
Application invariants: Design by Contract augmented with deployment correctness logic



Anthony Savidis^{*,†}

Institute of Computer Science, Foundation for Research and Technology—Hellas, Heraklion, Crete, GR-71110, Greece

SUMMARY

Design by Contract is a method for the development of robust object-oriented software, introducing class invariants as conditions corresponding to the design axioms that should be satisfied by every valid instance of a class. Additionally, the method states formally the way client programs should correctly utilize supplier classes, so that the composition of correct programs may be accomplished. However, the contextual correctness of supplier instances within client programs, only reflected in the client-specific semantics for supplier-class deployment, cannot be expressed through Design by Contract. For instance, supplier instances satisfying the supplier class invariant may not constitute plausible supplier instances in the context of a particular client program. In this context, we introduce application invariants as an extension to Design by Contract, for hosting the contextual-correctness logic for supplier instances, as conditionally defined by client programs. This allows stronger validation of supplier instances, through the dynamic encapsulation of client-specific acceptance filtering, enabling more intensive defect detection. Application invariants are implemented in the context of client classes as methods utilizing correctness condition expressions, are dynamically hosted within supplier instances, while always called by supplier instances when the basic supplier-class invariant test is performed. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: Design by Contract; defensive programming; embedded correctness checking; application invariants

INTRODUCTION

Design by Contract [1] is an object-oriented software design recipe, driving the implementation of self-checking classes through computable conditions that express the conformance to the class design semantics. The key deployment concept of Design by Contract is related to formalized obligations of both client and supplier classes so as to facilitate the composition of correct programs. Supplier classes

*Correspondence to: Anthony Savidis, Institute of Computer Science, Foundation for Research and Technology—Hellas, Science and Technology Park of Crete, Heraklion, Crete, GR-71110, Greece.

†E-mail: as@ics.forth.gr

are designed as providers of functions to client programs, explicitly specifying and implementing the following condition expressions:

- *preconditions* to public functions, which should evaluate to true, i.e. satisfied, so that a legal call can be made by clients;
- *postconditions* to public functions, which should evaluate to true after legal calls are made by clients;
- class *invariant*, satisfied by all instances of a class through which clients made legal calls.

The usage regulations for supplier functions, inherent in the previous definitions, establish a well-defined agreement between suppliers and client programs, commonly known as the *contract*. Formally, using the postfix symbol \vdash to denote satisfaction of an expression, the contract states that, \forall instance α and public function f of a class A

$$\alpha.\text{preconditions}.f \vdash \wedge \alpha.\text{invariant} \vdash \Rightarrow (\alpha.f \Rightarrow \alpha.\text{postconditions}.f \vdash \wedge \alpha.\text{invariant} \vdash)$$

In other words, if a public function is called with an instance whose invariant and respective precondition are satisfied exactly *prior* to the call, the invariant and respective postcondition will also be satisfied exactly *after* the call is thoroughly completed. From the design-logic point of view, invariants expose the correctness regulations for class instances, known also as axioms, rephrasing instance validity in the form of program expressions. Similarly, preconditions express the required instance state for eligible function calls, while postconditions express the internal state transition due to function calls. Consequently, the design semantics relating to instance validity, conditional function availability and required function behavior are explicitly encapsulated within designed classes in a directly computable form. This design technique promotes the implementation of self-checking code, enabling the detection of the following errors through assertions of design semantics.

- *Supplier deployment defects*, occurring if the client breaks the contract, e.g. precondition checking is dismissed prior to function calls. In this case, the behavior of the related function becomes undefined, implying that either a system crash will be caused inside the function or that the post-call invariant assertion may fail.
- *Supplier implementation defects*, appearing if the invariant fails following a legally called function. In this case, a class implementation error is signified, i.e. a typical design-coding mismatch. The offending statements causing an algorithmic deviation from the correct design logic should be manually investigated inside the subject function.

It should be noted that Design by Contract does not pose any restrictions on design scenarios where client logic may be registered and later called back by supplier instances. In this case, the behavior depends on the design choices made for the particular supplier class: if it should allow clients to implement callbacks that directly or indirectly call supplier functions, then apparently the supplier-class invariant should always hold before and after callback dispatching.

Client aspects of supplier-deployment correctness

While precaution for deployment errors is explicitly taken in Design by Contract, it is restricted to eligibility tests for member calls, defining basically when instances are capable of accepting function

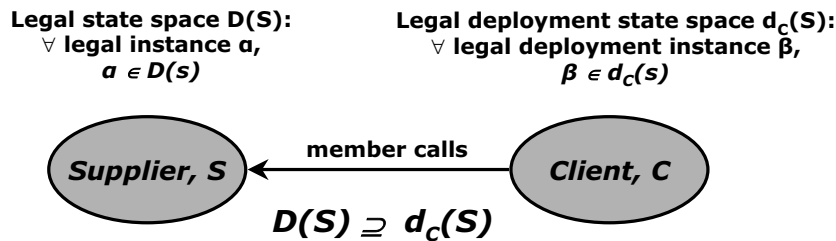


Figure 1. The different views of deployment correctness for supplier instances, as seen from the supplier and the client perspectives.

call requests, i.e. they merely express *call readiness*. Consequently, such deployment errors address only readiness violations, while particular deployment regulations for supplier classes, reflected in the client design logic, cannot be accommodated in the context of supplier classes. More specifically, even though the space of legal supplier states $D(S)$ effectively populates all instances resulting from legal function calls, the particular client design logic may only consider a smaller subset $d_C(S)$ as legal deployment instances (see Figure 1). For instance, let's consider a *Stack* supplier class for integers used by a client program to store at most 20 prime numbers. Clearly, this deployment logic of the *Stack* supplier class can be formally expressed as an application-specific invariant, to make more restrictive correctness tests of *Stack* instances in the context of the particular client program, using the following condition: $size \leq 20 \wedge \forall i \in (0, size] prime(stack[i]) \vdash$, where *prime* denotes the mathematical condition for prime numbers.

Following the previous example, apparently the client-side correctness logic cannot be embedded in the original supplier class; hence, it may only be encapsulated at client class side. Additionally, since all supplier instances encompassed in the client class constitute part of the client state, following Design by Contract, it is necessary to embrace the related supplier-state validity regulations inside the client class invariant. In this framework, when trying to accommodate client-specific supplier-deployment correctness in Design by Contract, the following software engineering issues emerge.

- Client classes may also play the role of supplier classes in component-based application architectures, offering a set of public functions reflecting their particular architectural role. Such client classes may utilize multiple supplier classes, thus embodying explicit tests of the various supplier-deployment invariants in their basic set of public functions. However, during runtime, such client functions normally engage in the execution of a considerable number of program statements, as they implement higher-level architectural operations. Effectively, since assertions must be only injected inside the basic client functions, if supplier-deployment invariant failures occur, the range of potentially offensive statements is practically so wide that Design by Contract fails to support direct defect detection.
- To better support direct defect detection, it is necessary to enclose the various source code fragments through which supplier instances are modified, with the corresponding invariant assertions. Since this directive is not part of the Design by Contract prescription, one might consider the implementation of smaller wrapper functions for such code fragments, subsequently

embedding the necessary invariant checks. However, decomposing clients in more functions just to ensure deployment invariant checks are performed in between a relatively smaller number of runtime statements is a design directive of questionable value that may conflict with other design principles.

- There is always a significant part of software applications dedicated to gluing and coordinating together all the rest of the architectural components, not programmed as such in the form of a supplier class, as it is not intended for direct deployment by other components. In most system libraries such components are known as ‘application classes’, usually offering a single public function commonly named ‘run’, being actually the application execution entry point. However, since such application classes do not provide a programming interface fitting the required profile to apply Design by Contract, we need alternative ways to test deployment correctness of the constituent supplier instances during execution. Although every instance I of a supplier class S created by the application class will be checked upon method invocation for satisfiability of its class S invariant, there is no entry point to call the specific correctness logic of I inside the context of the application class. The main reason for this ‘design gap’ is the lack of any guidelines in Design by Contract to test the correctness of application logic when this is not delivered as a supplier class.
- Since client classes may normally utilize multiple supplier classes, to overpass the potential problems of Design by Contract for direct defect detection of supplier-deployment invariant failures, one might heuristically encapsulate invariant assertions in all cases of supplier use. Effectively, this leads to source code growth that is dependent on the supplier deployment demands, while also increasing the potential source code pollution with largely repeating invariant tests. Alternatively, the design of client-side delegate functions with embedded invariant checks, for every referenced supplier function, may help to eliminate source code pollution, but leads to an unnatural supplier-interface replication at the client side.

