A CRITICAL COMPARISON OF EXISTING SOFTWARE CONTRACT TOOLS

Janina Voigt, Warwick Irwin and Neville Churcher
Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand

Keywords: Software contracts, Design by contract, Formal software specification.

Abstract: The idea of using contracts to specify interfaces and interactions between software components was proposed several decades ago. Since then, a number of tools providing support for software contracts have been developed. In this paper, we explore eleven such technologies to investigate their approach to various aspects of software contracts. We present the similarities as well as the areas of significant disagreement and highlight the shortcomings of existing technologies. We conclude that the large variety of approaches to even some basic concepts of software contracts indicate a lack of maturity in the field and the need for more research.

1 INTRODUCTION

When writing software, we aim to create programs which not only work correctly, but are also reliable, easy to use, understand and maintain. These and other factors combine to determine the level of quality in software.

Developing high quality software is a difficult, complex and time-consuming task. The sheer size and complexity of software contribute to these difficulties; it is not unusual for a single program to contain millions of lines of code, far too much for one person to understand. To manage this size and complexity, we break large systems into smaller components which can be developed independently. A developer working on one component does not need to know the internal details of other components of the system; he or she only needs to understand the other components' interfaces in order to use their services.

Software contracts (a subfield of formal specifications) are used to explicitly define the interfaces of software components, specifying the responsibilities of both the client using a service and the supplier of the service. This formalises the interactions between components of the software and ensures that two components interact correctly (Meyer, 1997).

When software contracts are not used, clients of a service usually have access to information about the service's interface, including method signatures, as well as, optionally, documentation about how to use the service. Software contracts elaborate on this by formally specifying protocols of interaction which otherwise may have remained implicit. Consequently, we regard contracts as a natural extension of explicit type systems; they specify interfaces fully rather than just specifying signatures.

We believe that software contracts can mitigate some of the problems surrounding large scale software development. They not only improve the correctness of software by explicitly specifying interaction protocols, but also serve as documentation and clarify correct use of inheritance (Meyer, 1997).

Further, formal specifications such as software contracts "represent a significant opportunity for testing because they precisely describe what functions the software is supposed to provide in a form that can easily be manipulated" (Offutt et al., 1999, page 119). In particular, software contracts describe valid inputs and outputs to methods; this information can be used by automatic testing tools to find valid test inputs and decide if particular test outputs are correct.

Despite the fact that the main ideas of software contracts were proposed several decades ago, they are still not commonly used in mainstream software development. Meyer remarks that

> In relations between people and companies, a contract is a written document that serves to clarify the terms of a relationship. It is really surprising that in software, where precision is so important and ambiguity so risky, this idea has taken so long to impose itself. (Meyer, 1997, page 342)

However, more recently several different tech-

Voigt J., Irwin W. and Churcher N..
A CRITICAL COMPARISON OF EXISTING SOFTWARE CONTRACT TOOLS.
DOI: 10.5220/0003421200380047
In *Proceedings of the 6th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2011), pages 38-47
ISBN: 978-989-8425-57-7

nologies supporting software contracts have been developed, including tools for mainstream programming platforms such as Java and .NET. Along with these technologies, a number of supporting tools are emerging. Testing tools such as AutoTest for Eiffel (Meyer et al., 2007) and Pex for .NET (Barnett et al., 2009; Tillmann and Halleux, 2008) automatically extract unit tests from contracts without the need for input from developers. Static analysers such as Boogie for the .NET contract language Spec# (Barnett et al., 2006) and ESC/Java for the Java contract language JML (Flanagan et al., 2002) attempt to prove the correctness of software at compile-time.

As more technologies supporting software contracts emerge and their usage becomes more common, it is important for us to take stock of current developments and uncover any issues and areas of disagreement which need to be addressed in the future. This is what we attempt to do in this paper, as part of a wider project in which we seek both to strengthen the theoretical underpinnings of contracts and to develop tools to support the adoption of contracts in modern software engineering environments.

The rest of this paper is structured as follows: Section 2 explains the background of software contracts. Section 3 presents a comparison of several contract technologies, highlighting the similarities and differences. A discussion of the issues and criticisms of existing approaches follows in Section 4 before we present our conclusions in Section 5.

2 BACKGROUND

The roots of software contracts run very deep in the field of computer science; although it has been little recognised in the literature, the origins of the idea can be traced as far back as Turing, who first presented the idea of assertions to check program correctness in 1949 (Turing, 1949).

