

MANAGED ASSERTIONS FOR COMPONENT CONTRACTS

Nam Tran, Christine Mingins, David Abramson
School of Computer Science and Software Engineering
Monash University
900 Dandenong Rd, Caulfield East 3145, Australia
Tel: (61 3) 9903 1218 Fax: (61 3) 9903 1077
Email: {*nam.tran, cmingins, david.a*}@csse.monash.edu.au

ABSTRACT: Behavioral specifications in interface contracts are important measures for improving quality of software components. Binary components of different language origins need a common understanding of behavioral contracts to work effectively in component-based systems. We propose a system by which behavioral specifications in the spirit of Design by ContractTM can accompany binary components and be available at runtime to enable flexible and correct treatment, in a language neutral manner. Behavioral contracts written in different specification notations, for components written in different programming languages, have common runtime semantics. Runtime monitoring is correct with regards to behavioral subtyping, object re-entrance and exception handling. The mechanism also enables runtime reflection of contract constructs. We show our design for this system, as an extension to the Common Language Infrastructure (CLI) standardized by ECMA. We also describe our prototype implementation based on Microsoft's Shared Source CLI.

I. INTRODUCTION

Component software technology has gained increasing acceptance as a way to build software systems with better quality and better reuse. Systems built from high quality reusable components are likely to perform better than systems developed from scratch, especially in terms of extensibility, flexibility and openness (Nierstrasz and Dami, 1995; Szyperski, 1998; Bachmann et al., 2000). The key issue is quality. It is now widely recognised that component quality is a key enabling factor to ensure the success of component-based software (Meyer et al., 1998; Szyperski, 1998; Meyer, 1999; Meyer and Mingins, 1999; Szyperski, 2000).

A key challenge for achieving high quality components is the need for better specification of component interfaces. In a component-based system, the constituent components are usually developed independently, possibly in different programming languages, interacting with one another through interfaces only (Szyperski, 1998; Bachmann et

al., 2000). A component interface specification is seen as a contract between the component and its environment, specifying what the component provides its clients and what it requires from the environment in which it executes (Szyperski, 1998; Bachmann et al., 2000; Meyer, 2000; Szyperski, 2000). Therefore, to achieve high quality for components, there should be complete and precise component interface contracts.

To be complete and precise, a contract needs to specify both functional and non-functional aspects (Szyperski, 1998). Functional aspects should include functional operations annotated with pre- and post-conditions, and safety and progress properties. Typical non-functional aspects are space and time properties. A more detailed classification, due to Beugnard et al. (1999), consists of four levels of contracts, namely basic contracts, behavioural contracts, synchronisation contracts, and quality of service contracts. The term behavioural contract is commonly used to refer to interface specifications consisting of functional operations signatures, invariants, pre- and post-conditions.

The Design by ContractTM (DbC) method introduced with the Eiffel language (Meyer, 1992; Meyer, 1997) enables class invariants, method pre- and post-conditions, loop variants and invariants to be written in the source and monitored at runtime. DbC-like support has also been added to other object oriented programming languages, such as Java (Duncan and Hölzle, 1998; Karaorman et al., 1998; Kramer, 1998; Cicalese and Rotenstreich, 1999; Leavens et al., 2000; Bartetzko et al., 2001; Findler and Felleisen, 2001; Leavens et al., 2002) and C++ (GNU Nana), and the OCL modeling language (OMG, 2001). DbC provides essential support for behavioural specifications in a single programming language.

However, as pointed out above, constituent components of a component-based system may be written in different programming languages. The ability to mix languages is an important enabling factor for a successful component software industry as it allows for the utilization of reusable libraries regardless of language of origin (Meyer,

2002). It also enables developers to leverage their skills in different languages. Indeed, most component models, including COM, CORBA, the Common Language Infrastructure (CLI) platform (ECMA, 2001), and the Web services paradigm (Freemantle et al., 2002; Stal, 2002), are inherently multi-language.

Therefore, it is important to have a common mechanism to support behavioral contracts for binary components, including general assertions and DbC, in a manner neutral of source programming languages and assertion notations.

We choose to prototype our ideas in the CLI (ECMA, 2001), as it is a multi-language component platform with a major commercial implementation (Microsoft .NET) and a number of open source implementations (Microsoft Shared Source CLI, Mono by Ximian, and Portable.NET by DotGNU).

The remainder of this paper is organized as follows. In section 2, we further describe DbC and show the need for a language neutral system in order to apply DbC to the world of binary components. We then explain our proposed managed assertions system and show how its common representation format and runtime service correctly support a common contract semantics. Section 4 describes our prototype implementation of that system based on the Microsoft Shared Source CLI in section 4. Related works are discussed in section 5. We draw some preliminary conclusions and suggest further works in the final section.

