this kind of software development starting in the mid-nineties is the growing number of publications in this area like e.g.:

- a considerable number of papers dealing with the concepts and the practical use of assertions in general, e.g. [Binder00, chapter 17], [Mannion98], [Meyer92], [Meyer97, chapter11], [Mitchell02], or [Payne97];
- the Assertion Definition Language (ADL) developed at Sun Labs (see [ADL]);
- newer Methods like Syntropy [Cook94], the Eiffel-related BON [Walden95] and Catalysis [D'Souza99], provide means for the inclusion of assertions into the graphical model;
- the Object Constraint Language (OCL) becoming a supplement to the original version of the Unified Modeling Language (UML). Making OCL a part of UML is a consequence of the fact that graphical models alone are not enough for a precise and unambiguous specification. There often is a need to describe additional constraints about the objects in the model.
- emerging support for assertions for Java (from which even the ANSI C assert mechanism has been removed at first, and reintroduced later on); see [Payne98], the iContract tool [Kramer98], Jass [Jass], or AssertMate from Reliable Software Technologies [AssertM];
- emerging support for assertions for C/C++; see the overview in [Maley00], [Binder99], [Porat95], or [Welch98].

### Assertion based programming

Although I assume many of the readers to be familiar with the topic under discussion, there will hopefullly also be some newcomers, and it is for them that I provide a short summary about the main concepts and ideas behind assertion based programming.



Software assertions are Boolean expressions that define the correct state of a program at a particular location in the code. It is useful to think of them as a kind of watchdog having abilities which can be very helpful for you - if you are willing to provide them and only when you allow them to be active. They can check method calls for proper invocation, method code for correct computation, states of objects for consistency, and also individual statements for errors. Therefore, assertions may act in the following different roles, where the first three groups form what I call *contracting assertions*:

- *Preconditions* express the requirements that clients must satisfy whenever they call a routine, and are therefore evaluated at the entry point of a method. Preconditions are obligations for the client (caller), and benefits for the server (callee).
- *Postconditions* inform about what the supplier (i.e. the routine) guarantees on return if the precondition was satisfied on entry. They have to be evaluated at all exit points of the method - if you allow more than one in your coding standard. Postconditions are obligations for the server, and benefits for the client.
- In object-oriented projects: *Class Invariants* define the consistency conditions for the state space of a class and must be satisfied by every instance of the class whenever this instance is externally accessible, i.e. after creation, and after any call to an exported routine. Class invariants have to be evaluated at the entry and all exit points of all externally visible routines of a class.
- *Data assertions* define the conditions that must hold at just their location in the code, whence they are evaluated only at just this location. You can take them for your own individually tailored and eventual only temporary tests. A special case of data assertions are loop invariants and loop variants.

Using formal languages and methods [BowenFM] to overcome the problems of software development outside of some safety-critical software systems is far from getting the rule: This kind of formally deduced correctness is often argued to be too difficult for the average system modeler and thus has the great disadvantage that it can be applied by people with a strong mathematical background only. Thus, a steep learning curve and a big initial effort is to be expected for becoming familiar with it.

Interesting alternatives in the area of formal languages and methods to overcome this gap between what is possible in theory and what is feasible in practice are e.g. the B-Language [Abrial96], [Escher] or [Liu04] which allow specifications in a rather programming-oriented style. Another possibility that seems to be more realistic nowadays is assertion based programming according to DbC principles.

As you might know, all these ideas are not really new: their roots date back at least to the late 60's and the early 70's. There is indeed a long and tedious way in formal methods from Floyd's *assigning meanings to programs* in [Floyd67], Hoare's axiomatic basis [Hoare69] to Abrial's *Assigning programs to meaning* subtitle of *The B-Book* [Abrial96]. Today's average software engineer usually does not make use, and often does not even know, of formal methods and corresponding tools. However, I think it is high time to take advantage of some of their findings, and assertion based software engineering may eventually prove to be a bridge between current practice and theory.

## What is Design by Contract™?

"Design by Contract" denotes a software development style which (1) emphasises the importance of formal specifications, and (2) interleaves them with actual code. DbC is a systematic method of assertion usage and interpretation introduced by Bertrand Meyer as a standard feature of the Eiffel language. Without it, no trial would have ever been made to provide a similar mechanism in other languages and, by no means, would we have discussion papers like this and the ones mentioned in the references.

A software contract is the specification of the behavior of a class and its associated methods. The contract outlines the responsibilities of both the caller and the method being called. Failure to meet any of the responsibilities stated in the contract results in a breach of the contract, and indicates the existence of a bug somewhere in the design or implementation of the software or - one must not forget - in the assertions themselves. Software contracts can be completely specified by preconditions, postconditions, and class invariants.

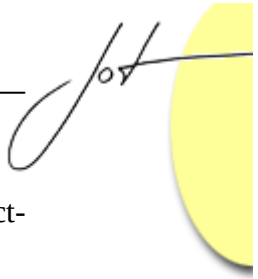
Software construction is based on contracts between clients and suppliers. Each party expects some benefits from the contract, and accepts some obligations in return. As in human affairs, the contract document spells out these mutual benefits and obligations and protects both he client, by specifying how much should be done, and the supplier, by stating that the supplier is not liable for failing to carry out tasks outside of the specified scope.

DbC is, in a way, the opposite of defensive programming, a method which recommends to protect every software module by as many checks as possible. This may result in redundancy and makes it also difficult to precisely assign responsibilities among modules. Software contracts are also a necessary prerequisite for introducing a notion of correctness: If you do not state what your program should do, you are lacking the norm to which to compare what your program does in reality.

## Contracts and inheritance

As the software community has learned during the last two decades, the object-oriented approach is very powerful for developing large software systems. Much of this power is due to the key concept of inheritance. However, the statically checks enforced by e.g. C++ or Java compilers upon derived classes test for such syntactic and typing restrictions only that guarantee the lack of runtime type errors. This is the contracting and specification level that has been used for too many years in the past by most software developers. Obviously, however, this is not enough to prevent surpising and often disastrous behavior of programs.