At this point, a seemingly rationale alternative approach would be to incorporate the prime checking logic in a class directly inheriting from stack, while extending the invariant of the derived class according to the prime checking condition. Clearly, the generalization of this approach is to accommodate client-side correctness regulations through alternative subtypes of the corresponding supplier class. Now lets hypothetically extend a little the correctness regulations of the prime stack as follows:

- each element has a property defined by a client-defined method;
- every two elements are related with a client-defined method;
- having at most N elements or at least M elements;
- successive elements form a numeric sequence (e.g. Fibonacci);
- can have arbitrary combinations of the above deployment properties.

Apparently, the birth of derived classes for each deployment regulation and for all the various combinations, although implementationally feasible, turns out to be totally impractical as it leads to an explosion of fragmented derived classes. This introduces an extra overhead for software maintenance and re-factoring, as programmers had to decide how to manage the variations of supplier classes (i.e. is it treated as a supplier library extension or as client-specific code?).

Another key deficiency of the inheritance-based approach lays in the fact that it cannot directly accommodate dynamically varying deployment regulations. For instance, assume a stack of prime

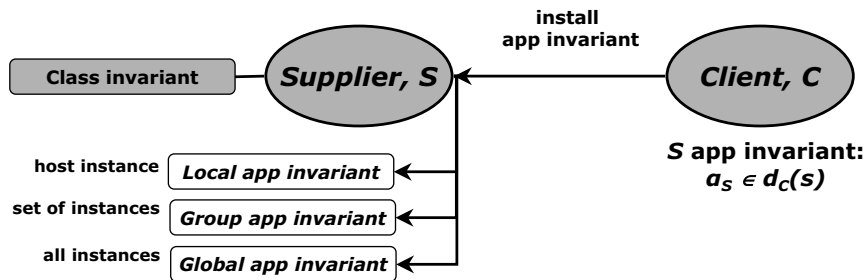


Figure 2. The role of application invariants as invariants hosted by suppliers, while implemented and installed by clients.

numbers that should be restricted after a certain point in time during execution, to have either at most N elements or at least M elements. Since the decisive condition is not known at compile time, the only feasible method through inheritance is to program as follows: create a new stack of the desirable derived type, copy the elements of the old stack and then destroy the old stack. Clearly, this introduces both runtime and development time overheads.

Overall, the key emphasis of application invariants, in a way similar to the Design by Contract method, is not put on feasibility but on software design quality, i.e. the design method optimally suits the design intention. In this context, application invariants are proposed as an effective design recipe for the following design need: *checking correctness violations of supplier instances according to client regulations.*

The role of application invariants

Application invariants are a purposeful extension to Design by Contract so as to effectively accommodate supplier-deployment invariants that are defined by client classes. Application invariants are hosted and asserted dynamically by supplier instances, while being implemented in the context of client classes.

Application invariants may be *global*, i.e. they concern all instances of a supplier class, *local*, i.e. they are associated to a particular supplier instance, or *group*, i.e. they concern only a set of supplier instances (see Figure 2). Supplier classes should offer an appropriate programming interface for the runtime registration of application invariants by client classes, while the supplier class invariant is redefined, becoming more restrictive:

$$invariant_{old} \wedge local\ app\ invariant \wedge global\ app\ invariant \wedge group\ app\ invariant$$

The key software engineering advantages of application invariants are:

- minimal implementation implications on client classes to support checking of supplier-deployment correctness, since only the programming and runtime registration of application invariants is required;

- runtime checking of supplier-deployment correctness is intensively performed every time supplier functions are called, thus enabling the detection of potential defects more close to their particular source;
- client implementation defects, such as design errors, coding mistakes and memory-corrupting statements, are easily identified through the client-specific supplier-instance state validation;
- the technique is applicable as a powerful test-first and defect diagnosis programming method, i.e. for defensive programming, even in case Design by Contract is not applied.

Global application invariants concern deployment properties that are applicable to all supplier instances of a client application. For instance, in an application making socket connections only with local processes, a global application invariant would express the condition that: *the address of the connecting process of the socket instance is the same as the local host address.*

Local application invariants aim to guard properties being unique to specific supplier instances. For instance, in a C compiler, there can be at most one entry in the symbol table of type function and name 'main'. In this case, the local application invariant would define that: *the name of the symbol is 'main', its type is 'function', and it has two optional arguments of type 'int' and 'char**'.*

Finally, group application invariants express properties common to multiple instances of a particular supplier class, thus enabling detection of broken conditions for any instance of the group. For example, all instances of a list class in a compiler application, which are dedicated to storing the formal arguments of functions, may be bounded by a group application invariant which has access to the variable holding the largest number of formal arguments. Such an invariant would express: *the size of the list instance is less than or equal to the content of the integer variable whose reference is stored in the invariant.*

It should be noted that there is no restriction on the combination of the previous categories, as this is always a matter of the client design logic. More specifically:

- there can be multiple global application invariants per supplier class;
- a single instance may be subject to multiple group invariants;
- a single instance may be subject to multiple local invariants.

Following the previous modified definition of the basic class invariant to actually encapsulate the satisfiability test for application invariants too (through the logical 'and' operator, thus leading to a stronger refined class invariant), it becomes clear that the application invariant test is performed during execution inside the basic class invariant test. Since due to the Design by Contract regulations the class invariant should always be satisfied before and after calling public, and some design-decided private or protected methods, it follows that application invariants will also be called in the context of the runtime invariant check points. Finally, according to Design by Contract, the class invariant may well be temporally broken in between method calls. Overall, application invariants are subject to the same satisfiability regulations as class invariants.

Throughout the rest part of the paper, C++ code examples will be employed practically to illustrate the programming method for application invariants. However, as it is also the case with Design by Contract, the proposed method is by no means specific to the C++ language, as it does not rely on any particular programming mechanisms, semantics or idioms supported only by the C++ language. However, the presented method constitutes an extension to the original Design by Contract technique, the latter being specific to object-oriented programming languages as it relies on abstract classes,

abstract methods, class inheritance and late binding. As a result, the application invariants technique is inherently dependent on, and applicable to, object-oriented languages.

The discussion continues with a detailed account of related work. Then the implementation pattern for application invariants is incrementally introduced, starting with an outline of the subject abstract data type. Next, the way application invariants can be used in client programs is shown followed by a discussion on consolidated experience for defect detection using the proposed technique. Finally, some additional extensions of Design by Contract are introduced, closing with a brief summary and the key technical conclusions.

RELATED WORK

Test-first programming

Test-first programming is a development philosophy that is met with its most representative instantiation in the context of *extreme programming* (XP) [2] (see the main XP site <http://www.extremeprogramming.org/>). Following test-first programming, test units are built first, even before the code unit to be tested is entirely implemented, while every development activity should be verified early with its corresponding test unit. The source code is intensively tested and verified, leading to a potentially robust code base that becomes easily sustainable, effectively surviving change; test units always evolve in parallel to the tested code units. Those methods are actually software development processes, primarily emphasizing software robustness and evolution, not explicitly suggesting a prescriptive programming technique for software self-checking regarding design conformance. For this purpose, more specialized methods exist such as unit tests and programmer tests.