In 1969, Hoare introduced *Hoare triples*. He used the notation $P\{Q\}R$ to mean that "If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion" (Hoare, 1969, p. 577); P is commonly called the *precondition*, while R is the *postcondition*. Three years later, Hoare also presented the concept of the *class invariant*, a logical predicate *I* where "each operation (except initialisation) may assume *I* is true when it is first entered; and each operation must in return ensure that it is true on completion" (Hoare, 1972, p. 275).

In the late 1980s, Meyer applied Hoare's work in his development of *Design by Contract*^{TM} and the programming language EIFFEL which included the concepts of preconditions, postconditions and class invariants (Meyer, 1989). Preconditions specify what the client must ensure before calling the service provider; this could for example include ensuring that the parameters are not null. Postconditions define what the service provider promises in return, given that the client has fulfilled the preconditions.

As an example, we define the contract for a simple `Stack` class with the three standard methods `push(Object obj)`, `peek()` and `pop()`:

```
class Stack {

  private Object[] stack;
  private static final int MAX_SIZE = 100;
  private int size;

  Invariant: size >= 0 && size <= MAX_SIZE;

  public Stack() {
    stack = new Object[MAX_SIZE];
    size = 0;
  }

  Precondition: !isFull()
  Postcondition: peek() == obj
      && size == old size + 1
  public void push(Object obj){
    stack[size++] = obj;
  }

  Precondition: !isEmpty()
  Postcondition: size == old size
  public Object peek(){
    return stack[size-1];
  }

  Precondition: !isEmpty()
  Postcondition: size == old size - 1
  public Object pop() {
    return stack[--size];
  }

  public boolean isFull(){
   return size >= MAX_SIZE;
  }

  public boolean isEmpty(){
    return size <= 0;
  }
}
```

Our `Stack` class uses a simple `Object` array to store its values. It keeps track of the current `Stack` size and also knows the maximum number of items it can store.

We have defined preconditions and postconditions for `push`, `pop` and `peek`. The preconditions for `pop` and `peek` ensure that the methods are not called when the `Stack` is empty; the precondition for `push` makes sure the method is not called if the `Stack` is already

full. These preconditions call the query methods `isEmpty` and `isFull` in their definitions instead of referring directly to the private `size` field. Since preconditions are the client's responsibility, they must be defined in such a way that the client can check them before calling a method; that is, their definition should only include members which are accessible to the client (Meyer, 1989; Fähndrich et al., 2010). The `isEmpty` and `isFull` methods therefore need to be public.

The postconditions of the three methods check that the size of the `Stack` has changed in the correct way by comparing it to the `old` size of the `Stack`; that is, the size before the method's execution. Using old values is a common occurrence in contracts and therefore contract specification languages usually provide syntax for doing so. Calling the `push` method increases the size of the `Stack` by one; calling `pop` decreases it by one; calling `peek` should have no effect on the size. Unlike the preconditions, the postconditions access the `size` field directly and do not make use of query methods. This does not cause any problems here because postconditions are the responsibility of the service supplier; that is, the `Stack` itself. They do not need to be checked by outside clients and can therefore refer to the private details of the `Stack`.

The invariant of the `Stack` ensures that its size never drops below zero or exceeds the array's capacity. This invariant must be satisfied in all observable states of every instance of a class (Meyer, 1989). Specifically, the class invariant must be true after the constructor has finished constructing a class instance and before and after each call to an exported method of the class; that is, a method accessible from outside the class. This implies that while methods of the class are executing, they may violate the class invariant, as long as it is again satisfied when the method returns (Meyer, 1997).

Software contracts also apply in the presence of inheritance, through the concept of *subcontracting* (Meyer, 1997); that is, the original contractor engages a subcontractor for part of or all of the work. For this to work, the subcontractor "must be willing to do the job originally requested, or better than requested, but not less" (Meyer, 1997, page 576).

Inheritance allows substitution of a subtype in place of an expected type. This means, for example, that a method expecting an object of type *A* may be given an object of type *B* as long a *B* inherits from *A*. Whenever a client makes use of a supplier, it does not need to know whether the supplier is an immediate instance of the specified type or an instance of some subtype. Therefore, for contracting to continue to work, the subclass must adhere to the contract specified by the superclass (Meyer, 1989). This means that

- Preconditions must be the same or weaker than in the superclass. The subclass cannot expect more of the client, although it may expect less;
- Postconditions must be the same or stronger than in the superclass. The client expects certain results which must be delivered by the subclass. In addition, the subclass may choose to deliver more than promised by the superclass; and
- Class invariants are inherited from the superclass. The subclass may introduce additional class invariants (Meyer, 1997).