In other words, the checks done by compilers are only part of what is needed to reason about the behavior (i.e. the semantics) of software, especially for object-oriented systems when new subtypes are added. Behavioral subtyping is a technique for preventing unexpected behavior in a modular way: it ensures that objects of new subtypes (instances of subclasses) "act like" objects of their supertypes, when used as if they were



supertype objects. This is what the Liskov Substitution Principle (LSP) for object-oriented design states [Liskov88]:

1. In class hierarchies, it should be possible to treat a specialized object as if it were a base class object.
2. Or: Object-oriented functions that use pointers or references to a base class must be able to use objects of a derived class without knowing it.

The basic idea here is as simple as it is important: Derived classes should not perform any actions that will invalidate the assumptions made by (the client of) a parent class. Put differently, any object of a subtype must be substitutable for an object of a supertype in the hierarchy without any effect on the program's observable behavior. If the Liskov Substitution Principle is followed, code using a base class pointer will never break after another class has been added to the inheritance tree.

This gives the basic rule governing the relationship between inheritance and assertions: A descendant class must obey all the ancestors' constraints, i.e. routine pre- and postconditions, and the class invariants still apply. Therefore, contracting assertions have to be checked along the class hierarchy in a suitable way, i.e. they are used not only on the places where they appear in code (contrary to data assertions which have local impact only). The assertions of a routine specify a range of acceptable behaviors for this routine and its eventual redefinitions which may specialize this range, but not violate it.

## Data assertions

Data assertions do not contribute to software contracts as outlined above, but rather are means to help doing some internal checks according to your specific, and usually only temporary, needs. They define conditions that must hold in a particular location in the code, and are, therefore, evaluated at this location only. As an exmaple we take a short look on loop variants and invariants.

The Eiffel mechanism is described in [Rist95], p.298, as follows: "If loop assertions are monitored at run-time, then both the variant and the invariant will be evaluated immediately after loop initialization. The invariant must evaluate to true, and the variant must be greater than or equal to zero. If either of these conditions is not met, then an exception will be raised and the system will terminate. As loop execution continues, both the variant and the invariant will be evaluated after each iteration of the body. As before, the invariant must be true and the variant must be non-negative or an exception will be generated. The system will also check that the variant is decreased by each execution of the loop body."

## How much assertion monitoring?

This section summarizes the considerations of an equally headed one in [Meyer97], pp. 394-398, often using direct quotations from there. What level of assertion tracing to enable at runtime "is a tradeoff between the following considerations: How much you trust the correctness of your software; how crucial it is to get the utmost efficiency; how

serious the consequences of an undetected run-time error can be." The two extreme cases are

- to enable assertion monitoring at the highest level for the classes of the system during testing it prior to release;
- to remove all monitoring for fully trusted systems in a runtime-critical application area where every microsecond counts.

Note, however, the following remark by C.A.R. Hoare [Hoare73]: "It is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in the production runs, when an erroneous result could be extensive or disastrous. What would we think of a sailing enthusiast who wears his life-jacket when training on dry land but takes it off as soon as he goes to sea?"

"An interesting possibility is the option that only checks preconditions ... it has the advantage of avoiding catastrophes that would result from undetected calls to routines outside of their requirements, while costing significantly less in run-time overhead than options that also check postconditions and invariants. …

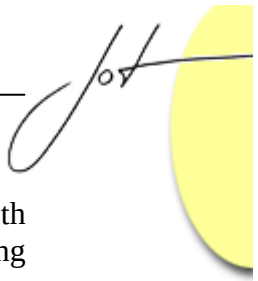
This option is particularly interesting for libraries. Remember the basic rule on assertion violation: a violated precondition indicates an error in the client; a violated postcondition or invariant indicates an error in the supplier. ... But even for a perfect library it is useful to check preconditions: the goal is to find errors in client software."

One cannot give a general answer to the question of how much assertion monitoring to do at runtime without having some rough ideas about the performance overhead caused by it. As a rule of thumb one can take the following: Preconditions are often relatively simple conditions checking for the client's obligations like lower<= idx && idx <= upper etc., whereas many postconditions and invariants may include a lot of relevant consistency conditions expressing more advanced semantic properties; moreover, invariants are evaluated twice. Thus, the above advice to adhere to Eiffel's default level of checking the preconditions seems to be a practicable one for many situations.

## 3 THE ABS++ SOLUTION

An obvious and important fact you always have to keep in mind is the following: Contrary to Eiffel compilers, C++ compilers do not support software development according to the DbC paradigm. As a consequence, you as the developer have to support the compiler, i.e. you alone are responsible not only for the correct implementation, but also for the correct placement of pre- and postconditions (with the exception of class invariants), of the check invocations, and of the loop variants and invariants. Again, following the schematic way as proposed in the ABS++ examples in this paper, this is an easy and systematic task.

In case you now have the feeling that, by using ABS++, you are forced to write a lot of additional code, remember the following: Whether you use Eiffel or some annotations in form of e.g. Java pseudocomments together with some preprocessing mechanism, in



any case you will have to formulate and write down your assertions – in a both syntactical and semantical correct way. What Eiffel or an intelligent preprocessing mechanism will save you, is the work you have with getting assertions handled right over class hierarchies. This can, however, be done in a simple and schematic way using the macros provided in QSubtype.h (see section 4).

## The theoretical background of ABS++

In this section we will give a short overview on the theoretical background of ABS++ in order to give the reader the possibility to compare our macro package with related work. Our presentation is mainly based on the results in [Chen00] and [Toth05].

Behavioral subtyping in ABS++ is done by the so-called *weak plugin* specification match: $(C_{pre} \rightarrow SC_{pre}) \wedge (SC_{pre} \wedge SC_{post} \rightarrow C_{post})$. This is not a most general reuse ensuring[1] match (see Theorem 7 in [Chen00]), but it is not far from it as has been shown in [Toth05]. It can thus be considered as a well-suited candidate for practical purposes.