II. DbC AND COMPONENT CONTRACTS

DbC method (Meyer 1997) involves specifying and run-time checking class invariants, method pre-conditions and post-conditions, loop variants and invariants, and general check assertions. The first three constructs correspond to typical behavioral specifications of objects, and are considered the contract between an object and its clients. The last three constructs are generally not part of an object's interface, and are used only for the benefits of the object's developer.

A method pre-condition is an assertion on the visible state of the object and its environment upon method entry. A pre-condition is an obligation on the caller, for the benefits of the method. A method post-condition is an assertion on the visible state of the object and its environment upon method return. A post-condition is an obligation on the method, for the benefits of its clients. A method is only obliged to establish its post-conditions if it has been called with all its pre-conditions satisfied. This essentially follows the total correctness formula, or a Hoare triple:

$$\{Pre\} A \{Post\}$$

which states that any execution of action A, starting in a state where Pre holds, will terminate in a state where Post holds.

A class invariant is a condition that an object of that class needs to maintain, at all points where clients may access the object. Typically, these client access points are entries and returns of public methods. A class invariant assertion can be considered semantically as part of the pre- and post-conditions of all public methods of an object.

DbC follows the behavioral subtyping rules (America, 1991; Liskov and Wing, 1994; Liskov and Wing, 1999). Subtypes may only weaken their method pre-conditions and strengthen their method post-conditions. Invariants, semantically both pre- and post-conditions, must stay the same in subtypes.

Given:

Type T extends type $T_1, \dots, T_n$

$P^T(m)$ denotes the pre-condition of method m in type T

$Q^T(m)$ denotes the post-condition of method m in type T

I^T denotes the invariants of type T

We have these behavioral subtyping constraints:

$$P^T_i(m) \supseteq P^T(m) \quad "i \hat{=} \{1..n\}$$

$$Q^T(m) \supseteq Q^T_i(m) \quad "i \hat{=} \{1..n\}$$

$$I^T \supseteq I^T_i \quad "i \hat{=} \{1..n\}$$

Contracts in DbC are specified in the source program, using native Eiffel (Meyer, 1992) Boolean expression notation for assertions. Pre-conditions are specified with the **require** keyword. Post-conditions are specified with the **ensure** keyword. Both are attached to individual methods. Class invariants are specified with the **invariant** keyword and are attached to the class as a whole. A redefinition of a method in a subtype uses the keywords **require else** and **ensure then** for pre- and post-conditions, respectively. These assertions are combined, through disjunction for pre-conditions and conjunction for post-conditions, with assertions of base types. This is to ensure correct behavioral subtyping for contracts. Followed is an example Eiffel code snippet extracted from Meyer (1997):

```
class Stack
feature
  top: INTEGER is
    require
      not_empty: not empty
    do
      Result := representation @ count
    end
  empty: BOOLEAN is
    do
      Result := (count = 0)
    ensure
      empty_def: Result = (count = 0)
    end
  invariant
    count_non_negative: 0 <= count
    count_bounded: count <= capacity
end
```

Contracts can be monitored at runtime by checking the assertions. The compiler can be instructed to generate runtime code that will check class invariants and pre-conditions upon a method entry, class invariants and post-conditions upon a method return. If a pre-condition fails to hold, a method is not executed and an exception is raised signalling an error in the caller. If a post-condition or a class invariant fails to hold, an exception is raised signalling an error in the object implementation.

DbC as described here is very useful for writing correct programs but is limited to individual object-oriented languages. This is because each of the systems supporting DbC, including the native Eiffel and Java systems referenced in the introduction, has its own assertion notation tied to its specific runtime semantics. Each system has its own way of representing DbC assertions at runtime. Each system also handles checking and violations differently.

The method of DbC is quite applicable to component-based systems. Components interact through interfaces whether or not they are written using the object-oriented approach. Therefore component behaviors can be specified in the interfaces to form component contracts. The runtime facilities enabling components interaction can be made to support runtime contracts checking.

The crucial issue is to support contracts for binary components written in different languages. As argued earlier, component models are inherently multi-language and the ability to mix languages is valuable. Independently developed components need a common understanding of contracts without relying on the availability of source code, just like they have common semantics for a set of basic types. A pre-condition of a method written in one language needs to be checked by callers written in other languages. This requires the component and its clients to agree upon what a pre-condition is, how to check it, and how to handle violations. Similar arguments apply for post-conditions and invariants.