Unit testing frameworks

Unit tests are the basic building blocks of intensive testing processes, verifying and ensuring that individual software components produce results according to their design-specific semantics, thus practically fulfilling their particular design contract. Testing frameworks provide template infrastructures for crafting unit test classes, the latter typically sending multiple messages (i.e. calls) to tested classes while expecting and asserting design-time predicted responses. Every time a tested class fails to return the expected responses, the test case is considered as failing and the specific testing transaction is appropriately recorded. Examples of powerful testing frameworks are JUnit [3], being a regression-testing framework (<http://www.junit.org>; [4]) for the Java language, recently extended with Design by Contract support (Java Modeling Language (JML) and `jmlunit`, <http://www.cs.iastate.edu/~leavens/JML/index.shtml>), and the CppUnit framework (<http://sourceforge.net/projects/cppunit/>), which is actually a C++ porting of the JUnit framework. Unit testing frameworks provide the programming ingredients for test-first programming, enabling massive tests to be incorporated in comprehensive test suites that may run in batch mode. However, in comparison to Design by Contract, unit testing does not focus on design-based verification, but merely on the design of representative tests which aim at catching programming faults. In this case, test units always reflect partial correctness checking, relying on the statistical hypothesis that the robustness of

software is proportional to the inverse number of the failed tests, in comparison to Design by Contract emphasizing self-checking according to design-conformance semantics.

Defensive programming

Defensive programming concerns prescriptions for increased program robustness through techniques for embedded bug defense, that can be employed orthogonally to software development processes. One of the key goals in defensive programming methods is *direct defect detection* (DDD) [5], aiming to minimize the time distance between the source of errors and the point of execution at which those become externally observable, by making code that includes potential errors to break instantly with appropriate informative feedback. We consider two types of commonly appearing programming errors:

- *design faults*, i.e. the design is inappropriate for the target problem, however this is only observed once the program produces other but the expected results;
- *design-coding mismatches*, i.e. the design intention is inaccurately translated to program source code.

In such cases, although the presence of software defects is apparent, it may happen that their detection during the implementation and testing processes cannot be certified, even when large-scale systematic unit testing is effectively applied. The key reason is that with typical tests we assert the program output; in such cases, if we trap incorrect values we *verify the presence of defects*, but in case we don't find erroneous output, we can never certify the *absence of defects*. This inadequacy of design-defect detection is the key concern of design-by-contact, which emphasizes *testable embedded design*. As a result, once design properties become expressed through computable expressions, the testing process does not merely verify program output, but verifies design properties, meaning design faults of any sort can be inherently detected. This particular defensive property of Design by Contract is considered to be a top-priority issue towards DDD, being the main motivating factor of the presented work towards extending Design by Contract to accommodate design logic not captured and expressed in its original definition.

In [6], the need to introduce defensive programming as a core computer science course is discussed. Design by Contract is a genuine defensive programming approach as it emphasizes the direct detection of design-deviating program behavior. Application invariants appropriately complement the Design by Contract by introducing formalized assertions engaging client-oriented supplier-deployment correctness logic, largely enhancing the defensive programming capability of the original method. An example of a defensive programming recipe concerns defensive smart pointers for the C++ language [4].

Automatic bug finders

Bug finders are tools supporting the automatic detection of potential software defects performing (a) static program analysis at development time, by matching against potentially erroneous code patterns (e.g. use of initialized pointers, broken boundary conditions, lacking fallback code in branch statements, etc.); or (b) runtime monitoring of system behavior, by trapping specific system faults (e.g. memory corruption, memory leaks, dead code, dead locks, etc.). Such tools might help a lot when none of the previously mentioned techniques is effectively applied, however, they suffer from the following drawbacks:

- they are constrained in detecting only a limited set of generic system errors;
- the source of the error is detectable only if it directly causes a system observable exception;
- while they offer limited defect diagnosis, they provide no software design therapy, meaning programmers are likely to be faced with continuously appearing errors once they fix old bugs and proceed to introduce new code.

In static analysis tools we can also consider software verification instruments (i.e. model checkers), like Spin (<http://spinroot.com/spin/whatispin.html>), which enable one to verify formal system properties (usually by testing against undesirable properties, like reachability of erroneous states). Such tools are mostly tied to formal implementation or specification techniques, like functional languages, temporal logic and state transition systems, being mostly appropriate for mission-critical small-scale software kernels; however, tools like Blast (<http://www-cad.eecs.berkeley.edu/~tah/blast/>) exist for verification of behavioral properties of C programs.

Examples of bug finders are JLint and FindBugs [7] for Java, the old Lint program for C, and the ITS4 tool [8], which are static program analyzers, as well as the Microsoft BoundsCheckerTM for C++ and Delphi, which is a runtime program monitor. Additionally, in this category we consider all defensive versions of memory managers to trap illegal memory accesses.

Design by Contract support tools

The Design by Contract method [1] is semantically encapsulated within the Eiffel programming language [9]. In this sense, the compilers of the Eiffel language directly embed semantic checking for Design by Contract conditions and automatically generate code to assert invariants during execution. However, for typical object-oriented languages like C++ or Java not directly encapsulating such contractual semantics within their original definition, programmers are required to implement the class correctness logic with explicit condition functions. As this process typically required extra coding efforts, third-party tools appeared which supported the semantic tagging of classes, enabling the injection of Design by Contract information directly into the source code units, using special-purpose specification patterns embedded within comments. Such tools parse the specifications and generate extra code for program classes that programmers would otherwise have had to explicitly produce from scratch. Representative examples of such instruments are:

- JML [10], and the accompanying tools (basically *jmlc*, the compiler) to support Design by Contract within Java;
- Spec# programming system (<http://research.microsoft.com/SpecSharp/>) [11] and the Spec# compiler, to support Design by Contract within C#, aiming to become applicable for .NET framework applications (although currently only supported for C#);
- The C² [12] is a tool to support Design by Contract specifications within the C++ class definitions, working in the same way as the previous tools, i.e. as a pre-preprocessor to the language compiler. Additionally, there is a recent proposal to add Design by Contract directly in the C++ language [13], which enumerates some key desirable characteristics, while emphasizing the criticality of Design by Contract methods for bug defense and software documentation.

Such existing instruments, supporting Design by Contract in the context of mainstream programming languages, directly adopt and reflect the regulations of the original Design by Contract method.

Formally, all such language extensions, like Spec#, JML and C², are actually split into two basic layers: (a) the bottom imperative language layer offered by the native programming language (i.e. C#, Java and C++, respectively); and (b) the top layer offering the contractual specification elements, actually supporting Design by Contract. Since application invariants constitute a typical algorithmic contract, as Design by Contract does, it is easily implementable using the imperative elements of any OOP language. In this context, we will provide an implementation pattern for C++, though not dependent on any particular C++ idioms or mechanisms. However, since the proposed approach constitutes an extension to Design by Contract, it is clear that to allow such existing tools to effectively accommodate support for application invariants through their contractual elements, they have to be necessarily extended according to the new proposed contractual features of application invariants. For instance, there is no contractual support in Spec# or JML for client-defined deployment conditions that can be attached dynamically and selectively to desirable supplier instances, on top of the original supplier-class invariants.

IMPLEMENTATION

Overview

The implementation pattern for application invariants is illustrated in Figure 3. Following the Design by Contract, every supplier class is actually distinguished amongst its abstract representation, i.e. the Abstract Data Type (ADT), and its concrete implementation-specific representation, i.e. the REP. An application invariant is actually a placeholder to manage during runtime an arbitrary number of condition expressions; we consider condition expressions for invariants to be functions accepting a single supplier instance argument, while returning a Boolean value. The basic application invariant class is named `AppInvariant` and is supplier polymorphic; in C++ this is implementable via a compile-time type-safe polymorphism, i.e. templates, while in Java this is 'backed-up' with runtime type identification. The evaluation of an application invariant, through a call to `Eval()` (or by overloading `operator()` in C++), returns the conjunction of the values of constituent condition expressions.

Next, the `AppInvariants` class, which is also supplier class polymorphic, collects the three basic types of application invariants, i.e. *global*, *local* and *group* invariants, implemented as a trivial container class. Now the `AppInvariants` class offers all the necessary functionality that a supplier class needs so as to enable its instances directly support application invariants. As a result, the supplier class `REP` also inherits from `AppInvariants`, so supplier `REP` instances encapsulate all the required inherited functionality. This design link makes supplier instances directly support the dynamic registration of deployment-correctness conditions. Finally, the original Design by Contract invariant function `Invariant()` of supplier class `REP` is logically restricted through a conjunction with the result of evaluation of application invariants, the latter returned by the call to `APPINVARIANT()`.