Compared to the percolation pattern – which is the one used in Eiffel and all the other DbC implementations for C++ or Java I know (Jass excluded!) – there are two definitive advantages of the ABS++ mechanisms: (1) hierarchy errors are detected for both pre- and post-conditions, as well as class invariants; (2) never will a routine with its own precondition evaluating to **false** be executed. (Suppose that for a method *m* $SC_{pre}$ fails and $C_{pre}$ holds; according tho the percolation pattern the effective precondition (i.e. what is really checked at runtime) for *m* in SC is $SC_{pre} \vee C_{pre} \equiv$ **false** $\vee$ **true** $\equiv$ **true**. Thus the execution of method *m* of class SC will start notwithstanding the fact that its own precondition $SC_{pre}$ evaluates to **false**, and that there is an undetected hierarchy error from class C to class SC with respect to *m* violating the Liskov Substition principle.)

It is important to know at what points in the program execution different assertions have to be checked. The following table shows how pre-, postconditions and class invariants are used in the context of a class.

[1] A specification match *M* is *reuse ensuring* if $M(SC,C) \wedge \{SC_{pre}\}m_{SC}\{SC_{post}\} \rightarrow \{C_{pre}\}m_{SC}\{C_{post}\}$, where $\{P\}m\{Q\}$ represents a Hoare triple, informally stating that method *m* if started in a state such that *P* holds does stop in a state where *Q* holds.

| | | Invariant | Pre-condition | Post-condition |
|---|---|---|---|---|
| **Method Entry** | *Instance Method* | yes | yes [1)] | - |
| | *Constructor* | no | - | - |
| | *Private Method* | no | yes [4)] | - |
| | *Static Method* | no | yes | - |
| **Method Exit** | *Instance Method* | yes [2)] | - | yes |
| | *Constructor* | yes [3)] | - | - |
| | *Private Method* | no | - | yes [4)] |
| | *Static Method* | no | - | yes |

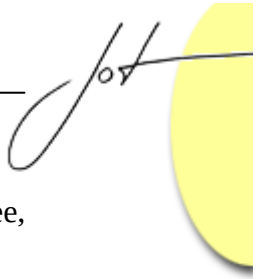
1. The preconditions of a method are checked only in case that the class invariant has been found valid.
2. The class invariant is checked at the end of a method only if its postcondition has been found valid.
3. You have to insert the `classInvariant(_FL_)` calls manually into constructors. This means that you have to provide at least one in order to avoid C++'s default constructor.
4. You may think of doing these checks (which are not 'official' contracts for external clients of the class) using the `check` mechanism, in order to clearly differentiate between contracting and other constraints.

## Down-calling or How to manage visibility

In this section we shortly present the technique of down-calling as proposed in [Payne97]. Down-calling can be used to ensure the semantic consistency of method inter-faces for polymorphic methods.

Polymorphic methods with public visibility can become problematic, because clients may provide different implementations of both assertion and application code in derived classes. Down-calling solves this problem by structuring the polymorphic call so that a method's pre- and postconditions are always evaluated consistently across all derivations of a class. The technique uses two types of methods:

- *Interface methods* are public but not polymorphic (whence a derived class will inherit them directly in the base class form); they manage the preconditions and postconditions for the class methods, as well as the class invariant. These methods are thus responsible for ensuring the logical semantics of a method across the class hierarchy.
- *Implementation methods* are not public but polymorphic (whence a derived class can adapt their behavior to its specific needs) and provide the implementation for a



particular interface method. Since their visibility is restricted to the inheritance tree, clients cannot introduce improper redefinitions.

This technique finally results in the Template Method pattern (see [Gamma95]) and looks like the following example:

```
void C::foo(int aParam, _DefFL_)
{
   fooPre(aParam, _CallFL_);
   fooImp(aParam);
   fooPost(_CallFL_);
}
```

The interface method of our example clearly is `C::foo`, structured in three parts: `fooPre` checks foo's precondition (and class invariant), `fooImp` contains the application code for foo, and `fooPost` checks the methods postcondition (and class invariant).

## The typical ABS++ pattern

By putting the contracts into the .h files, we can emulate something which is similar to Eiffel's short form, but we will never get such a beautiful and elegant layout. You can do one more step by using an IclassName.h file for the client interface containing all public methods in addition to the usual className.h (which then has to include the IclassName.h file) and className.cpp.

Combining this with the considerations of the foregoing section we get a typical (empty) template including the class invariant and the constraints for method foo in e.g. C.h. Do carefully note which methods are virtual and which are not, as well as which methods are public and which are not.

```
#ifndef C_hpp
#define C_hpp 1

#include "QSubtype.h"

class C
{
public:
   // ...
   void foo(int aParam, _DefFL_);

protected:

   int* natSequ;
   const int  seqLen;

   virtual void fooImp(int aParam);

   // § § § § § § § § § § § § § § § § § § § § § § § § § § § § § § § § § § § § § §
   // ------------------------------------------------------------------------
   virtual bool classInvariant(_DefFL_) const
   {
       checkInvariants(
          invariant(true, "C");    // this is an empty template
       , file, line, C);
   }
```

```
    // ------------------------------------------------------------------
    virtual bool fooPre(int aParam, _DefFL_) const
    {
       checkPreconditons(
          require(true, "preCond_1 of C::foo");
       , file, line, C);
    }

    // ------------------------------------------------------------------
    virtual bool fooPost(_DefFL_) const
    {
       checkPostconditons(
          ensure(true, "postCond_1 of C::foo");
       , file, line, C);
    }

    // obligatory, because internally used in ABS++, are:
    //
    static bool _inABSppHierarchyCheck_;

public:

    static bool _ABSppChecksEnabled_;
// § § § § § § § § § § § § § § § § § § § § § § § § § § § § § § § § § § §
}; // end of C.hpp

#endif   // C_hpp
```

## How to handle empty assertions

It may, of course, occur that you do not have specific constraints of a certain kind for either a method or a class. The question, then, arises what default values to assign to such “empty” pre- or postconditions, or class invariants. Empty assertions are trivially set to **true**, since no constraint means no restriction whatsoever. Hence, for pre- and post-conditions as well as for class invariants everything is okay – always, regardless of the concrete program state.