In addition, the specification notations should not be restricted. Assertion notations tied to programming languages may be easier to learn and may make developers more productive. Support for different notations will enable developers to leverage existing tools such as parsers, compilers, or modeling tools. Preference of certain notations may also be a matter of personal style.

We can conclude that a common support system for component contracts, neutral of programming languages and specification notations, is essential for contracts interoperability just as a common type and runtime support system is essential for component interaction. In the next section, we will describe our common contracts system.

III. COMMON CONTRACTS SYSTEM

We propose a system that has a common binary representation for component contracts and a common runtime

support service tightly integrated with the overall runtime. Contract specifications are left up to the higher layers, such as program source compilers or constraints compilers. These compilers translate contract specifications into our binary format. The common runtime service manages the compiled assertions to perform runtime checking and reflection. We call this a *managed assertions system*, similar in spirit to managed execution systems such as the Java Virtual Machine (JVM) or the Common Language Runtime. Figure 1 shows the architecture of our system.

Common semantics. A common representation and a common runtime support (both explained in details below) enable a common contract semantics. With a common representation format, contract constructs are compiled into first class entities with explicitly designated roles. A source level pre-condition assertion of a method is compiled into a code construct designated as a pre-condition for that method. A post-condition assertion is compiled into a code construct designated as a post-condition, and so on. The common runtime service then treats these constructs uniformly based on their designated roles. In other words, an assertion always maintains its explicit role throughout its lifetime, from source notation, to binary representation in the component, and at runtime. This way, the system provides a common semantics to all component contracts, neutral of source languages and assertion notations.

Common representation. The common representation format is a logical extension to the format of binary components. Major systems supporting component-oriented programming, such as Java and .NET, support this type of extension. The extension allows compiled assertions to be stored as separate code blocks, designated with specific roles. The code blocks may be implemented by normal methods. Metadata in the binary components enables a definitive designation of each code block with a role, as a pre-, post-condition, or an invariant. This explicit designation through metadata eliminates the need to rely on naming conventions and runtime introspection. Behavioral contracts then *become part of* binary components. Figure 2 illustrates the logical organization of this representation.

This representation scheme has a number of important features. Class invariants, method pre-conditions and post-conditions are not merged into bodies of methods or wrapper methods at compile time as in many other systems (Kramer, 1998; Cicalese and Rotenstreich, 1999; Bartetzko et al., 2001; Findler and Felleisen, 2001; GNU Nana). This means, the decision to perform runtime check of assertions is not committed to at compile time. As runtime checks can be expensive, it is best to provide the flexibility in deciding whether checks are performed. In addition, in component-based systems, integration testing by component producers is usually not possible. Therefore the producers may wish to have their pre-conditions moni-

tored. Conversely, a client of these components might require their post-conditions to be checked initially. System integrators might want to change these policies based on their experience with the constituent components.

Moreover, separate representation of assertions enables correct and more efficient runtime check. In the presence of dynamic dispatch, a compiler may not be able to decide which instance of a virtual method will be invoked. Therefore it cannot decide what pre-condition checking code to generate for the caller while logically checking pre-conditions is the caller's responsibility. Additionally, invariants checking needs to differentiate between internal and external calls. These are further explained below.

Runtime support. Our managed assertions system provides runtime checking, correct with respect to behavioral subtyping, exception handling and object re-entrance. It also supports reflection of contract constructs.

To decide whether to perform runtime check, our system first checks relevant configuration information. This check is done at runtime, thereby affording users the utmost flexibility in specifying what contracts to be checked and when.

If the pre-condition assertions of a method are to be checked, the system retrieves the code representing those assertions from the binary component and executes it. Due to tight integration with the overall runtime, the system can rely on it to resolve method dispatch to identify exactly which method is being invoked. The explicit designation through metadata enables easy identification of the relevant code blocks. No introspection of method names following naming conventions is required. Separate assertion code blocks lend themselves to easy retrieval. If the pre-condition holds, the system gives back control to the method for its execution. If the pre-condition fails, the system raises a pre-condition violation exception to the caller without executing the method.

The system performs invariants and post-conditions checking using similar algorithms.

Behavioral subtyping. This mechanism of runtime check correctly follows the rules of behavioral subtyping (America, 1991; Liskov and Wing, 1994; Liskov and Wing, 1999). Recall that according to these rules, the pre-condition of an overridden method must imply the pre-condition of an overriding method. Conversely, the post-condition of an overriding method must imply the post-condition of an overridden method. The invariants of a base type and its subtypes must be logically equivalent. Here we do not explicitly deal with invariants as their treatment easily follows from treatment of pre- and post-conditions.