The supplier class ADT

For the detailed presentation of the implementation method regarding application invariants, a *Stack* supplier class is firstly defined, employing the Design by Contract method, programmed in

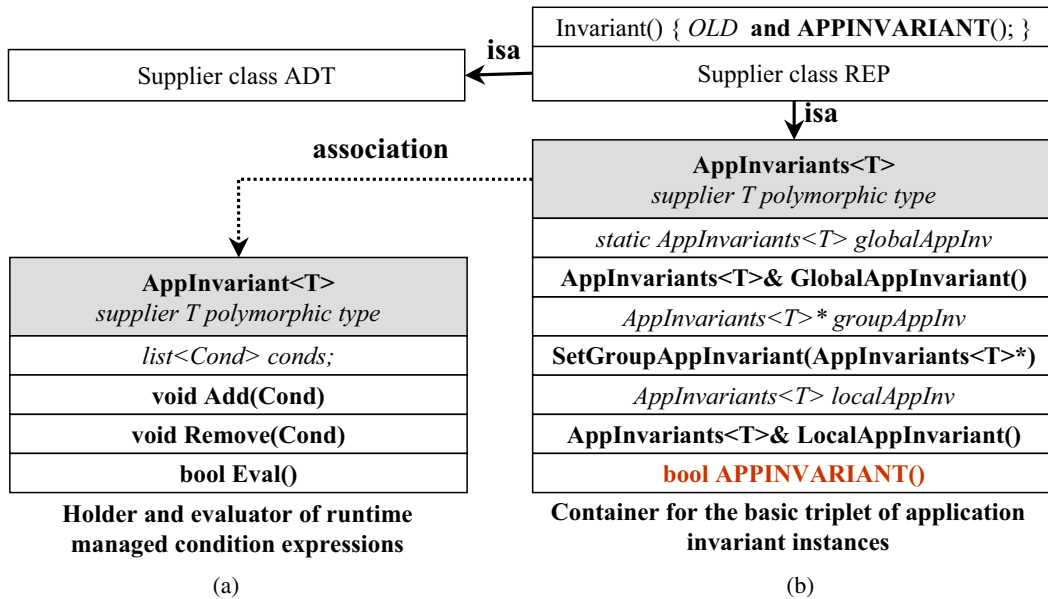


Figure 3. The implementation pattern for application invariants relying on two basic classes: (a) a single invariant, i.e. AppInvariant; and (b) the container of the types of invariants i.e. AppInvariants.

standard C++. The *Stack* ADT is defined as an abstract template class in Figure 4. As shown in Figure 4, all member functions introduced to support the Design by Contract must have the standard signature `bool (StackADT::*)(void) const`, indicating that they are merely Boolean condition expressions, not affecting the internal state of the calling instance. Additionally, they are polymorphic functions, as defined by the `virtual` qualifier, since, following Design by Contract, they are subject to formalized regulations for refinement by derived classes.

The supplier class REP

Following the definition of the *Stack* ADT, derived classes may provide concrete realizations with particular derived representations, called REP (i.e. representations) in the Design by Contract terminology. Such REP classes refine the ADT invariant by restricting its satisfiability domain with implementation-oriented knowledge. More specifically, the REP invariant is defined as a conjunction of the original ADT invariant and the specific correctness conditions for REP instances. In Figure 5, the *Stack* REP, with a single-linked list data structure, is provided. Notice that in the REP invariant, correctness checking is encapsulated for the list-linking of stack elements (i.e. see in Figure 5, expression `Down(top, total) == base`).

```

template <class T> class StackADT {
protected:
    unsigned total;
    unsigned oldTotal;
    T        lastPushed;
public:
    virtual bool IsEmpty (void) const = 0;
    virtual const T& Top (void) const = 0;
    virtual bool Push (const T&) = 0;
    unsigned Total (void) const { return total; }
protected:
    virtual bool INVARIANT (void) const
        { return total==0||!IsEmpty(); }
    virtual bool POSTPush (void) const
        { return INVARIANT() &&
            !IsEmpty() &&
            total==oldTotal+1 &&
            Top()==lastPushed; }
    virtual bool POSTPop (void) const
        { return INVARIANT() && total==oldTotal-1; }
public:
    virtual bool PREPush (void) const
        { return INVARIANT(); }
    virtual bool PRETop (void) const
        { return INVARIANT()&&!IsEmpty(); }
    StackADT (void) : total(0), oldTotal(0) {}
    virtual ~StackADT(){}
};

```

Figure 4. The StackADT template class, with Design by Contract support. The names of special purpose members for preconditions and postconditions are indicated by the PRE and POST prefixes, while the member function for the invariant is simply named INVARIANT.

Application invariants basic template class

The basic facilities for the dynamic registration and evaluation of application invariants are provided as a generic template class, which is to be employed and delivered to clients by the developers of supplier classes (as discussed in the next section). As shown in the implementation of the `AppInvariant` template class (see Figure 6), the application invariant is defined as a conjunction of condition expressions of `CondFunc` type, which can be separately registered or removed by client applications during runtime. As expected, the evaluation function, i.e. `operator()`, requires the particular supplier instance for which the application invariant evaluation will be performed. As indicated in the definition of the `CondFunc` type, within the `AppInvariant` template class, condition functions are actually pointers to functions with a global, i.e. static, linkage.

Consequently, in client programs, they will have to be defined either as global `extern` or `static` functions, or as local member `static` functions of client classes. Additionally, condition functions

```

template <class T> class StackREP : public StackADT<T> {
private:
    struct Node { T val; Node* below; };
    Node* top;
    Node* base;
    const Node* Down (const Node* p, unsigned n) const
        { return n <=1 ? p : Down(p->below, n-1); }
public:
    bool IsEmpty (void) const;
    const T& Top (void) const;
    bool Push (const T&);
    void Pop (void);

protected:
    bool INVARIANT (void) const {
        return StackADT<T>::INVARIANT()          &&
               total || (!base && !top && IsEmpty()) &&
               total != 1 || (base==top && !IsEmpty()) &&
               total <= 1 || (base != top && !IsEmpty()) &&
               Down(top, total)== base;
    }
public:
    StackREP(void) : top((Node*)0), base((Node*) 0){}
    ~StackREP() { while (top) Pop(); }
};

```

Figure 5. The StackREP having a single-linked list implementation, showing the refined REP invariant as a conjunction of the REP implementation correctness conditions and the StackADT invariant.

may be added or removed on the fly, effectively supporting scenarios where the client semantics for the deployment of supplier instances are not constant during runtime, but may vary dynamically.

Encapsulation in supplier classes

The deployment of the application-invariant template class in the context of supplier classes should reflect the requirement to support application invariants having either *global*, *local* or *group* scope.

- *Global* application invariants are applicable to all instances of their supplier class. They support scenarios in which the client-specific deployment regulations for supplier classes apply to all supplier instances.
- *Local* application invariants are applicable only to their host instance. This reflects situations in which client-deployment logic differentiates even at the instance level.
- *Group* application invariants are applicable only to a specific set of instances. This concerns typical cases where clients deploy a group of supplier instances in the same manner.

```

template <class T> class AppInvariant {
public:
    typedef bool (*CondFunc)(const T&);
private:
    typedef std::list<CondFunc> CondList;
    CondList conds;
public:
    void Add (const CondFunc& f) {
        assert(
            std::find(conds.begin(), conds.end(), f) ==
            conds.end()
        );
        conds.push_back(f);
    }

    void Remove (const CondFunc& f) {
        assert(
            std::find(conds.begin(), conds.end(), f) !=
            conds.end()
        );
        conds.remove(f);
    }

    bool operator()(const T& inst) const {
        for ( CondList::const_iterator i = conds.begin();
            i != conds.end();
            ++i )
            if (!(**i)(inst))
                return false; // Conjuncting conditions.
        return true;
    }

    AppInvariant(void) {}
    ~AppInvariant() { conds.clear(); }
};

```

Figure 6. The application invariant template class, with the evaluation function, i.e. operator(), the condition function type CondFunc, and the registration function Add/Remove. The template parameter type T denotes the supplier class.