In the next section, we present several different contract technologies and contrast their approaches to the implementation and interpretation of software contracts.

3 CONTRACT TECHNOLOGIES

We investigated a number of technologies and programming languages which allow the addition of software contracts to programs, with a particular focus on the following eleven:

- Java contract tools, including
  - JAVA MODELING LANGUAGE (JML) (Leavens et al., 2006; Leavens et al., 2005; Leavens and Cheon, 2006);
  - ICONTRACT (Kramer, 1998);
  - CONTRACT JAVA (Findler and Felleisen, 2000);
  - HANDSHAKE (Duncan and Hoelzle, 1998);
  - JASS (Bartetzko et al., 2001);
  - JCONTRACTOR (Karaorman and Abercrombie, 2005; Karaorman et al., 1999); and
  - JMSASSERT (Man Machine Systems, 2009).
- .NET contract languages, including
  - SPEC# (Barnett et al., 2004b; Leino and Monahan, 2008); and
  - CODE CONTRACTS (Fähndrich et al., 2010; Microsoft Corporation, 2010).
- EIFFEL (Meyer, 1989; Meyer, 1992; Meyer, 1997); and
- OBJECT CONSTRAINT LANGUAGE (OCL) (Object Management Group, 2010; Warmer and Kleppe, 2003)

The large number of existing software contract tools made it impractical to consider all of them and we therefore focused our investigation on the main

technologies which add contract support to the popular programming platforms Java and .NET. In addition, we looked at Eiffel, the original software contract language. OCL was included because of its close links to Java technologies such as JML.

All the tools we investigated aim to support software contracts, most of them at the implementation level. OCL is the only technology to work exclusively at the software design level; it allows contracts including preconditions and postconditions to be added to UML diagrams, while all other tools we looked at allow developers to augment source code using contracts.

We have identified significant differences and shortcomings in what they deliver. Table 1 gives an overview of the similarities and differences of the tools. In the following section, we describe the main characteristics of the technologies, and in the subsequent section we summarise the most important themes and highlight areas of inconsistency.

3.1 Core Contract Support

All of the technologies we looked at provide core contract support, allowing the specification of preconditions, postconditions and class invariants, with the exception of CONTRACT JAVA which omits class invariants.

In addition to the basic contract specifications, some technologies offer additional constructs. SPEC#, JML and JASS allow the specification of *frame conditions*. Frame conditions specify which parts of the memory a method is allowed to modify. This ensures that a method does not unexpectedly change the value of variables it should not be allowed to modify (Barnett et al., 2004b; Leino and Monahan, 2008). A variable is deemed to have been modified if it is accessible at the start and the end of a method and its value has been changed. This means that newly created objects and local variables are not included in the restrictions of frame conditions (Leavens et al., 2006).

SPEC#, CODE CONTRACTS and JML further allow the definition of *exceptional postconditions*, which specify conditions that need to be satisfied if the method terminates with an exception.

Of the technologies we considered, JML provided the most extensive contract support. Among other constructs, it also supports history constraints which describe how the value of a field is allowed to change between two publicly visible states. This can for example be used to express that the value of a field may only increase (Leavens et al., 2006). JML further introduces the concept of *model fields* which can be used when the inner data representation of a class needs to be changed but the developer does not want to update all of the contracts to the new data format. The model field of the old data format can be used from within the contracts and a correspondence is defined between the new data format and the model field (Leavens and Cheon, 2006).

3.2 Special Operators and Quantifiers

The different technologies also offer varying amounts of special operators and quantifiers for use in contracts. All allow postconditions to refer to the return value of the method; this functionality is usually provided by the `result` or `return` operator. In addition, all except CONTRACT JAVA and HANDSHAKE also allow postconditions to refer to the value of a variable before method execution, often through the `old` operator. This is important to check that the value of a field is changed correctly by a method, as we did in our `Stack` example above.

Most technologies also offer some quantifiers such as *for all* and *exists*; no such quantifiers are available in EIFFEL, but Meyer argues that they can be easily emulated using conventional programming language constructs (Meyer, 1989). Several tools, including JML, SPEC#, JCONTRACTOR and OCL, have a sophisticated range of additional operators including quantifiers, counting functions and predicate logic operators.