In our managed assertions system, specification of assertions follows the DbC method closely. When subtyping, the programmer should only specify (using, for example,

the **require else** and **ensure then** keywords) extra assertions specific to overriding methods and must take into account assertions of base types.

By definition, as shown below, the total pre-condition of a method in a subtype is the disjunction of its own specific partial pre-condition and all partial pre-conditions of corresponding methods in base types. The total post-condition of a method in a subtype is the conjunction of its own specific partial post-condition and all partial post-conditions of corresponding methods in base types. Obviously, the total pre-condition of an overriding method in a subtype is implied by the total pre-condition of the overridden method in any base type. The total post-condition of an overriding method in a subtype implies the total post-condition of the overridden method in any base type.

Given:

Type T extends types $T_1, \dots, T_n$

$Pre^T(m)$ is partial pre-condition specified in T for method m

$Post^T(m)$ is partial post-condition in the same sense

$TotalPre^T(m)$ is the total pre-condition of T 's instance of m

$TotalPost^T(m)$ is the total post-condition in the same sense

We have:

$$TotalPre^T(m) = Pre^T(m) \vee Pre^{T_1}(m) \vee \dots \vee Pre^{T_n}(m)$$

$$TotalPost^T(m) = Post^T(m) \wedge Post^{T_1}(m) \wedge \dots \wedge Post^{T_n}(m)$$

$(TotalPre^{T_i}(m) \vdash TotalPre^T(m) \text{ "i" } \hat{I} \{1..n\})$ is a tautology

$(TotalPost^T(m) \vdash TotalPost^{T_i}(m) \text{ "i" } \hat{I} \{1..n\})$ is a tautology

By definition, then, specification of behavioral contracts in this manner is *always* correct with regards to behavioral subtyping. It is the matter for the method implementation to respect these contracts. We believe this arrangement simplifies correct checking a great deal, at least because there are no "hierarchy errors" as defined by Findler and Felleisen (2001).

To check a method's pre-condition at runtime is to check its total pre-condition. Our system traverses the type hierarchy, constructing the total pre-condition by collecting in a disjunction all the partial pre-condition of the corresponding method in base types. In other words, our system executes these partial conditions and collecting their Boolean results in a disjunction. If the end result is a false, the caller has violated the method's pre-conditions.

Checking post-conditions follows in the same fashion except a conjunction is used instead of a disjunction. If the end result is a false, the method implementation has failed to establish the post-conditions as promised by its contract.

Exception handling. Our system raises appropriate exceptions when contract violations occur. We define special exception types, representing pre-condition violations, post-condition violations and invariant violations accordingly.

If checking of pre-condition upon a method entry identifies a violation, a pre-condition violation exception is raised such that the method is not executed and the caller receives the exception. Our system ensures that the caller receives the exception by raising the exception outside the range of all exception handlers, if any, of the callee method. Additionally, since the method is not executed at all, no recovery is required. This excludes recovery from any change that may have occurred as part of arguments evaluation, which is generally required for assertions evaluation anyway.

If violation of a post-condition occurs upon a method return, an exception is raised to the caller to signify a failure of the callee method to keep its promise. It does not make sense to try raising an exception to the callee. This is because the method has effectively completed its attempt at carrying out the work its contract promises. If any meaningful recovery from a post-condition failure is possible, it should be programmed into the logic of the method itself.

There are two cases of invariant violation. In both cases invariant violation exceptions are raised to the caller, but they signify different errors. If a violation occurs upon a method return, this is an error in the callee method. The callee method is the exact culprit because it fails to re-establish invariants that hold upon its entry¹. If a violation occurs upon a method entry, this indicates an error that occurred earlier in the lifetime of this object. This error could have occurred as part of the interaction of this object with other clients, not necessarily the present caller, because of object aliasing.

Object re-entrance. As noted earlier, checking of invariants needs to differentiate between internal and external calls. This is because we want to allow an object implementation to temporarily break its invariants internally. Requiring all methods accessible to outsiders to check invariants regardless of the callers, i.e. to establish all specified invariants at all method entry and return points, would certainly ensure that outsiders never see the object in a state inconsistent with respect to the invariants. However, this is quite restrictive as an object implementation may rely on the internal interaction of its own public methods to perform a task and establish its invariants.

Our system checks invariants (when turned on by configuration) only for external calls, i.e. calls that cross object boundaries. Thanks to its tight integration with the overall runtime, the system can check if the caller and the callee objects are the same before committing to checking the invariants. It therefore can allow internal interactions to temporarily break invariants. However, it ensures that external callers can never call a method, and thereby may see the object's state, if not all specified invariants hold.