The embedding of application invariants within supplier classes is straightforward, using the implementation of the `AppInvariants<T>` template class encapsulating the *global*, *local* and *group* functionality (see Figure 7); the template parameter T corresponds to the supplier class. Reuse is facilitated using a special inheritance pattern for template classes, through which the newly defined derived template class, e.g. `StackREP`, inherits from an instantiation of a base template class, e.g. `AppInvariants`, such that the type argument in base template instantiation is actually the newly

```

template <class T> class AppInvariants {
private:
    static AppInvariant<T> globalAppInv;
    AppInvariant<T>         localAppInv;
    AppInvariant<T>*       groupAppInv;
public:
    static AppInvariant<T>& GlobalAppInvariant (void)
        { return globalAppInv; }
    AppInvariant<T>& LocalAppInvariant (void)
        { return localAppInv; }
    void SetGroupAppInvariant (AppInvariant<T>* group)
        { groupAppInv = group; } // Null value allowed.
    bool APPINVARIANT (const T& inst) const {
        return globalAppInv(inst) &&
               localAppInv(inst) &&
               (groupAppInv ? (*groupAppInv)(inst) : true);
    }
};

template <class T>
class StackREP : public StackADT<T>,
                public AppInvariants< StackREP<T> > {
protected:
    bool INVARIANT (void) const {
        return all-expressions-as-before &&
               APPINVARIANT(*this);
    }
    ...
};

```

Figure 7. Embedding application invariants in the StackREP supplier class. A template class AppInvariants is implemented, where T is the supplier class, enabling direct re-use of application invariants implementation by all supplier classes through inheritance.

defined derived template class, e.g. the StackRep<T> class inherits from the AppInvariants< StackRep<T> > class (see bottom part of Figure 7).

Following Figure 7, the storage for *global* application invariants is *static*, effectively being shared by all supplier instances, while, as normally expected, it is *local* for local application invariants. Additionally, supplier instances encapsulate a pointer to their *group* application invariant that is externally stored; the group may be changed dynamically for supplier instances. An appropriate implementation of *group* application invariants is through singleton patterns, inheriting from the instantiation of the AppInvariant<T> template class that results by supplying as T type parameter the particular supplier class. The suggested software pattern for *group* application invariants is demonstrated with an example showing type-parameterized *group* application invariants for the StackREP class in Figure 8. As reflected in Figure 8, *group* application invariants are defined as singletons, encompassing the partial conjunct conditions as *static* functions that are explicitly

```

template <class T, const unsigned N>
class StackREPSizeGroup : public AppInvariant< StackREP<T> > {
private:
    static StackREPSizeGroup* singleton;
public:
    static StackREPSizeGroup* GetSingleton (void) {
        if (!singleton)
            singleton = new StackREPSizeGroup;
        return singleton;
    }
    static bool Invariant (const StackREP<T>& stack)
        { return stack.Total() <= N; }
    StackREPSizeGroup(void) { Add(Invariant); }
};

// Declarations of static members for instantiated
// templates.
//
StackREPSizeGroup<std::string, 32>*
    StackREPSizeGroup<std::string, 32>::singleton;
AppInvariant< StackREP<std::string> >
    SupplierAppInvariants< StackREP<std::string> >::globalAppInv;

// Instantiation and use of the template class
// for group invariants.
//
StackREP<std::string> stringStack;
stringStack.SetGroupAppInvariant (
    StackREPSizeGroup<std::string, 32>::GetSingleton()
);

```

Figure 8. The implementation pattern for group application invariants as singletons, inheriting from `AppInvariants<T>`, `T` being the supplier class, encapsulating the invariant condition expressions as static functions.

registered within the class constructor, e.g. `Add(Invariant)` call. The `StackREPSizeGroup` template class of Figure 8 covers all *group* invariants for the `StackREP` supplier class that constitute upper-bound stack-size conditions. For instance, the *group* invariant for stack instances having up to 32 elements of `std::string` type is defined as `StackREPSizeGroup<std::string, 32>`.

Turning condition functions to functor classes

For simplicity, in the code fragment of Figure 6 implementing the `AppInvariant` template class, condition expressions are implemented as normal functions. However, this approach has severe limitations. For example, functions cannot support modifiable local state, which can be appropriately altered depending on the condition evaluation calls.

```

template <class T>
class CondFunctor : public std::unary_function<bool, T> {
public:
    virtual bool operator()(const T&)=0;
};

template <class Tcont, const int K, const int L>
class IntegerRangeCondition : public CondFunctor<Tcont> {
private:
    class NegPred : public std::unary_function<bool, int> {
public:
        bool operator()(int i) const
            { return i<K || i>L; } // 'i' not in [K,L].
    };
public:
    bool operator()(const Tcont& c) {
        return std::find_if(
            c.begin(), c.end(), NegPred()
        ) == c.end();
    }
};

template <class T, const unsigned N>
class StackREPSizeGroup : public AppInvariant< StackREP<T> > {
private:
    class Invariant : public CondFunctor< StackREP<T> > {
public:
        bool operator()(const StackREP<T>& stack)
            { return stack.Total() <= N; }
    };
    ...
};

```

Figure 9. Employing functor classes, in contrast to normal functions, for condition expressions of application invariants.

Additionally, static functions require syntactic visibility to any external referenced variables at the point of their definition, a fact that which may introduce unnecessary structural dependencies in the source code. A better way is to implement condition expressions as functor classes, i.e. classes overloading the function call operator (). This extension is illustrated in Figure 9, providing the `CondFunctor` super-class for condition expressions and an example demonstrating the capability to program more sophisticated generic conditions. More specifically, the `IntegerRangeCondition` is a template condition functor for range checking of integer values in container classes supporting iteration through Standard Template Library (STL) iterators. This is accomplished by reusing the `find_if` algorithm of STL, passing a predicate functor instance of the class `NegPred`, actually encapsulated inside the condition functor class `IntegerRangeCondition` (see Figure 9).

The negation predicate implements the opposite test, i.e. checks if the argument is outside the specified range, so that a failure of `find_if` to detect an element satisfying this predicate effectively implies that all elements are within the legal range. At the bottom of Figure 9, the re-implementation of the *group* invariant template class `StackREPSizeGroup` for upper-bound size check using a functor class is provided.

Using application invariants

The deployment of application invariants enables the detection of numerous software defects, ranging from malicious side effects or memory corruption to logical data manipulation errors (a more detailed discussion for types of detectable defects follows later). From the programming point of view, clients are only required to implement the condition functors and then exploit appropriately the dynamic registration capability of application invariants; i.e. add/remove condition functors to supplier instances according to the particular needs for runtime supplier instance verification. Apart from the previously discussed examples, we will briefly outline a scenario from real practice in the context of network service development: multiple network connections can be opened with remote devices, as implementations of derived supplier classes inheriting from the `NetLink` base class; derived classes such as `SocketLink` (i.e. TCP/IP) or `L2CAPLink` (Bluetooth™) are typically offered. Additionally, service-specific supplier classes derived from the `Service` base class are delivered, the latter accepting a `NetLink` instance as a constructor argument. Finally, there are two basic singleton collector classes, the `NetLinkCollector`, which is a list of all `NetLink` instances, and `ServiceCollector`, which is a list of all `Service` instances. In this case, in our specific client application, the deployment-specific design logic encompassed the following regulations:

- all derived instances of `NetLink` are constructed dynamically using the placement syntax of the C++ `new` operator (allows instance creation at programmer supplied memory addresses), taking memory from a static program buffer;
- in the `NetLinkCollector` class there are no replicate occurrences of `NetLink` instances;
- a `NetLink` is always used by some `Service` instance.

Such application-specific regulations were directly mapped to condition functors, dynamically installed in the global application invariant of the `NetLink` super-class. The implementation of some of those regulations is outlined in Figure 10, representing collector lists as STL `std::list` classes for clarity (in the actual implementation, collectors were made as STL `std::list` derivatives with application invariants support). As shown in Figure 10, for each supplier class the correctness condition functors of the application invariant are collected in an appropriate C++ namespace. Additionally, the `NetLink` class is made to support application invariants as it inherits from the `AppInvariants<NetLink>` class.