3.3 The Contract Language

Contract specifications for Java and .NET represent additions to an existing programming language. Some tools, including CODE CONTRACTS and JCONTRACTOR, specify contracts in the existing language. EIFFEL and SPEC# are both languages which natively support contracts and thus the language used to specify contracts is part of the wider programming language. The advantage of this approach is that there is no need for a separate compiler and contracts can be processed by standard tools along with the remainder of the program. In CODE CONTRACTS, contracts are specified by calling the static methods of the `Contract` class; for example, preconditions are defined by calling the `Requires` method of the `Contract` class. In JCONTRACTOR, method contracts are specified in *contract methods*, using standard Java. Postcondition methods take the additional parameter `RESULT`, which can be used to refer to the return value of the method. Postconditions also have access to a special object called `OLD`, which contains the state of the current object as it was

Table 1: Overview of Contract Tools.

		JML	iContract	Contract Java	Handshake	Jass	jContractor	JMSAssert	Spec #	Code Contracts	Eiffel	OCL
Contract Support	Pre / Postconditions	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Class Invariant	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
	Frame Conditions	✓				✓			✓			
	Exceptional Postconditions	✓							✓	✓		
Operators	Result	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Old	✓	✓			✓	✓	✓	✓	✓	✓	✓
Contract Language	Original Language						✓		✓	✓	✓	✓
	Modified Language	✓	✓	✓	✓	✓						
	Scripting Language							✓				
Contract Placement	Comment	✓				✓		✓				
	Annotation		✓									
	With Program	✓	✓			✓	✓	✓	✓	✓	✓	
	Separately			✓	✓		✓					✓
Method Purity	Enforced	✓							✓			✓
Precondition	Visible Members Only	✓				✓				✓		
Invariant Check	After Method		✓			✓				✓		
	Before and After	✓		N/A	✓		✓	✓			✓	N/A
	Expose Block								✓			
Invariant Check	All Methods					✓						
	Non-private Methods		✓	N/A	✓			✓	N/A		✓	N/A
	Public Methods Only	✓					✓			✓		
Contract Inheritance	Enforced	✓	✓	✓	✓		✓	✓	✓	✓	✓	N/A
	Precondition Weakening	✓	✓		✓	✓	✓	✓	✓		✓	N/A
Contract Compilation	Preprocessor		✓			✓		✓				
	Custom Compiler	✓		✓								N/A
	Standard Compiler								✓	✓	✓	
	Runtime Linking				✓		✓					

before the method executed (Karaorman et al., 1999).

The remaining tools we considered take a slightly different approach: they take the original programming language as a basis but augment it using additional keywords and operators. This approach is taken by iCONTRACT, JML and others; it requires special tools to translate the contracts into the original programming language.

JMSASSERT takes this approach a step further by using a full scripting language, JMScript, for contract specification. While JMScript is similar to Java, the underlying programming language, it differs sufficiently that developers need to learn the scripting language before being able to write contracts, significantly steepening the learning curve.

3.4 Integration of Contracts into Source Code

There are several ways in which contracts can be incorporated into source code. Some contract technologies, including JML, JASS and JMSASSERT, require contracts to be added in the form of comments, while in iCONTRACT they are defined as annotations. The advantage of these two approaches is that they work when the contract language is not the same as the standard programming language; the contracts are simply ignored by the standard compiler, meaning that no special compiler is needed when working with contracts. Instead, the contracts are inserted into the source code by a preprocessor and the program is then compiled using the standard compiler.

In EIFFEL, SPEC#, CODE CONTRACTS and jCONTRACTOR, contracts are defined as an integral part of the program and are compiled and checked by the standard compiler. This approach works for these technologies because the contracts are expressed in the same language as the rest of the program.

The placement of contracts in the programs also varies between different technologies. In most cases, for example in JML, iCONTRACT and SPEC#, method contracts including preconditions and postconditions are specified as part of the method header. In CODE CONTRACTS, preconditions and postcondi-

tions are placed inside the method body along with the method implementation. These two approaches have the advantage of clearly showing which contracts apply to which methods.

Other technologies enforce a separation between contracts and the code to which they apply. In HANDSHAKE, specifications are placed in separate contract files (Karaorman et al., 1999); in CONTRACT JAVA they are placed in separate interfaces (Findler and Felleisen, 2000). This approach has the advantage of clearly separating contracts from standard code, allowing them to be considered independently of implementation. It further allows the addition of contracts even when source code is not available, for example when working with third party software.