¹ We only consider sequential programs here

This prevents many dangerous re-entrant calls by outside clients. Self re-entrant calls can also be dangerous. But the implementer of the object can see all the code involved in such interaction and must manage the risks that entails.

Reflection. Reflection, as supported by common systems such as Java or .NET, enables meta-programming and facilitates dynamic late type binding. Reflection allows runtime creation of meta-objects such as types, classes, methods, fields, etc. These objects enable both runtime code generation and dynamic binding.

Our system, having compiled contracts as first class entities in binary components, supports the runtime creation of objects corresponding to these constructs. It is envisaged that useful tools can be built to make use of this facility. For example, a verifier can conceivably be built to extract the compiled contracts of a component to check for its conformance to some specific requirements before committing to integrating it into a system.

IV. PROTOTYPE IMPLEMENTATION

We prototype our managed assertions system in the Microsoft Shared Source CLI (Rotor). The Common Language Infrastructure (CLI) is the core of Microsoft .NET technology standardized by ECMA (2001). It supports managed execution of components written in various languages that all compile to a Common Intermediate Language (CIL). These components consist of CIL code and metadata, organized in an extensible format, which itself is an extension of the standard Windows executable format.

Binary representation. In the prototype, we represent compiled assertions as special methods. These are CLI methods marked as special using tags defined by the CLI. They need not follow special naming conventions. Each method represents one named assertion. There may be multiple named assertions for a partial condition. For each method, only partial pre- and post-condition are attached in the type defining the method. A total condition will be constructed at runtime as described above.

We extend the metadata format by creating extra metadata tables to link named assertions to their owner methods and types. Our new Assertion metadata table groups all named assertions of a CLI module (part of a CLI assembly – component). Each row represents one named assertion. In this table, each method having pre- and post-conditions has its named assertions stored in consecutive rows. A flag in each of these rows marks it as of kind ‘pre-condition’ or ‘post-condition’ accordingly. Each type also has its named assertions, representing its invariants, stored in consecutive rows. Each of these rows is marked as of kind ‘invariant’. Each row in this table also has a pointer back to its owner method or type. And obviously, each row also has a name field. Each row can also represent a

general named assertion with a specific target application point in the component's code. This organization is depicted in figure 3. We might add an index table and eliminate owner pointers to improve efficiency in the future.

Runtime support. Our prototype system also includes extensions to standard unmanaged and managed metadata APIs of the CLI. These extensions enable retrieval of named assertions and contract constructs, and creation of object representing them at runtime. We have not tried to support assertions creation through the managed API but that would be relatively straightforward to implement.

To provide configurability for runtime checking, the prototype extends the configuration facility of Rotor. Users can toggle checking of a particular kind of assertions by editing configuration files.

Before committing to perform invariants check for a method call, the system checks the stack frame to see if the caller and the callee are the same object. This check enables it to handle object re-entrance properly as described above.

To perform runtime assertion checks for a method, the prototype rewrites its CIL body, just before the method is compiled to native code by the Just-In-Time compiler. It does this by replacing the body of the method with a new one, using the facility provided for code profiling. However, due to its tight integration, it does not need to go through the external profiling interface, thereby improving efficiency and also not monopolizing that interface.

The system rewrites CIL code in a method's body to insert necessary checking and exception handling code. The pseudo-code below illustrates the transformation. New exception types are added to the CLI to indicate contract violations.

```

Sm( $A_1 a_1, \dots, A_n a_n$ ) {
    S result; // extra variable to hold result
    if not ( $T.m\$pre \vee T_1.m\$pre \vee \dots \vee T_n.m\$pre$ )
        throw PreconditionException;
    result = { inlined old body }
end:
    if not ( $T.m\$post \wedge T_1.m\$post \wedge \dots \wedge T_n.m\$post$ )
        throw PostconditionException;
    return result;
}

```

To perform a pre-condition check, it first traverses the type hierarchy and constructs the total pre-condition using the extended metadata APIs. It then performs inlining of calls to the special methods representing the pre-condition at the front of the method body. It inserts code to check the resultant Boolean value and to raise a pre-condition violation exception if the value is false. The original method body is inlined next. If the method is declared to return something non-void, an extra local variable is inserted to hold the return value. Each return instruction is

replaced by an assignment (if the method returns something) and a jump to a fixed label beyond the end of the old body.

To perform a post-condition check, the system first constructs the total post-condition in similar manner. It then performs inlining of calls to the special methods representing the post-condition at the fixed label after the original method body. Extra code to check the resultant Boolean and raises exception as appropriate is then inserted. To perform invariant check, the system generates code similar to the combination of checking pre- and post-conditions.