**Note**: Do always provide the empty pre- and postconditions, and the empty class invariants in order to enable proper subtype and class invariant checks ! (The information on the class names is passed in the corresponding macros.)

## Contract Support Evaluation of ABS++

The criteria used for evaluation of contract support are taken more or less literally from [Plösch02] and [Plösch04], and are split into four groups as shown in the tables below. (Note: For simplicity reasons the term 'system' is used for programming languages, programming language extensions and programming systems.)

| Basic assertion support (BAS) | |
|---|---|
| ***BAS-1 (Basic assertions)*** | |
| Does the system support basic assertion annotations in the implementation of a method? | yes |
| ***BAS-2 (Preconditions and Postconditions)*** | |
| Does the system support preconditions? | yes |
| Does the system support postconditions? | yes |
| May assertion expressions access properties of a class? | yes |
| Are properties of a class guaranteed to remain unchanged during assertion checking? | yes |
| May assertion expressions also contain method calls? | yes |
| Is it guaranteed, that a method call does not produce any side effects (especially changes of the state of the object)? | yes [1)] |
| ***BAS-3 (Invariants)*** | |
| Is it possible to formulate invariants? | yes |
| Are the restrictions in formulating invariants the same as for the formulation of preconditions or postconditions? | yes |

1. Use `const` methods for pre- and postconditions and class invariants, but be cautious using the `check` macro.

| Advanced assertion support (AAS) | |
|---|---|
| ***AAS-1 (Enhanced assertion expressions)*** | |
| May assertions contain Boolean implications? | yes |
| May postconditions access the original values of parameters, i.e., the values at method entry? | yes [1)] |
| May an arbtrary expression be evaluated at method entry? | yes |
| ***AAS-2 (Operations on collections)*** | |
| Does the system support assertion expressions on collections? | yes [2)] |
| Is it guaranteed that collections remain immutable in assertion expressions? | yes [3)] |
| May universal quantifications be expressed in the expression language? | yes [2)] |
| May existential quantifications be expressed in the expression language? | yes [2)] |
| ***AAS-3 (Additional expressions)*** | |
| Does the assertion expression language have additional features? | no |
| Are these additional features guaranteed to be side effect free? | no |

1. You have to provide the values at entry by using a macro.

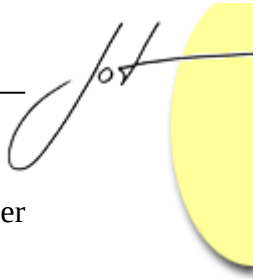
2. The mechanism works for iterators too; see the vector example at the bgin of ABS.cpp.
3. See note to table for Basic Assertion Support on page 11.

| Support for Behavioral subtyping (SBS) | |
|---|---|
| ***SBS-1 (Interfaces)*** | |
| Is it possible to specify contracts for interfaces? | (yes) |
| May contracts be added for classes implementing assertion-enriched interfaces? | - |
| ***SBS-2 (Correctness I)*** | |
| Does the system impose any restrictions on subcontracts? | yes |
| Does the system ensure that preconditions may only be weakened? | yes |
| Does the system ensure that postconditions may only be strengthened? | yes |
| ***SBS-2 (Correctness II)*** | |
| Does the system impose stronger requirements on subcontracts as specified in SBS-2? | yes [1)] |
| Does the system ensure, that the correctness rules for behavioral subtyping [Liskov94] are not violated? | yes |

1. Does subtype checks for class invariants too (analogous to postconditions).

| Runtime monitoring of assertions (RMA) | |
|---|---|
| ***RMA-1 (Contract violations)*** | |
| Is an exception handling mechanism available in case of violations of assertions? | yes[2)] |
| Are there additional features available for dealing with assertion violations (e.g., log files)? | yes |
| ***RMA-2 (Configurability)*** | |
| Is it possible to enable and disable precondition checking, postcondition checking and invariant checking selectively? | yes |
| Is it possible to enable and disable assertion checking on a package, class or even method level? | no[3)] |
| ***RMA-3 (Efficiency)*** | |
| Are there any additional memory requirements even when assertion checking is disabled? | no |
| Is there any additional processor usage even when assertion checking is disabled? | no |

2. Depends on how you define the `assertAndReact` macro in QAssert.h. Alternatively, you may think of integrating exeception handling in another way, e.g. as shown in section 5.



3. Possible on class level; but does not make very much sense in the light of proper subtype checking.

## 4 THE ABS++ MACRO PACKAGES AND SOME EXAMPLES

### QAssert.h

This part of ABS++ is due to [Mueller94], with some slight modifications and renamings. It is here where quantifiers are made possible. For technical details we have to refer the reader to [Mueller94], but in order to make the current paper a rather self contained piece of information and to prevent any difficulties concerning the availability of the October 1994 CUJ issue, we give some hints on how to use this macro package.