JCONTRACTOR allows both of these approaches: contract methods to define preconditions and postconditions may be placed in the same class as the methods to which they apply; alternatively, they can be defined in a separate contract class named `ClassName_CONTRACT`, which must extend the class to which it is adding contracts in order to inherit relevant behaviour and to make the objects with contracts substitutable for objects without contracts (Karaorman et al., 1999).

3.5 Side Effects in Contracts

Preconditions, postconditions and invariants should not call methods which cause side effects since this can create bugs which are difficult to trace. Some technologies, including SPEC# and JML, enforce this and allow only methods which have been declared free of side effects (pure methods) to be called from within contracts. Pure methods may only call other pure methods and may not modify any part of the memory. For example, the two query methods we used to define our `Stack` contract, `isEmpty` and `isFull`, have no side effects and can therefore safely be called from within a contract.

Most of the technologies do not explicitly enforce method purity; they only recommended that no methods with side effects are called from within contracts. CODE CONTRACTS is expected to enforce purity in the future (Microsoft Corporation, 2010). OCL is a modeling language and all its code is implicitly free of side effects and thus any methods called from the contract are guaranteed to have no side effects.

3.6 Precondition Visibility

Contract theory requires clients to ensure that preconditions hold; therefore, it is important to ensure that preconditions do not refer to any data or methods which are not visible to clients. Some contract technologies enforce this restriction, while others do not.

CODE CONTRACTS ensures that anything used to define the precondition is visible to clients. JASS and JML require anything referred to by the precondition to be at least as visible as the method itself. Thus, the preconditions of `public` methods must be defined using only publicly visible data and methods; preconditions for `protected` methods may refer to both `public` and `protected` items.

3.7 Checking of Class Invariants

Class invariants are constraints that need to be maintained in all visible states of the objects of a class; that is they must be true at the start and the end of each method that can be called by a client. For this reason, Meyer asserts that each invariant essentially represents an additional precondition and postcondition for each exported method in a class (Meyer, 1989). EIFFEL, JML and JMSASSERT therefore check class invariants at the start and end of each method execution.

However, seeing the invariant as an addition to each method's precondition raises a new problem:

> The object invariant of class T is a condition on the internal representation of T objects, the details of which should be of no concern to a client of T, the party responsible for establishing the precondition. Making clients responsible for establishing the consistency of the internal representation is a breach of good information hiding practices. (Barnett et al., 2004a, page 30)

For this reason, other technologies, including CODE CONTRACTS, ICONTRACT and JASS, check the class invariant only at the end of method executions; that is, only in the postconditions, not the preconditions.

SPEC# takes a more complex approach to invariant checking. It allows changes to memory only inside special `expose` blocks because such changes could invalidate class invariants. At the start of each `expose` block, the object's invariant is set to `false`. Changes to data are now allowed and at the end of the `expose` block the invariant is re-checked. This protects invariants even in the presence of concurrency and reentrancy: an `expose` block can only be entered when the object's invariant is `true`; that is, it can only be entered by one thread of execution at a time (Barnett et al., 2004b). While this approach has the advantage of working in the presence of concurrency, it

greatly increases the complexity of writing programs with contracts.

Apart from the disagreement over when the invariant needs to be checked, there is also some debate about which methods this check applies to. Strictly speaking, the class invariant must be maintained in all externally visible states but may be broken while internal methods are executed. For example, a recursive method needs to maintain the invariant only for its outermost invocation. `Private` methods should be allowed to break the invariant; only methods called by the client should need to maintain it.

Of the technologies we considered, only JASS checks the invariant after each method execution, effectively forcing all methods, including `private` methods, to maintain the invariant. EIFFEL, ICONTRACT, HANDSHAKE and JMSASSERT require all non-private methods to maintain the invariant, while CODE CONTRACTS, JML and JCONTRACTOR only require `public` methods to do so.

Some of the Java technologies allow only `private` methods to break the class invariant, while others allow `private`, `package` and `protected` methods to do so. The latter approach is problematic, since calls to `package` and `protected` methods may come from a different class, and therefore should be forced to maintain the invariant. On the other hand, this allows methods from the subclass to call methods in the superclass while the invariant is broken, which may provide valuable flexibility.

3.8 Inheritance of Contracts

Inheritance is an important mechanism in object oriented (OO) programming and consequently contract tools need to support it. In many technologies, including EIFFEL, ICONTRACT, JML and JCONTRACTOR, correct contract inheritance is enforced by disjuncting inherited preconditions and conjuncting inherited postconditions; this leads to a weakening of preconditions and a strengthening of postconditions and invariants. CODE CONTRACTS and CONTRACT JAVA take a more restrictive approach: while postconditions and invariants may be added by subclasses, preconditions must be specified completely in the superclass; subclasses are not allowed to specify any additional preconditions. This ensures that preconditions are not strengthened, but also makes developers unable to weaken them.