In this particular software application, we have decided to initiate exhaustive tests across all `NetLink` instances every time a member function of any particular `NetLink` instance is called. For instance, the `AddressCond` condition functor, although called for the particular hosting `NetLink` instance, performs a global address verification of all `NetLink` instances of the `NetLinkCollector` singleton class. This type of flexibility in testing deployment correctness is representative of the power of application invariants, as they allow developers to dynamically handle the granularity of runtime client-specific supplier-state verifications. For example, developers may

```

namespace AppInvariants_NetLink {

class ServCond : public CondFunctor<NetLink> {
private:
    const ServiceCollector& C;
    class Pred :
        public std::binary_function<bool, Service*, NetLink*> {
    public:
        // Returns if 'l' the network link of service 's'.
        bool operator()(Service* s, NetLink* l) const
            { return s->GetNetLink() == l; }
    };
public:
    bool operator()(const NetLink& p) {
        return std::find_if(
            C.begin(), C.end(), bind2nd(Pred()), &p
        ) != C.end();
    }
    ServCond (const ServiceCollector& c): C(c){}
};

class AddressCond : public CondFunctor<NetLink> {
private:
    unsigned L, R; // Memory address boundaries.
    const NetLinkCollector& C;
    class NegPred : public std::unary_function<bool, NetLink*> {
    public:
        bool operator()(NetLink* p) const
            { return L > ((unsigned) p) &&
                R < ((unsigned) p) + p->Size(); }
    };
public:
    bool operator()(const NetLink& p) { // All links are checked!
        return std::find_if(
            C.begin(), C.end(), NegPred()
        ) == C.end();
    }
    AddressCond (unsigned l, unsigned r, const NetLinkCollector& c):
        L(l), R(r), C(c){}
};
} // AppInvariants_NetLink

NetLink::GlobalAppInvariant().Add(
    new AppInvariants_NetLink::ServCond(
        ServiceCollector::GetSingleton()
    )
);
NetLink::GlobalAppInvariant().Add(
    new AppInvariants_NetLink::AddressCond(
        Mem::Start(), Mem::End()
    )
);
};

```

Figure 10. Extended example for deployment of application invariants; notice that the correctness conditions for the NetLink class are all collected in a global AppInvariants_NetLink namespace.

install more intensive application invariant tests during execution, prior to calling potentially offensive code, thus enabling more detailed monitoring for state-correctness violations.

Extending the scope of preconditions

Although application invariants are not concerned with preconditions, by being a genuine defensive programming method they are accompanied with a specific proposition to revisit the traditional treatment of preconditions, in order to accomplish better bug defense. Following the original definition of preconditions in Design by Contract, they actually enable clients to identify when calls to member functions are semantically eligible. Practically, preconditions define instance *readiness* for posting member function calls, while clients are requested to always test instance readiness prior to calling member functions. However, preconditions do not filter eligibility of call parameters, i.e. *appropriateness* of a call, while there are no regulations in Design by Contract regarding the way ill-formed function calls are to be explicitly checked and discarded by clients.

As a result, it is implicitly considered that this checking logic becomes an integral part of the function implementation semantics. However, this assumption formally implies that all incorrect parameter values actually become part of the function source domain, leading to an apparent design flaw. Consequently, as with readiness preconditions, appropriateness testing is similarly defined as an additional required category of function preconditions, asserted after readiness testing and prior to the respective function calls. Appropriateness preconditions have identical signatures with their associated member functions, accepting the exact actual arguments prior to the call. Additionally, by being implemented as functors, they can store locally the evaluated actual arguments, supplying those automatically to the target function calls, thus avoiding duplicate argument evaluations. This software pattern is illustrated in Figure 11; once a well-formed call is made, the argument list functor is marked as consumed (i.e. `f.args.used = true` statement). Such appropriateness-oriented filtering of actual arguments in member-function calls is allowed to change during execution, varying with time either for a single instance or for distinct instances of a supplier class.

For instance, *Stack* instances may dynamically allow or forbid replicate elements, or elements within client-specified ranges, reflecting dynamic argument eligibility logic. In this case, the explicit introduction of call-eligibility filtering enables developers to modularly host correctness logic that cannot be accommodated in a documented and elegant manner through Design by Contract.

As a more elaborate example, consider the case of a simple *ClientSocket* supplier class (see Figure 12). This class offers a basic `Connect` function, accepting a single argument named `address` of `string` type. Following the definition of call readiness, the `Connect` function of a *ClientSocket* instance may be well called as far as the invariant is satisfied. However, in such a call, there is no guarantee that the `address` argument supplied actually corresponds to a valid IP address string (also called dotted address); i.e. formally speaking, that the `address` argument belongs to the source domain of the `Connect` function. Such an argument mismatch event is semantically different from the typical failure to establish a connection, as the latter is an operational (functional) error, while the former is actually a case of an ill-formed method call. Clearly, ill-formed calls are not merely a matter of compile-time checks, neither a postponed obligation to be checked by supplier classes, since it clearly denotes erroneous information delivered from clients to suppliers. Hence, such typical call eligibility tests need to be explicitly formalized as a client contractual obligation per se, to allow better runtime defect detection. Theoretically, call-eligibility preconditions document and guard explicitly

```

class Supplier {
private:
    class f_Args {
        // Supplier needs access to actual arguments.
        friend class Supplier;
    private:
        T1 x1; ... Tn xn; // Argument value copies.
        Supplier& x; // Supplier reference.
        bool used;
    public:
        bool operator() const {
            return !used &&
                (x1,...,xn) is a well formed x.f call;
        }
        f_Args(T1 y1,...,Tn yn, Supplier& _x) : x(_x), used(false)
            { for each j in(0, n]: xj = yj; }
    };

    // Argument class needs access to Supplier.
    friend class f_Args;
    f_Args f_args;

public:
    bool PREREADY_f (void); // readiness precondition
    bool PREAPPR_f (T1 y1,...,Tn yn){ // appropriateness precondition
        new (&f_args) f_Args(y1,...,yn, *this);
        return f_args();
    }
    void f (void) { // Originally it was f(x1,...,xn)
        assert(f_args());
        f_args.used = true; // Arguments were consumed
        f logic here for arguments: f_args.x1, ... ,f_args.xn
    }
};

Supplier x;
if (x.PREREADY_f() && x.PREAPPR_f(y1,...,yn))
    x.f();

```

Figure 11. The software pattern for changing supplier classes to accommodate appropriateness preconditions; duplicate evaluation of actual arguments is eliminated through temporary storage in appropriateness-evaluation functors.

```

class ClientSocket {
public:
    bool Connect(string address);
    ...
};

ClientSocket s;
s.Connect(
    ``ill formed address, but no contractual check for that!``
);

```

Figure 12. An example showing ill-formed calls that can only be detected with explicit client contractual obligation for call eligibility tests.

partial procedures, i.e. those not defined for all types of input values, clearly exposing errors on the supplement of procedure arguments.

DISCUSSION

Experience in defect detection through application invariants

Naturally, the scenarios for defect detection through Design by Contract in general, and via application invariants in particular, cannot be limited to any actual taxonomy or classification scheme. However, in the course of real practice, we have identified particular cases where, while Design by Contract turned out to be more *entropy*-oriented [16] in terms of the required source code update patterns, application invariants were modularly injected to detect largely hidden defects like the following.

- *Memory corruption.* While one of the most common defects, it can be caused due to many reasons, like miscoding, logical errors, unhandled exceptions and erroneous use of pointers. It may occur in the context of data values such that the Design by Contract invariant of supplier instances remains legally satisfied. For example, the corruption in the content of a text buffer is observable only by using client-specific knowledge (i.e. content structure), or in some cases it may only be identified by the end-user. In this case, client conditions for more detailed filtering of correct states help quickly trap corruption scenarios; e.g. "myva%\$%^????is\|" is an acceptable string content value for a generic symbol table in Design by Contract, but clearly not a valid program identifier for the lexical analyzer of a particular language-specific compiler. Unless the symbol table is extended to encapsulate client filters, the traditional class invariant will never detect this particular error.
- *Logical errors.* Since this is a very general term, we contextually define as logical errors all situations where *programmed artifacts deviate from the program design intention*, i.e. there is a design–implementation mismatch. Such logical errors may be caused ‘intentionally’, i.e. the programmer does not understand the design and/or the program behavior, or accidentally,

i.e. mistyping that does not result in a syntax error. Logical errors may cause memory corruption, most frequently leading to unwanted program behavior, that can be externally observable (the good scenario) or not (the worst case scenario, but unfortunately very common). In this case, the encapsulation of design-conformance assertions, i.e. programs test their implementation correctness, is arguably the best diagnostic and therapeutic programming method. With application invariants, the client-program design logic, concerning deployment of supplier classes, is modularly encapsulated and intensively tested, enabling the dynamic configuration of the testing granularity, effectively supporting quick defect detection. The more design logic is embedded in the form of asserted condition expressions, the better the client self-checking for design conformance.