- assertAndReact has the same effect as the normal `assert`, but additionally it outputs all values given in the second parameter to a special output stream, and thus it avoids the secrecy of `assert` about the values that caused the assertion to fail. This is especially useful in connection with the `forall` and `exists` conditions, because knowing which cycle of the internal for loop failed is important for tracing an error. If you want to output the loop variable in case an assertion fails, you must declare it at the global scope as shown in the examples of in the subsequent section.
  Both the output stream and the reaction on errors can be changed simply by redefining the macros `ASSERTSTREAM` and `REACT_ON_ERROR`, respectively. Note, however, that due to the definition of assertAndReact as an expression, `REACT_ON_ERROR` must be an expression, too.
- There are two restrictions on `forall` and `exists` conditions: They cannot be used as conditions in if statements, and they cannot be used inside a parenthesized subexpression. To work around the following:
  ```
  require( constraint_1 && ( constraint_2 || forall((int . . .)) );
  ```
- you have to write:
  ```
  require( constraint_1 );
  require( constraint_2 || forall((int . . .) );
  ```
- There may also arise the need for adapting the QAssert.h macros to the specific needs of your project, e.g in the following way:
  ```
  #define assertAndReact(errorId,line,file,errorInfo)\
     YourErrorHandler::errorHandler().\
        raiseError(errorId, line, file, errorInfo)
  ```

## QSubtype.h

The ABS++ macro package (in file QSubtype.h) provides some simple macros that correspond to Eiffel's DbC philosophy and approximate also its syntax. It consists of two logical layers:

- The basic assertion layer defines the four kinds of clauses that Eiffel provides for formulating constraints on methods, classes, or statements:
- require for preconditions;
- ensure for postconditions;
- `invariant` for class invariants; and
- check for non-contracting constraints.
  Furthermore we use
- `LoopInvariant` for loop invariants;
- InitVariant and LoopVariant for loop variants.

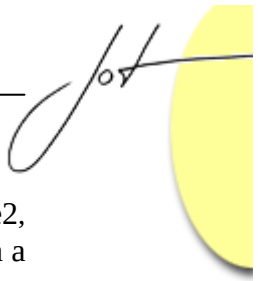
Via `CONTRACT_LEVEL` you can define check levels similar to Eiffel:

```
//------------------------------------------------------------------------
// CONTRACT_LEVEL - this defines the various levels of checking.
//    (see p.133 of "Eiffel: the language" by Bertrand Meyer)
//------------------------------------------------------------------------
//
#define  CHECK_NOTHING    0
#define  CHECK_REQUIRE    1   // require
#define  CHECK_ENSURE     2   // require + ensure
#define  CHECK_INVARIANT  3   // require + ensure + invariant


//CONTRACT_LEVEL - the level of checking to be used for next compilation.
//
#ifndef CONTRACT_LEVEL
//#define CONTRACT_LEVEL CHECK_NOTHING
//#define CONTRACT_LEVEL CHECK_REQUIRE
//#define CONTRACT_LEVEL CHECK_ENSURE
#define CONTRACT_LEVEL CHECK_INVARIANT
#endif
```

In addition to and independent from this Eiffel-like layering for contracting assertions, in ABS++ you can also define data assertions using `check` macros which also provide different levels, as well as constraints for loops.

Examples for the usage of the check feature can be found e.g. in Anything.cpp or at the begin of ABS.cpp in conntection with the iterator tests, and for loops a little bit later in this paper. You have control over these checking mechanisms if you e.g. `#define CHECK_LEVEL 1` or `#define CHECK_LOOPS` according to your needs.

- The subtype control layer macros enable you to write
  - Preconditions (checkPreconditions),
  - Postconditions (checkPostconditions, upwardsCheck),
  - class invariants (checkInvariants, checkInvars), and



- subtype checks (checkPreSubtype, checkPreSubtype_P1, checkPreSubtype2, checkPostSubtype, checkPostSubtype2, checkInvSubtype, checkInvSubtype2) in a comfortable way.

Note, that you do not need to bother with the invariant checking macros: they are implicitly used by the pre- and postcondition checks. Do also note that it is straightforward how to extend the subtype checking macros to deal with more than two parent classes if you ever will want to use multiple (not simple code inheritance but) subtyping, or to properly handle more than one method parameter according to the following scheme. (This is a stupid thing, but somehow you must provide the intelligence that is otherwise included in tool algorithms.)

```
#define checkPreSubtypeM_PN(myCond, superCond, par1, …, parN, \
          file, line, myClass, superClass1, …, superClassM)
```

Where:
`M` >= 2: number of direct parent classes in case of multiple subtyping, and
`N` >= 1: number of parameters.

Our general checking policy is as follows:

- Each of the `checkXXX` blocks can have several clauses all of which we want to check in order to get as much error information as possible. Therefore, we do not stop at the first clause that has been found to fail, but continue checking all clauses. So we need a `checksOkay_` flag for each of the `checkXXX` blocks for making the final decision if an error has been detected in the sequence of its clauses.
- Each class with pre- or postconditions has to provide a function with the name `classInvariant`, which is called from both the `checkPreXXX` and `checkPostXXX` macros in an appropriate way. This is the only thing you have to do for class invariant checks.
- The preconditions of a method are checked only in case that the class invariant has been found valid, and the class invariant is checked at the end of a method only if its postcondition has been found valid.
- Contrary to Eiffel, where non-contracting data assertions can be activated only on top of the other three kinds of assertions (i.e. only if require, ensure, and invariant have already been switched on), ABS++ keeps such data assertions via check completely independent from contracting assertions and also provides a separate level mechanism for them), in order to eventually avoid runtime penalties.
- In postconditions it sometimes makes sense to compare the new value of a variable with the old one, i.e. we want to check a certain relationship *x' = f(x, …)*, with *x'* denoting the new value. Contrary to Eiffel, C++ does not provide a mechanism that saves the old value before the execution of a method's body; therefore, it is up to you as the programmer to do that. Using QSubtype.h you can use the old-mechanism in a very simple way as shown in the following code (from C.hpp):