While almost all technologies we investigated always enforce correct use of contract inheritance, JASS takes a more flexible approach. It can check for correct inheritance using refinement checks, but this is optional and can be turned off by the developer. In OCL, the semantics of contract inheritance are not fully specified because it is a general purpose modelling language rather than a concrete implementation.

3.9 Conversion of Contracts into Runtime Checks

Once contracts have been written, they can be turned into runtime checks that report whenever a contract is violated. This conversion may be done in several ways.

Programs written in EIFFEL, CODE CONTRACTS and SPEC# can simply be compiled using a standard language compiler, since contracts are expressed in the same language as the rest of the code. The EIFFEL and SPEC# compilers insert runtime checks for contracts during compilation; CODE CONTRACTS uses library classes to implement contract checking. JML and CONTRACT JAVA provide a customised Java compiler which not only compiles the program but also generates the runtime checks. ICONTRACT, JASS and JMSASSERT all use a preprocessor which inserts Java statements into the code before it is compiled by the standard Java compiler. This has the advantage that the standard Java compiler can be used after preprocessing is completed. HANDSHAKE and JCONTRACTOR use a dynamic library and class loader to inject runtime checks when the program is executed, rather than at compile time.

4 DISCUSSION

In our investigation of existing software contract technologies we have found some areas of significant disagreement. The approaches of the technologies vary widely and from Table 1 it becomes clear that no two tools take exactly the same approach.

Interestingly, we have uncovered some relatively basic issues which are handled inconsistently, for example concerning the checking of class invariants. We believe that it is important that the inconsistencies are resolved - or at least justified - in order to increase developers' confidence in contract tools and the practice of using software contracts in general.

We found good support for core contract concepts, including preconditions, postconditions and invariants, in nearly all tools. We believe that any contract tool which does not support these basic constructs is inadequate for practical use. CONTRACT JAVA, for example, does not support the specification of class invariants, representing a serious gap in this tool.

In addition to preconditions, postconditions and class invariants, we find the concept behind frame conditions useful. It is often difficult to know what data is changed when calling a method, particularly if this method calls other methods. In some cases, unexpected data changes can be difficult to trace to their origins. Defining frame conditions forces developers to think carefully about which parts of the memory a method should be able to access and modify. They inform the programmer of inappropriate memory modifications, reducing the incidence of unexpected data changes.

Some contract technologies provide a wide range of special operators and quantifiers; most tools provide at least two: the `result` or `return` operator to access the return value of a method and the `old` operator to refer to the value of variables before the method execution. However, two tools, CONTRACT JAVA and HANDSHAKE, do not provide an `old` operator. This is a serious omission and severely restricts what contracts can be expressed, such as the size checks in our Stack example.

Most tools we considered here declared contracts using the same programming language as for the rest of the program, although many introduced small additions in the form of operators and quantifiers. Only one tool, JMSASSERT, used a significantly different language to define contracts. We suggest that this is an unnecessary burden on developers and is likely to inhibit uptake of the technology.

With the exception of CODE CONTRACTS, all of the technologies we investigated use contract definition syntax that groups contract information with method declaration information. CODE CONTRACTS places contracts inside the actual methods. We feel that this approach is not ideal, since it mixes contracts with implementation code and makes it difficult to distinguish between them. We suggest that contracts should ideally be declared separately from the implementation as part of a type definition. This is consistent with existing literature, which suggests that public interfaces, or types, should be separated (Bruce, 2002; Canning et al., 1989); that is, the type definition should contain signatures of visible methods, but no internal details. By extension, such a type definition should include contracts for publicly visible methods since, similarly to method signatures, contracts provide vital information to clients wanting to use a service.

Some tools do not allow contracts to call methods with side effects since this can create bugs which are difficult to trace; other technologies do not impose this restriction. We agree with Barnett et al., who claim that the latter approach gives developers too much freedom and is unsound (2004). As we argued above, it can be difficult to see which parts of the memory a method modifies; similarly, it can be difficult to determine whether or not a method is pure, particularly when this method calls other methods, which in turn could have side effects. This makes both frame conditions and explicit declarations of pure methods very useful.