To ensure proper exception handling, the system also makes appropriate modifications to the original exception handler table (if any). Existing handlers are modified to reflect adjustments in code offsets of the original body so they continue to function properly. These modifications also ensure that any handler in the callee method never catches a contract violation exception. This is because any existing handler would range over code offsets in the inlined old body only.

The system also performs any adjustment of the stack parameter for each rewritten method as necessary.

As of this writing, we have not completed the prototype implementation. However, we plan to make it available soon.

V. RELATED WORKS

Our work demonstrates the need for common behavioral contract semantics for binary components and shows a system that supports the representation and runtime checking of DbC style contracts in a language neutral manner.

There have been many works in adding contracts to programs and supporting their runtime verification. However, most works have been tied with specific programming languages and assertion notations. Also, no works have argued for the need to represent contract constructs as first class entities in the binary format and at runtime. The lack of such common ground in most existing works prevent contracts from interoperating.

Closest to our work is AsmL.NET by Barnett and Schulte (2002). They use AsmL, an executable specification language based on Abstract State Machine (ASM), to write conditions and model programs for .NET components. Their model programs can specify the equivalence of pre-, post-conditions and invariants, plus mandatory calls to constrain component interactions. Our present paper has not concerned much with specific source level notations.

AsmL.NET represents assertions and perform runtime checks in a manner fairly similar to ours. When specified

by model programs, their component contracts are compiled into separate .NET types. When written as DbC style conditions, they are compiled into methods following a naming convention. This is different to our approach where contracts are truly first class entities with designated meanings, requiring no name-based introspection. Each AsmL.NET condition is treated as atomic and unnamed. This contrasts with our support of multiple named assertions for each contract, allowing fine-grain treatment. Also, they do not explicitly show how they handle contracts with subtyping. AsmL.NET also performs method rewriting at immediate code level. However, their contract checking catches normal program exceptions. We believe this is unnecessary since when a program exception occurs no post-condition or invariant checking is meaningful. AsmL.NET also does not deal with object re-entrance.

Another close work to ours is obviously Eiffel itself (Meyer, 1992; Meyer, 1997). We base our contracting method on DbC, originally developed for Eiffel. DbC in Eiffel remains a one-language feature, however. At the present, Eiffel programs can be compiled as .NET components. Nevertheless, any contract written in the original Eiffel source would be unknown to other .NET components. Our present work elevates DbC to language neutral level where binary components written in different languages can all interoperate with contracts.

Many existing systems add DbC to Java. In a multi-language component world, all these works have the same limitations as mentioned at the beginning of this section.

Handshake (Duncan and Hölzle, 1998) allows assertions to be written separately and only instruments Java classes at load time. The instrumentation renames existing methods and provides in their place wrapper methods that check conditions outside the original method. It intercepts calls from the JVM to the operating system, instruments and returns the modified class files. Therefore Handshake does not modify the JVM or the Java compiler.

jContractor (Karaorman et al., 1998) performs similar instrumentation to Handshake but it does so by a specially modified class loader. It can also provide instrumented subclass for explicit instantiation by programmers through factory classes. Using jContractor, programmers write explicit contract methods using standard Java syntax and some special library facilities.

iContract (Kramer, 1998) is a pre-processor system that allows assertions specification using special comment tags. The assertions are written in a specific notation. It then translates these assertions to instrument Java methods at source level.

Jass (Bartetzko et al., 2001) is also a pre-processor system that operates at Java source level to instrument methods. Jass supports detection of hierarchy errors in the spirit of Contract Java (Findler and Felleisen, 2001). The pro-

grammer is required to provide a casting method to create an object of the immediate base type from the current object. The system then checks the conditions on this object as well to detect hierarchy errors. Jass also supports specification of call sequence by trace assertions. This is similar in spirit to mandatory calls in AsmL.NET.

Contract Java (Findler and Felleisen, 2001) adds behavioral contracts to Java, based on a rigorous contract soundness theorem to ensure proper behavioral subtyping. Following this theorem, contract checking at runtime also check if the behavioral subtyping rules hold by walking back the typing hierarchy. If they do not hold, hierarchy errors are flagged. Using this theorem, it was pointed out that many previous Java DbC systems do not check contracts correctly. Our present approach differs markedly in that specification of behavioral contracts is *always* correct by definition. With our approach, checking contracts is more efficient and there are no hierarchy errors.