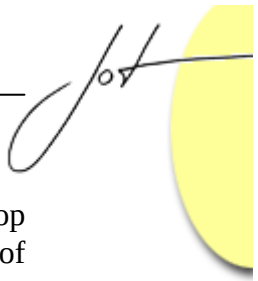
```
// ------------------------------------------------------------------------
virtual bool fooPre(int aParam, _DefFL_) const
{
    saveold("seqLen" , seqLen);
    saveold("aLong"  , aLong);
    int j = 0; // NOTE: You MUST declare the loop variables here in order to
               //       have them ready for output !!

    // check for nonnegative parameter & bounded range
    checkPreconditons(
       require(aParam >= 0, " preCond1 failed in C::foo");
       require(
           forall((j=0; j<seqLen; j++),
                  1 <= natSequ[j] && natSequ[j] <= seqLen),
           ": for j=" << j << ": natSequ[j]=" << natSequ[j] << " in C::foo!"
           );
       require(implies(true, true),
              ": Demo of implication usage in C::foo!");
    , file, line, C);
}

// ------------------------------------------------------------------------
virtual bool fooPost(_DefFL_) const
{
   // check for special sequence &  some unchanged values
   checkPostconditons(
        ensure(forall((j=0; j<seqLen; j++), natSequ[j] == seqLen - j),
            " for j=" << j << "! ( =" << natSequ[j] << " ) in C::fooPost"
            );
        ensure(old("seqLen") == seqLen,
               " seqLen has changed in C::foo to " << seqLen);
        ensure(old("aLong")  == aLong,
               " aLong has changed in C::foo to " << aLong);
    , file, line, C);
}
```

In ABS++ you can easily turn on and off (in part (1) of Qsubtype.h)

- a trace of the assertion mechanism, either for debugging this mechanism itself, or to gain a better understanding of it by activating the appropriate definition of the `TRACE_CONTRACT` macro;
- the check of invariants in preconditions (in case you are sure that you need not check for the indirect invariant effect; see [Meyer97]);
- the data assertions mentioned above using the `check` and the `CHECK_LEVEL`, and the `CHECK_LOOPS` macros mechanisms as desired;
- the checking of the contracting assertions on three increasing severity levels:
- (a): preconditions, (b): (a) + postconditions, and (c): (b) + class invariants;
- on a per class basis all kinds of contracting assertions by appropriately initializing the public `static bool _ABSppChecksEnabled_` attribute. In the light of subtype checking however, this is a possible but not very meaningful strategy.

With ABS++ you can easily

- redirect the output of ABS++ via the `ASSERTSTREAM` macro in Qassert.h (in the example I use `cout` or `ABSppLog` connected to a file in ABSppGlobals.h).
- adapt the `REACT_ON_ERROR` macro in Qassert.h to your preferred error handling strategy by redefining it to a call of a project specific error handler (in the example I



simply use `exit(333)` or `WaitForEnterKey()` which does not immediately stop after an assertion violation, but allows to continue, mainly for testing purposes of ABS++ itself.

- handle recursive function calls correctly: Due to the separation of the contracts from the implementation, the recursion naturally is in the `fooImp`; so only the outermost call is checked against the contract.
- follow the "Don't check the checker" rule: Just call `fooImp` instead of `foo`. This may eventually interfere with the visibility constraints if called from outside the implementing class; if this is the case, you might provide a suitable query for clients [Mitchell02, ch. 2]. Remember: Contracts have to be side-effect free!
- (A feature of a class, i.e. an attribute or a method, is either a query (which yields information about but does not change the visible properties of an object) or an command (which might change the object but does not return a result). The idea is that queries do not need any precondition, nor does it guarantuee anything: by definition it must not change an object's state.)
- start into an already running project: Doing things correctly, there should be no troubles.

## Some small examples

- We start with a small and simple example for loop checking (the loop variant construction has been adapted from [Marin96]):

```
int k;
int total;

total = 0;
InitVariant(k);

for (k=1; k <= 10; k++) {
   total += k;

   // check invariant: total == sum of all integers from 1 to k ?
   LoopInvariant((1 <= k && k <= 10 && total == k*(k+1)/2),
                 "loop invariant for k !!! (total = " << total
                 << "/k = " << k << ")"
                );
   LoopVariant(10-k, "loop variant is " << 10-k
               << "(for k = " << k <<")", k);
   }
```

By placing the loop variant and invariant at the end of the loop body you only loose their check after initialization. However, it is reasonable to assume that a wrong initialization will have some severe impacts on the calculation of the loop's body and will thus not go undetected through the check at its end. Though you will not get the exact Eiffel behavior in that way, you will be able to work with a sufficiently close approximation to it.

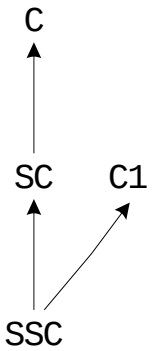
- Here is an example for the nested usage of quantifiers (but perhaps not necessarily for clever C++ data structures) that checks if all possible triples are there:

```
   forall((i = 0; i < MAX_i; i++),
   forall((j = 0; j < MAX_j; j++),
   forall((k = 0; k < MAX_k; k++),
      exists((int ii = 0; ii < MAX_i; ii++),
      exists((int jj = 0; jj < MAX_j; jj++),
      exists((int kk = 0; kk < MAX_k; kk++),
         (*table->operator()(ii,jj,kk)).i == i &&
         (*table->operator()(ii,jj,kk)).j == j &&
         (*table->operator()(ii,jj,kk)).k == k
        )))
      )))
```

- Here is an example that shows how subtype checks over the class hierarchy have to be written for the precondition of method foo of class `SC` which is a subclass of `C`, using one parameter:

```
virtual bool fooPre(int aParam, _DefFL_) const
{
   int i=0, j=0;
   checkPreSubtype_P1(
      require(aParam >= 0, " preCond1 failed in SC::foo");

      // check for permutation containing all of 1,2,3, ... seqLen
      require(
         forall((i=1; i<=seqLen; i++),
            exists((j=0; j<seqLen; j++),
               natSequ[j] == i
              )
            ),
         ": i = " << i << " is missing in SC::foo!"
         );
      , fooPre, aParam, file, line, SC, C);
}
```

## The demonstration example




Let us have a short look on how these principles are used in the attached example files where you can find the main program together with a sample output for which the assertions have been configured in a way to give a complete walk through. Thus you can trace how assertion checking is done across the class hierarchies according to DbC and subtype requirements.