Clients are responsible for ensuring that preconditions are met before calling a method. We are therefore surprised that not more tools ensure that methods and data referred to in preconditions are visible to clients. If this is not the case, clients may not be able to check preconditions and may therefore fail to fulfill their responsibilities under the contract. Contracts are based on the idea of shared responsibility between clients and service providers and having potentially invisible preconditions violates the foundation of software contracts.

In the tools we studied, we found a particularly variable approach to invariant checking. Some tools check invariants after each method, others before and after; some tools require the invariant to hold at the start and end of all methods while others only apply this restriction to `public` methods. In our view, the wide range of approaches stems from the incomplete body of theory about this aspect of contracts. We have found no research that explains when invariants should be checked and what implications the different approaches have. Given the wide range of different approaches, we feel that this is an area where further investigation is warranted.

Most of the technologies allow private methods to break the invariant temporarily. This makes sense because the internal operations of an object may not always maintain the invariant at all times; however, it needs to be restored before returning control to the client to ensure that the object is left in a consistent state. We therefore argue that ideally the invariant should to be checked before and after every method call originating from outside the object. This would allow the object to break its own invariant temporarily (possibly while calling code in the superclass) but would also ensure that the object remains in a consistent state when it returns control.

In the context of invariant checking, SPEC#'s approach is far more complex than that of any other technology we investigated. It requires the object to be explicitly exposed whenever its state is modified to ensure that its invariant cannot be violated by operations from the outside or through the presence of reentrancy and concurrency. Although this approach is sound, we argue that it is too complex; it requires the use of complicated constructs even when writing

simple programs. We believe that this complexity is likely to alienate new users and slow the uptake of SPEC# and software contracts in general.

Support for inheritance of contracts is essential for their use in OO programming. We found that all the tools with the exception of JASS ensure that contracts are inherited correctly. JASS also allows correct contract inheritance to be enforced but makes this optional. We are encouraged by this high level of support for correct inheritance. Using inheritance correctly is notoriously difficult and our intuition sometimes leads us to use it incorrectly. This is particularly evident in the well-known *square-rectangle problem* (Martin, 1996). Our own experience shows that contracts are very valuable when creating inheritance hierarchies because they force us to ensure that an instance of the subclass is substitutable for an instance of the superclass; problems with contract inheritance usually signal incorrect use of inheritance.

Most of the tools we looked at enforce the correct use of contracts by allowing weaker preconditions through disjuncting inherited preconditions and allowing stronger postconditions through conjuncting inherited postconditions. CODE CONTRACTS uses this approach to ensure postconditions are strengthened; however, the tool does not allow the weakening of preconditions because "We just haven't seen any compelling examples where weakening the precondition is useful" (Microsoft Corporation, 2010, page 15). CODE CONTRACTS forces developers to declare all preconditions on the root method of an inheritance chain. In our work with CODE CONTRACTS, we have found this approach very frustrating because it does not allow for flexible precondition definition. In particular, problems arise when a class inherits the same method from multiple interfaces. In this situation, the preconditions of this method in all ancestors must be compatible; this is an example where we feel that allowing precondition weakening is essential.

5 CONCLUSIONS

In our investigation of existing software contract tools we have uncovered a range of differences, clearly demonstrating a level of confusion and conflict surrounding even some basic concepts of software contracts. This indicates to us that more work is needed in this area to resolve these issues and create a consensus or at least a clear taxonomy of the different semantics of software contracts. We have identified a number of shortcomings of existing tools and areas that require more research, including:

- The checking of class invariants;
- The separation of contracts and implementation; and
- The inheritance of contracts, particularly the weakening of preconditions.

We believe that using software contracts has the potential to greatly increase the quality of software and speed up software development. Not only do they ensure that different components of a system know how to interact with each other correctly, but they also serve as documentation of developers' intentions and can be used as a basis of automated testing tools. Furthermore, we believe that they are a highly valuable tool for creating correct inheritance hierarchies.

We are currently developing our own contract tool; having carefully studied other contract tools, we are now aware of the major issues and questions in the area. We plan to create a contract tool that emphasises:

- Rigorous separation of interface (i.e. contracts) from implementation. This will ensure clients can depend only on public information.
- Enhanced explicit support for inheritance and substitutability and enforcement of correct contract inheritance.
- Prevention of invalid contracts, such as preconditions that cannot be tested by clients, or use of methods with side-effects in contracts.
- Support for more flexible and expressive definition of contracts.

REFERENCES

Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. (2006). Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, volume 4111 of *Lecture notes in computer science*. Springer Verlag.