JML (Leavens et al., 2000; Leavens et al., 2002) is a comprehensive specification language for Java, similar in many ways to AsmL. Assertions written in JML syntax are embedded in Java source programs as special comments. A number of static checking tools have used part of JML as their specification language. A JML runtime assertions checker tool (Cheon and Leavens, 2002) has been built to operate as a source level translator.

VI. CONCLUSIONS AND FUTURE WORKS

This paper presents a managed assertions system that supports representation and runtime verification of behavioral contracts for components, neutral of source programming languages and assertion notations. This system is similar in spirit to a managed execution system where binary components of various source language origins can interoperate.

We have argued that a common behavioral contracts system is essential for achieving component quality. Because independently developed constituent components of component-based systems interact through interfaces only. They need a common understanding of behavioral contracts if they are to obey contracts of one another.

By developing a representation for contract constructs as first class entities and a runtime service, both neutral of source notations, we have a system to provide that common ground. First class representation of contracts enables flexible and dynamic checking. In addition, our tightly integrated runtime service enables correct contracts checking with respect to behavioral subtyping, exception handling, and object re-entrance.

We have also sketched our prototype implementation in the Microsoft Shared Source CLI. Our implementation makes use of standard CLI facilities such as metadata

extension, special methods, and dynamic code re-writing without compromising compatibility.

We have followed the original DbC method faithfully. With Eiffel now supported in .NET, we will next investigate integration of our system with Eiffel.NET. To demonstrate the language neutral nature of our system, we will also try to integrate it with another .NET language, such as C#.

This paper has not tackled the issue of contracts specification at source level except for following the Eiffel DbC approach. Exploration of using more expressive notations with our system is also a topic for future research.

VII. ACKNOWLEDGEMENTS

We thank our colleague Bertrand Meyer for useful discussions on the ideas presented here. We also thank Iman Poernomo for his valuable comments on a draft of this paper.

The work reported here is partly supported by a Monash Graduate Scholarship and a DSTC top-up scholarship for Nam Tran, and a Microsoft Research grant.

REFERENCES

1. P. America, 1991. Designing an Object-Oriented Programming Language with Behavioral Subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of Lecture Notes in Computer Science, pages 60-90. Springer-Verlag, New York, NY, 1991.
2. F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Technical Concepts of Component-Based Software Engineering. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, 2000.
3. M. Barnett and Wolfram Schulte. Contracts, Components, and Their Runtime Verification on the .NET Platform. Technical Report MS-TR-2002-38, Microsoft Research, April 2002.
4. D. Bartetzko, C. Fischer, M. Moller and H. Wehrheim. Jass - Java with Assertions. In Workshop on Runtime Verification, 2001, held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01.
5. A. Beugnard, J.-M. Jezequel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *Computer*, July 1999.
6. G. Cheon and G. T. Leavens. A Runtime Assertion Checker for the Java Modeling Language. Technical Report 02-05, Department of Computer Science, Iowa State University, March 2002.
7. C. D. T. Cicalese and S. Rotenstreich. Behavioral Specification of Distributed Software Component Interfaces. *Computer*, July 1999.
8. A. Duncan and U. Hölzle. Adding Contracts to Java with Handshake. Technical Report TRCS98-32, University of California, Santa Barbara, 1998.
9. ECMA, *Standard ECMA-335: The Common Language Infrastructure*, December 2001.
10. R. B. Findler and M. Felleisen. Contract Soundness for Object-Oriented Languages. In *Object Oriented Programming, Systems, Languages, and Applications*, 2001.
11. P. Freemantle, S. Weerawarana, R. Khalaf. Enterprise Services. *Communications of the ACM*, vol. 45 no. 10, pages 77-82, 2002.
12. GNU Nana Project. Available at <http://www.gnu.org/manual/nana/>.
13. M. Karaorman, U. Hölzle, and J. Bruno. jContractor: Reflective Java Library to Support Design-by-Contract. Technical Report TRCS98-31, University of California, Santa Barbara, 1998.
14. R. Kramer: iContract -- The Java Design by Contract Tool, *Proceedings 26th TOOLS 1998*, pages 295-307, Santa Barbara California.
15. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design Of JML: A Behavioral Interface Specification Language for Java. Technical Report 98-06r, Department of Computer Science, Iowa State University, August 2002.
16. G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby and B. Jacobs. JML: Notations and Tools Supporting Detailed Design in Java. In *OOPSLA 2000 Companion*, pages 105-106, 2000.
17. B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, November 1994.
18. B. H. Liskov and J. M. Wing. Behavioral Subtyping Using Invariants and Constraints. Technical Report CMU-CS-99-156, School of Computer Science, Carnegie Mellon University, July 1999.
19. B. Meyer. Contracts for Components. *Software Development*, July 2000.
20. B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