In order to facilitate understanding I have chosen a very simple class hierarchy as follows: we have two base classes C and C1, with SC inheriting from C, and SSC from SC and C1, i.e. we work with multiple inheritance. The consequence of this you can see in the implementation methods of the class SSC, viz. `SSC::ClassInvariant`, `SSC::fooPre`, and `SSC::fooPost`, where you have to check for both ancestor classes, i.e. C1 and SC. Note that, while traversing across the class hierarchy is done in the implementation methods, the invocation of the class invariants and the checks for the instrumentation level is done in the Qsubtype.h macros.

Do also be aware that the contents of the pre- and postconditions and of the invariants in our example serve mainly for documentation purposes. The §-ed comment

lines are bordering those parts of the program text that deal with DbC stuff: Do not get fooled by their size compared to the rest, but take into account that, practically, we do not have any real code.

Scattered about the code there are lines marked with '`failure# # # # # # #`'; they contain statements that cause the error as indicated in the comment above it. The kind of error occurs only when the activation of the corresponding statement is done starting from the original source, i.e. assertion violations are – obviously - not independent from each other. Eventually, you will thus have to do a short play for producing them.

## 5 SUMMARY AND CONCLUSIONS

Following the DbC paradigm in the way as proposed here has some remarkable advantages:

- You do not need a tool in addition to the C++ compiler of your choice.
- You do not need to learn another language or pseudocomment syntax: Following the method proposed in this paper you write normal C/C++ code. The only thing you have to take care of is to place and type things appropriately; but this can be done in an easy and schematic way using the template routines of the example.
- You can use quantified assertions explicitly. Thus you can translate predicate logic specifications of e.g. BON, Catalysis, or Syntropy (which you may have formulated in OCL, eventually) directly into code.
- You gain *seamlessness*, i.e. it is possible to use a single notation and a single set of concepts throughout the software life cycle, from analysis and design, to implementation and maintenance.
- You may also overcome the programmer's usual resistance against what they feel excessive documentation requests burdened upon them by management, an emotion that - we should frankly admit it - is not unknown to most of us. Where does this come from? Rather sooner than later in the design phase, experienced software engineers tend to write down syntax fragments instead of prose, because at some point they start to think mainly or also in programming language constructs. Using assertions they can do just that, normally gaining a lot of benefits of both psychological (they feel good because they write down what they can use immediately), but also of technical kind (they save the error prone activity to translate everything from ambiguous natural-language texts or UML diagrams to exact, but executable specifications in a programming language). And managers should not forget that, in addition to the just mentioned advantages, software contracts also can provide documentation which is much more exact than just pure prose. As always, a good mixture of the two will yield the best results.
- You can - and this is the most important of all the benefits - apply a win-win strategy using assertion based software engineering with obvious advantages for both your customer and you as the developer of a software system.

Applying the ideas underlying DbC can help us a lot on the way towards correct and reliable software:

- Writing contracts from the very beginning enforces developers to state what they are trying to do already during design. This definitely helps doing it right!
- Assertion based programming in general, and DbC as a systematic and well defined variant of it especially, should be regarded as built in self tests and as a permanent online review that both help in early error detection and give the software developers a reasonable amount of security about the correctness status of their programs.
- Contracts can, and usually should, also serve as a basis for (technical) documentation, especially for an up-to-date description of the public interfaces.
- Contracts enable the top developers to express the intent behind their designs and hence to leave behind a clear statement of their original concepts, reducing the risk that further contributors or maintainers will destroy the software's consistency and quality.
- Contracts are a useful testing and debugging tool: (i) they save debugging time due to the improved observability, where failures occur close to the bugs; (ii) they express unambiguously what an author expects and what (s)he guarantees in turn. So one has a clear statement not only of *What Is* (in the implementation part) but also of *What Should Be* (in the assertions part). And only under such circumstances one can definitely identify a discrepancy between the two.; and (iii) they help you in designing and performing your tests.

My main aim was to put emphasis on the benefits of assertion based software development, as well as to provide a basic framework for C++ to start with. As it is usual not only for a first version, this framework is open for extensions and improvements. Some major topics not handled are:

- a more sophisticated Anything class for saving the "old" values;
- coding conventions for enforcing a 'result style', i.e. the possibility to check the calculated result of a non-void method immediately before it will be returned to the caller;
- considerations on the integration possibilities of a rescue mechanism;
- a more explicit integration of an exception handling mechanism. This, usually, depends on the project specific coding conventions. ABS++ can eventually be adapted to your needs by a suitable definition of the `assertAndReact` and `REACT_ON_ERROR` macros in Qassert.h.
- As alternative consider the adaptation of a scheme like the one below. But be aware that properly designing an application for exception handling is a topic of its own.

```
void foo()
{
   assert(PreCondition);
   try {
      fooImp;
      }
   catch (const fooException fooE) {
      // rethrown to client; part of the interface
      throw;
```

```
        }
    catch (...) {
        // catch exceptions that are not part of interface;
        // they violate Postcondition
        throw PostconditionFailure();
        }
    assert(Postcondition && Check_Invariants);
}
```

## REFERENCES


[Abrial96] J.-R. Abrial: *The B-Book: Assigning Programs To Meanings*, Cambridge University Press, 1996.

[ADL] Assertion Definition Language (ADL) at http://adl.opengroup.org/

[AssertM] AssertMate at http://www.cigital.com/

[Binder99] R. Binder, The Percolation Pattern – Techniques for Implementing Design by Contract in C++, *C++ Report*, May 1999, 38-44.

[Binder00] R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 2000.