- *Unexpected overwriting.* This is a particularly tough memory-corruption side effect, which results in client-acceptable values, either because clients cannot define more strict acceptance criteria, or because the specific nature of the defect leads to overwrites with seemingly legal values. It is very hard to trace unexpected overwriting merely with unit testing, since it results in acceptable results, while in certain cases it tends to appear as memory corruption defects in production/release mode, while remaining silent in debugging mode. The reason for such dual external behavior is primarily the extra debugging information padded in debug mode, resulting in different memory requirements between debug mode and production mode. Application invariants have been employed in two ways to attack unexpected overwriting.
 - (a) Further restricting the range of acceptable values through customized unit tests, effectively implementing filtering conditions which rejected values outside those appearing in unit tests. For instance, in a compiler implementation case, the number of function arguments stored as a byte was actually overwritten by an array of enumerated values in the range [0,9). After observing the value overwriting pattern, but having no clue of the real cause, test programs in which all functions had more than 10 arguments were defined, while a global test unit specific application invariant was also easily attached to the function class, asserting that the number of formal arguments is greater than 10. Apparently, it is not this particular diagnosis trick that is important, but the modular way it can be easily accommodated by application invariants.
 - (b) Keeping synchronized supplier instance replicas, asserting perfect matching in every supplier member call. This approach relies on the significantly small probability of an instance replica to be corrupted in exactly the same manner as its original image. In this case, a *local* application invariant (i.e. attached to every instance) was implemented, one per supplier class, consisting of a single condition functor encompassing the replicated supplier instance as a local variable. Each such condition functor asserted the matching of the functor-calling instance with the functor-local instance. One important detail is that the replicas were only memory images of the supplier instance, i.e. a byte array, not truly functioning instances, by performing bit-level matching. The production of memory-identical functional copies may not always be enabled by the copy semantics of supplier classes, while depending on the semantic role of supplier instances, such copies may change the overall program behavior. Finally, in some cases only value copies of the observably offended members were kept, by performing member-specific equality tests.

Supplier asserted preconditions

Following the Design by Contract, clients are responsible for checking preconditions prior to calling member functions of supplier instances, while supplier classes are responsible for asserting the satisfaction of the corresponding preconditions after the member functions complete. In this context, the original method does not guarantee a safe program state if clients dismiss precondition checking at some point. However, a public function may assume the precondition to hold, without having to explicitly check it. This type of contractual assumption introduces an explicit type of ‘interaction’ with the client program that is called the *generous specification*, since it always presumes that the client code is correct, in the sense it respects method preconditions [14,15]. However, generous specifications cannot certify that trusted clients respect the contract.

In a best-case scenario of a failed precondition under generous specifications, the function postcondition or the instance invariant will fail directly inside a statement of the called function, thus instantaneously signaling the programming error. However, such a hypothesis cannot be proved for all types of supplier classes, since it is possible that the state of a particular supplier instance is not always corrupted after an illegally called member function. For instance, consider supplier instances offering coordination commands to other managed software components (i.e. actually manipulating the state of other components), or the more common case of observer member functions that do not alter the state of the caller instance (like the `Top` function of the *Stack* family of classes). Clearly, such observer functions return erroneous data, likely to unpredictably poison the client state.

To avoid such scenarios, *defensive specifications* are defined [14], effectively denying method invocation in the case of a false precondition. Such specifications ensure that supplier methods will not be called, so error propagation will be forbidden. In this case, we propose stronger defensive specifications for methods, effectively causing either an immediate failed assertion, or an exception for precondition violation if the precondition does not hold.

The main reason is that, in such situations, the defect may travel far enough within program components, causing unexpected and non-directly detectable side effects, especially if some program units do not embed sophisticated bug defense. In this context, it is argued that *it is unsafe to assume that documented programming contracts imply trustworthy clients*. The proposed alternative is to program supplier classes to have explicit embedded precondition checking, following the modified software pattern illustrated in Figure 13.

Normally, failed preconditions indicate an exceptional situation in the normal control flow, which should be handled by programmers through an alternative execution path, like the `else` statement on the left side of Figure 13. However, semantically this source fragment does not signify the particularly exceptional situation, since, if the programmer dismisses either the precondition checking or the `else` part, the execution proceeds without interruption, meaning the defect is silently propagated. However, through the software pattern on the right side of Figure 13, client programmers are explicitly concerned with such documented exceptions, while precondition checking is automatically performed internally by supplier instances. Consequently, when programmers forget to explicitly handle such ‘precondition violation’ exceptions, a runtime error is instantaneously posted, forbidding the defect to travel further through program execution.

The rest of the discussion that follows is motivated by two key possible concerns regarding the proposed technique: (a) why the application of the original Design by Contract over clients is not adequate to effectively and efficiently capture the deployment correctness of supplier classes in the

<i>Client precondition checking</i>	<i>Supplier precondition checking</i>
<pre> X::f() { } if (x.PRE_f()) x.f(); else <i>Logic for failed precondition</i> </pre>	<pre> X::f() { if (!PRE_f()) throw Precondition violation } try { f(); } catch (Precondition violation) { <i>Logic for failed precondition</i> } </pre>

Figure 13. The transformation of the original pattern with client-side precondition checking (left) to supplier-embedded precondition checking (right); the second pattern ensures that the function is never performed with an unsatisfied precondition.

context of a client implementation; and (b) the capability to port the suggested implementation pattern easily and intuitively in other languages (e.g. Java, C#, Action Script, Python, etc.), apart from C++, which is adopted for the presentation of the implementation details.

The practical need for application invariants

Application clients are not typically structured in the form of classes used by third parties, i.e. it is uncommon to wrap them up as supplier classes. However, let's assume that we transform a client application into a supplier class, and then try to implement the logic of application invariants through traditional Design by Contract. In this case we should collect all correctness expressions for every supplier instance into a single invariant. This will typically require multiple registration lists, one per differentiated supplier-state correctness logic, effectively necessitating the bookkeeping of every supplier instance. Those lists would need to be maintained centrally during runtime, as supplier instances are created or destroyed. However, the capability to centrally register callbacks on supplier instance construction/destruction may not be supported by every supplier class, neither may it be a built in property of object classes in the language. But even if it is, the distinction among supplier-instance roles requires that programmers inject within different locations of the client source code explicit registration/removal statements, just after/before the statements creating/destroying supplier instances.

Once the necessary implementation infrastructure of the client invariant is completed, the key question becomes *where should the invariant check call be incorporated?* The only rational way to resolve this issue is to put invariant checks inside every client function. However, application functions are not so intensively called within application clients. Typically, client functions may cause such a considerable number of supplier-function calls that the relative frequency of client-invariant checks becomes unacceptable for direct defect detection. Consequently, to overcome this issue, more intensive checks are heuristically distributed across the source code. Overall, the practicing of the previous recipe reveals severe software engineering problems, such as:

- low re-usability, as each client effectively rebuilds the necessarily customized infrastructure;
- tight coupling, as refinements on which supplier classes are employed, or updates on the correctness conditions for supplier instances, typically introduce updates of the infrastructure;
- performance barriers, as the invariant always checks every supplier instance in evaluations, which may have to be altered for practical reasons.