Barnett, M., Deline, R., Fähndrich, M., Leino, K. R. M., and Schulte, W. (2004a). Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56.

Barnett, M., Fähndrich, M., Halleux, P. d., Logozzo, F., and Tillmann, N. (2009). Exploiting the synergy between automated-test-generation and programming-by-contract. In *Proceedings of ICSE 2009, 31th International Conference on Software Engineering, Companion*, pages 401–402.

Barnett, M., Leino, K. R. M., and Schulte, W. (2004b). The Spec # programming system: an overview. In *CASSIS 2004*, volume 3362 of *Lecture notes in computer science*. Springer Verlag.

Barnett, M., Naumann, D., Schulte, W., and Sun, Q. (2004c). 99.44% pure: useful abstractions in specifications. In *ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP) 2004*.

Bartetzko, D., Fischer, C., Möller, M., and Wehrheim, H. (2001). Jass - Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2).

Bruce, K. B. (2002). *Foundations of object-oriented languages: types and semantics*. MIT Press, Cambridge, MA, USA.

Canning, P. S., Cook, W. R., Hill, W. L., and Olthoff, W. G. (1989). Interfaces for strongly-typed object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 457–467, New York, NY, USA. ACM.

Duncan, A. and Hoelzle, U. (1998). Adding contracts to Java with Handshake. Technical Report TRCS98-32, University of California at Santa Barbara, Santa Barbara, CA, USA.

Fähndrich, M., Barnett, M., and Logozzo, F. (2010). Embedded contract languages. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2103–2110, New York, NY, USA. ACM.

Findler, R. and Felleisen, M. (2000). Behavioral interface contracts for Java. Technical Report TR00-366, Rice University.

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. (2002). Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA. ACM.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

Hoare, C. A. R. (1972). Proof of correctness of data representations. *Acta Informatica*, 1(4):271 – 281.

Karaorman, M. and Abercrombie, P. (2005). jContractor: Introducing design-by-contract to Java using reflective bytecode instrumentation. *Formal Methods in System Design*, 27(3):275–312.

Karaorman, M., Hölzle, U., and Bruno, J. L. (1999). jContractor: A reflective Java library to support design by contract. In *Reflection '99: Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*, pages 175–196, London, UK. Springer-Verlag.

Kramer, R. (1998). iContract - the Java(tm) design by contract(tm) tool. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295, Washington, DC, USA. IEEE Computer Society.

Leavens, G., Baker, A., and Ruby, C. (2006). Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38.

Leavens, G. and Cheon, Y. (2006). Design by contract with JML.

Leavens, G., Cheon, Y., Clifton, C., Ruby, C., and Cok, D. (2005). How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208.

Leino, K. R. M. and Monahan, R. (2008). Program verification using the Spec # programming system. http://research.microsoft.com/en-us/projects/specsharp/etaps-specsharp-tutorial.ppt.

Man Machine Systems (2009). Design by contract for Java using JMSAssert. http://www.mmsindia.com/DBCForJava.html.

Martin, R. (1996). The Liskov Substitution Principle. *C++ Report*, 8(3):16 – 17, 20 – 23.

Meyer, B. (1989). Writing correct software. *Dr. Dobb's Journal*, 14(12):48–60.

Meyer, B. (1992). Applying "design by contract". *Computer*, 25(10):40–51.

Meyer, B. (1997). *Object-oriented software construction*. Prentice Hall, 2nd edition edition.

Meyer, B., Ciupa, I., Leitner, A., and Liu, L. L. (2007). Automatic testing of object-oriented software. In *SOFSEM '07: Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science*, pages 114–129, Berlin, Heidelberg. Springer-Verlag.

Microsoft Corporation (2010). Code contracts user manual. http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf.

Object Management Group (2010). Object constraint language version 2.2. http://www.omg.org/spec/OCL/2.2.

Offutt, A. J., Xiong, Y., and Liu, S. (1999). Criteria for generating specification-based tests. In *ICECCS '99: Proceedings of the 5th International Conference on Engineering of Complex Computer Systems*, page 119, Washington, DC, USA. IEEE Computer Society.

Tillmann, N. and Halleux, J. d. (2008). Pex - white box test generation for .NET. In *Proceedings of TAP 2008: the 2nd International Conference on Tests and Proofs*, Lecture Notes in Computer Science, pages 134 – 153. Springer Verlag.

Turing, A. (1949). Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67 – 69.

Warmer, J. and Kleppe, A. (2003). *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.