21. B. Meyer. *Object Oriented Software Construction*. 2nd Edition. Prentice Hall PTR Upper Saddle River, 1997
22. B. Meyer. Polyglot Programming. *Software Development*, May 2002.
23. B. Meyer. Rules for Component Builders. *Software Development*, May 1999.
24. B. Meyer and C. Mingins. Component-Based Development: From Buzz to Spark. *Computer*, July 1999.
25. B. Meyer, C. Mingins, and H. Schmidt. Providing Trusted Component to the Software Industry. *Computer*, May 1998.
26. O. Nierstrasz and L. Dami. Component-Oriented Software Technology. In *Object-Oriented Software Composition*, O. Nierstrasz and D. Tschritzis (Eds.), pages 3-28, Prentice Hall, 1995.
27. Object Management Group. *OMG Unified Modeling Language Specification*, Version 1.4, 2001.
28. M. Stal. Web Services: Beyond Component-Based Computing. *Communications of the ACM*, vol. 45 no. 10, pages 71-76, 2002.
29. C. Szyperski. Components and Contracts. *Software Development*, May 2000.
30. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

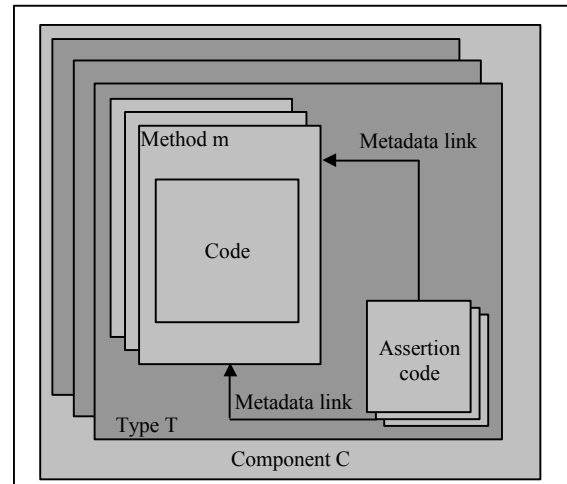


Figure 2. Contracts representation

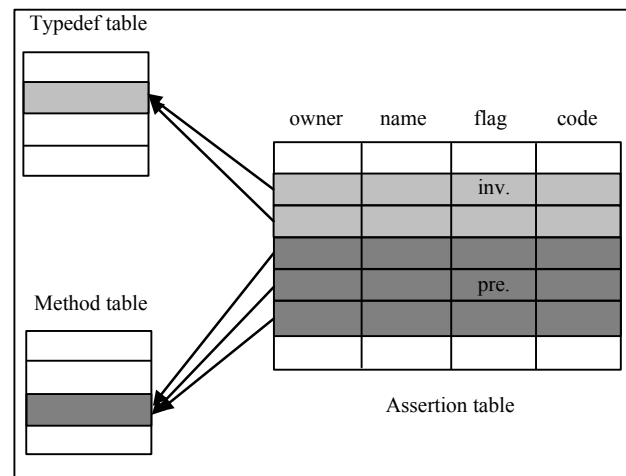


Figure 3. Contracts in CLI binary format

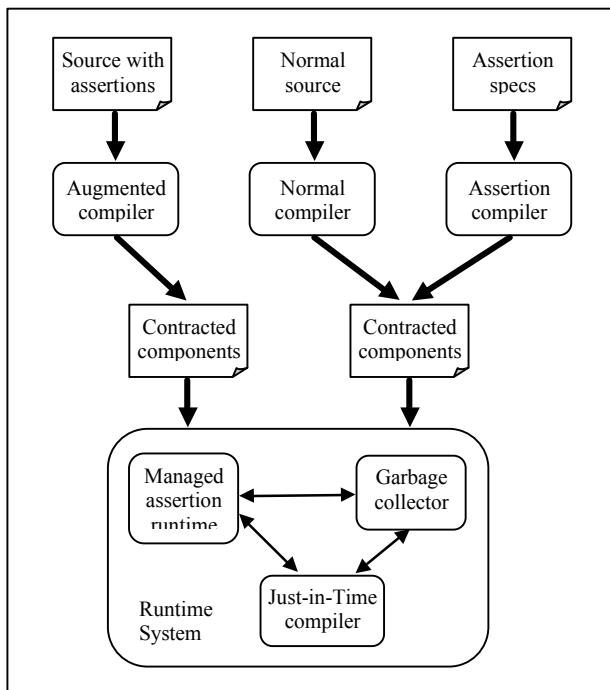


Figure 1. Managed assertions system architecture