From my experience in applying the above recipe in the course of real-life projects, it came out that such monolithic gathering of large-scale client-deployment correctness logic tends to become a serious software engineering bottleneck. The design and implementation of application invariants as a new contractual feature, and its transformation to an implementation pattern with distributed management at supplier-class side, allowed far better handling of client-oriented supplier-instance correctness, while it facilitated the direct re-use of the mechanism across different projects (supplier classes were simply extended to inherit from the application invariants class). Finally, application invariants allowed us to have less dependencies among programmers' tasks, since, with the traditional recipe, it was necessary to wait and synchronize the 'central invariant logic' every time new supplier-instance roles, i.e. correctness regulations, appeared during the design and implementation processes.

Portability of the implementation pattern

The suggested implementation pattern relies on a typical callback mechanism, since dynamically registered condition expressions actually constitute client-supplied functions to be called later by supplier instances. Such mechanisms are directly implementable in known object-oriented languages like Java or C# as follows: instead of using function registration (like in C++), instance registration is employed, for instances of a class having a standard evaluation function, e.g. *eval* returning a Boolean value. Then, the callback action is effectively equivalent to calling the *eval* member function.

The feasibility of the above scenario is apparent when the client and supplier class are both in the same process space, also implemented in the same language. The callback linkage is less evident when supplier classes are wrapped up around a component-ware infrastructure. In this case, the application invariants API should be delivered as any other API of a supplier class, enabling asynchronous notifications and client-side callbacks; clearly, the latter is a matter of putting the necessary effort into the supplier-class side to normally export the application invariants API over the component-ware infrastructure, as any other callback API. In most known component-ware frameworks, like DCOM or CORBA, the registration and call of client-side callback functions is explicitly handled at client-side space, with an appropriate extension of the client component stub. During runtime, call notifications that arrive from the component instance are effectively dispatched within client space, inside the implementation of the component stub.

SUMMARY AND CONCLUSIONS

Design by Contract is a well-known method for the design of object-oriented software, which introduced contractual obligations between supplier classes and client classes. Historically, it has been the first reported method to emphasize implementation correctness as a quality that can be directly tested at runtime by software programs, as far as they systematically embed design specifications in

the form of computable expressions. In Design by Contract, global rules of implementation correctness for supplier instances, as defined in the context of supplier class development, are basically supported. However, the case-specific correctness logic for supplier instances, reflecting application restrictions and regulations defined by client classes for supplier instances, is not explicitly accommodated in Design by Contract.

In this framework, *application invariants* reveal the fundamental bilateral runtime cooperation between client classes and supplier classes, to convey and assert application-oriented supplier-instance correctness. Application invariants host correctness logic, which, although defined by client classes, primarily engages supplier instances. Effectively, without application invariants, the manual embedding and checking of correctness logic within client classes is a software pattern introducing a large number of replicated assert calls fused with client source code, additionally requiring increased source code complexity to accomplish the dynamic granularity of the correctness testing units. In this framework, application invariants aim to address this software engineering issue, offering a well-defined placeholder to dynamically host such correctness logic within supplier instances, in a modular and reusable manner, while requiring minimal and constant injections within the client source code.

Additionally, an enhanced view of preconditions is introduced, differentiating between the two fundamental prerequisites for eligible functions calls, namely *readiness*, i.e. testing whether an instance is ready to accept a call request for a particular member function, and *appropriateness*, i.e. testing whether a particular member call is made with a well-formed list of actual arguments. Clearly, the traditional preconditions in Design by Contract may only assert readiness, since appropriateness checking requires an explicit list of actual arguments. Finally, an inverted deployment method for preconditions is proposed, requiring supplier classes to explicitly assert preconditions internally, and clients to be aware of precondition violation exceptions. This approach emphasizes enhanced bug defence, since supplier deployment errors will cause instant posting of runtime exceptions, in comparison to the traditional style of precondition assertions by clients where missed checks lead to dangerous defect propagation. Clearly, this proposition introduces a stricter contractual obligation on behalf of the client:

Function calls are committed only if their preconditions are satisfied. Otherwise, precondition violation exceptions are raised, which, unless handled, will cause instant program interruption.

While Design by Contract is a genuine defensive programming method, it is not a testing method. Typically, programs written with embedded bug defence always require an appropriate method to perform exhaustive testing of program units. In this context, the two key approaches towards software robustness are revealed: (a) *embedded self-verification* of program units for conformance of their implementation to the corresponding design semantics, i.e. Design by Contract; and (b) *diagnostic functionality tracing* of program units, asserting test requests with expected correct responses, i.e. unit testing.

It is argued that these two approaches have to coexist in a software engineering process, since none of the two can substitute the other. More specifically, although Design by Contract supports embedded verification of implementation conformance to design regulations, it cannot guarantee implementation correctness, neither design correctness. Similarly, while unit testing reflects embedding of systematic tests of program behavior, it is mostly based on statistical evidence of program correctness, meaning it can only asymptotically approach provability of implementation correctness, as the number of

performed tests is closer to the totally possible test cases (the latter may well be infinite). Although there has been some criticism regarding the utility of the original Design by Contract method in software safety and robustness [17], it is believed that when Design by Contract is effectively married with sophisticated testing methods, it will lead to higher software quality and safety. Additionally, suggested improvements effectively contribute towards the evolution of the method, as is the case with application invariants, precondition categories, and supplier-asserted preconditions, introduced in the context of this paper, so as to better support defensive programming.

REFERENCES

1. Meyer B. *Object-Oriented Software Construction* (2nd edn). Prentice-Hall: Santa Barbara, CA, 1997.
2. Beck K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley: Reading, MA, 1999.
3. Beck K, Gamma E. JUnit test infected: Programmers love writing tests, 2001. <http://junit.sourceforge.net/doc/testinfected/testing.htm>.
4. Hunt A, Thomas D. Pragmatic unit testing—in Java with JUnit. The Pragmatic Programmers, LLC, 2003. <http://www.pragmaticprogrammer.com/>
5. Savidis A. The implementation of generic smart pointers for advanced defensive programming. *Software—Practice and Experience* 2004; **34**:977–1009.
6. Cress D, Roberts B, Simmons J. A strategy to integrate defensive programming into the undergraduate computer science curriculum at UMBC, 2003. <http://www.cs.umbc.edu/~cress1/ia/Deliverables/Final-Report.PDF>.
7. Hovemeyer D, Pugh W. Finding bugs is easy. *Proceedings of 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, British Columbia, 24–28 October 2004. ACM Press, 2004.
8. Viega J, Bloch T, Kohno Y, McGraw G. ITS4: A static vulnerability scanner for C and C++ code. *Proceedings of the 16th Annual Computer Security Applications Conference—ACSAC 2000*, New Orleans, LA, 11–15 December 2000. IEEE Press, 2000.
9. Meyer B. *Eiffel: The Language*. Prentice-Hall: Santa Barbara, CA, 1992.
10. Burdy L, Cheon Y, Cok D, Ernst M, Kiniry J, Leavens G, Rustan K, Leino M, Poll E. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer* 2005; **7**(3). Available at: <ftp://ftp.cs.iastate.edu/pub/leavens/JML/sttt04.pdf>.
11. Barnett M, Leino K, Schulte W. The Spec# programming system: An overview. *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS 2004)*. Springer, 2004. Available at: <http://research.microsoft.com/SpecSharp/papers/krml136.pdf>.
12. Aechmea, C², 2005. http://www.aechmea.de/html/german/Information01_e.htm.
13. Ottosen T. Proposal to add Design by Contract to C++, 2004. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1613.pdf>.
14. Abrial J. *The B Book*. Cambridge University Press: Cambridge, 1996.
15. Milanovic M, Malek M. Extracting functional and non-functional contracts from Java Classes and Enterprise Java Beans. *Proceedings of the Workshop on Architecting Dependable Systems (WADS 2004), International Conference on Dependable Systems and Networks (DSN 2004)*, Florence, Italy, 2004.
16. Hunt A, Thomas D. Software entropy. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley: Reading, MA, 1999. Available at: http://www.pragmaticprogrammer.com/ppbook/extracts/no_broken_windows.html.
17. Garlington K. Critique of ‘Put it in the contract: The lessons of Ariane’, 1998. <http://home.flash.net/~kennieg/ariane.html>.