[BowenFM] Formal Methods Home Page at http://www.afm.sbu.ac.uk/fm/

[Chen00] Y. Chen/B.H.C. Cheng, A Semantic Foundation for Specification Matching. In G.T. Leavens and M. Sitaraman (Eds.), *Foundations of Component-Based Systems*, Cambridge Univ. Press, 2000, 91-109.

[Cook94] S. Cook/J. Daniels, Designing *Object Systems: Object-Oriented Modelling With Syntropy*, Prentice Hall, 1994.

[D'Souza99] D. D'Souza/A.C. Wills, *Objects, Components, and Frameworks with UML. The Catalysis Approach*, Addison Wesley, 1999.

[Escher] Perfect Developer, at http://www.eschertech.com/index.php

[Floyd67] R.W. Floyd, Assigning meaning to programs. In *Proc. of Symposia in Applied Mathematics, Mathematical aspects of computer science Vol. 19*, American Mathematical Society, 1967, 19-32.

[Gamma95] Gamma, E./Helm, R./Johnson, R./Vlissides, J., *Design Patterns*, Addison Wesley, 1995.

[Hoare69] C.A.R. Hoare, An axiomatic basis for computer programming, *Communications of the ACM*, Oct. 1969, 576-580.

[Hoare73] C.A.R. Hoare, Hints on Programming Language Design, Stanford University Artificial Intelligence memo AIM224/STAN-CS-73-403. Reprinted in [Hoare89], 193-214.

[Hoare89] C.A.R. Hoare/C.B. Jones (Eds.), *Essays in Computing Science* (reprints of Hoare's papers), Prentice Hall, 1989.

[Jass] Jass at http://semantik.informatik.uni-oldenburg.de/~jass/

[Karaorman99] Karaorman, M./U. Hölzle/J. Bruno, jContractor: A Reflective Java Library to Support Design By Contract, in: Proceedings of Meta-Level Architectures and Reflection, 2nd International Conference, Reflection '99. Saint-Malo, France. *Lecture Notes in Computer Science* #1616, Springer Verlag, 1999, pp. 175-196.

[Kramer98] R. Kramer, iContract - The Java™ Design by Contract™ (formerly available at http://www.reliable-systems.com/tools/iContract/iContract.htm)

[Liskov88] Liskov, B., Data Abstraction and Hierarchy, *SIGPLAN Notices* **23**(5), May 1988.

[Liskov94] Liskov, B./Wing, J.M., A behavioral notion of subtyping, *ACM Transactions on Programming Languages* **16**, 1811-1841, November 1994.

[Liu04] Liu, Shaoying, *Formal Engineering for Industrial Software Development. Using the SOFL Method*, Springer, 2004

[Maley00] Maley, D. Mind the Gap – Formalising the Development of Scientific Software. Ph.D. Theses, Queen's University of Belfast, March 2000.

[Mannion98] Mannion,M./Philips,R., Prevention Is Better Than A Cure, *Java™ Report*, September 1998, 23-36.

[Marin96] Marin, M.A., Effective use of Assertions in C++, *ACM SIGPLAN Notices*, Nov. 1996 (also available at http://www.cs.rpi.edu/~wiseb/sigplan/issues/ctb9611.ps ).

[Meyer92] Meyer, B., Applying "Design by Contract", *IEEE Computer* **25**(10), 40-51, October 1992.

[Meyer97] Meyer, B., *Object oriented software construction*, 2nd Ed., Prentice Hall, 1997.

[Mitchell02] Mitchell, R./McKim, J., *Design by Contract, by Example*, Addison-Wesley, 2002.

[Mueller94] Mueller, H.M., Powerful Assertions for C++, *C/C++ Users Journal*, October 1994, pp. 21-37

[Payne97] Payne, J.E./Alexander, R.T./Hutchinson, C.D., Design-for-Testability for Object-Oriented Software, *Object Magazine*, July 1997, pp. 35-43.

[Payne98] Payne, J.E./Schatz, M.A./Schmid, M.N., Implementing Assertions for Java, *Dr. Dobb's Journal*, January 1998 (also available at http://www.ddj.com/ddj/1998/1998_01/payn/payn.htm )

[Plösch02] Plösch, R., Evaluation of Assertion Support for the Java Programming Language, *Journal of Object Technology*, vol. 1, no. 3, special issue: TOOLS USA 2002 proceedings, 2002, pp. 5-17.

[Plösch04] Plösch, R., *Contracts, Scenarios and Prototypes. An Integrated Approach to High Quality Software*, Springer, 2004

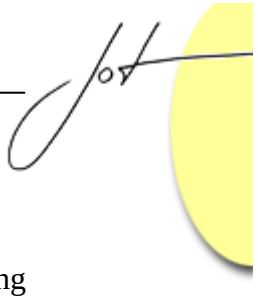
[Porat95] Porat,S./Fertig, P., Class assertions in C++, *J. of Object-Oriented Programming*, May 1995, pp. 30-37.

[Rist95] Rist, R./Terwillinger, R., *Object-Oriented Programming in Eiffel*, Prentice Hall, 1995.

[Toth05] Toth, H., On theory and practice of Assertion Based Software Development, in *Journal of Object Technology*, vol. 4, no. 2, March-April 2005, pp. 109-129, http://www.jot.fm/issues/issue_2005_03/article2

[Walden95] Walden, K./Nerson, J.-M., *Seamless Object-Oriented Software Architecture. Analysis and design of reliable systems*, Prentice Hall, 1995 (see http://www.bon-method.com )

[Welch98] Welch,D./Strong, S., An Exception-Based Assertion Mechanism for C++, *J. of Object-Oriented Programming*, July/August 1998, pp. 50-60



## About the author

**Herbert Toth** is a senior software engineer in the Program and System Engineering Department of SIEMENS AG Austria. He holds a diploma degree in computer science from the Technical University, and a Ph.D. degree in mathematical logic from the University, both in Vienna. During the eighties and nineties his research interests led to some publications on the foundations of fuzzy set theory. He can be reached at mailto:herbert.toth@siemens.com.