Design and implementation of assertions for the common language infrastructure

N. Tran, C. Mingins and D. Abramson

Abstract: Assertions are a well established mechanism for the specification and verification of program semantics in the forms of pre-conditions, post-conditions and invariants of object and component interfaces. Traditionally, assertions are typically specific to individual programming languages. The ECMA Common Language Infrastructure (CLI) provides a shared dynamic execution environment for implementation and interoperation of multiple languages. The authors extend the CLI with support for assertions, in the Design by Contract style, in a language-agnostic manner. Their design is flexible and powerful in that it treats assertions as first class constructs in both the binary format and in the run-time while leaving the source level specification choices completely open. The design also enforces behavioural sub-typing and object re-entrance rules, and provides sensible exception handling. The implementation of run-time monitoring in Microsoft's Shared Source CLI (a.k.a. Rotor) integrates with the dynamic run-time, performing just-in-time code weaving in a novel way to maximise efficiency while operating at the platform-neutral level.

1 Introduction

Design by Contract (DbC) is both a method and a set of language features that support component interface specifications. These specifications are seen as sets of contracts between a component and its environment, specifying what the component provides its clients and what it requires from the environment in which it executes [1–4]. The DbC method introduced with the Eiffel language [5, 6] enables class invariants, method pre- and post-conditions, loop variants and invariants to be written in the source and monitored at run-time. DbC-like support has also been added to other object oriented programming languages, such as Java [7–14] and C++ [15] and the OCL modelling language [16]. The work on extended static checking (ESC) supports specification and static checking of invariants, pre- and post-conditions for Modula-3 [17] and Java programs [18]. However, there is currently no single mechanism that supports DbC-like assertions in a language-neutral manner.

Our work investigates the design and implementation of DbC-style assertions in a language-neutral run-time environment, specifically the ECMA common language infrastructure (CLI) [19]. The CLI is a multi-language component platform with a major commercial implementation (Microsoft .NET) and a number of open source implementations (Microsoft Shared Source CLI [Note 1], Mono [Note 2] and Portable.NET [Note 3]). We implement our current prototype for the Microsoft Shared Source CLI [20].

1.1 Rationale

In the CLI, compilers for all source languages emit a common intermediate language (CIL); they in effect transform the source language types and abstractions into a representation that conforms to a common type system (CTS). This enables components written in different languages to interoperate, regardless of the original source language. The ability to mix languages is an important enabling factor for a successful component software industry as it allows for the utilisation of reusable libraries regardless of language of origin [21]. Equipping such binary components with DbC-like behavioral assertions would provide the component model with much stronger interface semantics.

Independently developed components need a common understanding of contracts without relying on the availability of source code. 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 violation. Similar arguments apply for post-conditions and invariants. 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 run-time support system is essential for component interaction.

Our approach provides support for assertion-based component contracts as first class constructs in the binary format and a common run-time support service tightly integrated with the overall run-time. Thus behavioural contracts become part of binary components. Contracted components understand contracts of one another regardless

IEE Proceedings online no. 20030988

doi: 10.1049/ip-sen:20030988

Paper received 29th June 2003

The authors are with the School of Computer Science and Software Engineering, Monash University, 900 Dandenong Road, Caulfield East 3145, Australia

Note 1: Available at http://msdn.microsoft.com/net/sscli/

Note 2: Available at http://www.go-mono.org

Note 3: Available at http://www.southern-storm.com.au/portable_net.html

of source languages. Reference [22] gives a more detailed discussion of the benefits of such a system.

1.2 CLI overview

The CLI is the core of Microsoft .NET technology and it has been standardised by ECMA/ISO [19]. The CLI supports managed execution of components written in various languages that all compile to a common intermediate language (IL). Compilers targeting the CLI emit intermediate language (IL) byte code. IL byte code is then JIT compiled to native code and executed in the CLI run-time. These components consist of IL code and metadata, organised in an extensible format.

The CLI specifies an IL assembler textual format that can represent all possible elements of a CLI component. Source level compilers and tools can generate code in IL assembler format, which can then be assembled by the IL assembler tool to generate binary components.

The CLI also specifies an equivalent binary component format consisting of metadata elements and IL code. Compilers and tools can also target this binary format directly.

1.3 Design and implementation overview

Our system supports DbC-style assertions, namely pre-conditions, post-conditions and invariants. Pre-conditions may be checked before a method is executed. Post-conditions may be checked after the method has been executed. Invariants may be checked both before and after method execution for calls crossing object boundaries. Assertion failures result in appropriate exceptions. In our current work, we do not include supports for loop variants and invariants as they are more specific to the implementation of a component and typically not visible as part of a component contract. We will investigate this issue further in our future research into the expressiveness of assertion notations.

The design includes an extension to the IL assembler grammar [19] for assertion specifications in textual IL and an extension to the binary metadata format [19] for assertion designation in binary components. Essentially, code for evaluating assertions is represented by special IL methods. Assertions may be part of the component at the source level in any suitable notation or be added to a binary component after compilation. Figure 1 illustrates the overall architecture of our system.

Assertion monitoring is dynamic and configurable. Monitoring policy is checked at run-time and assertions monitoring code is dynamically weaved into program code. The same binary component can be run with assertions selectively turned on or off.

The design also supports assertion inheritance and behavioural sub-typing [23–25]. A subtype inherits its base types' assertions. Its contract must not require more and must not ensure less than a base type's contract.

Our implementation performs just-in-time code weaving. Activation of assertion evaluation code occurs, according to monitoring policy, just before the IL-to-native code compiler compiles a method.

The assertion monitoring service differentiates between internal calls and calls that cross object boundaries in its handling of invariants. It also dynamically walks the type hierarchy to handle assertion inheritance.

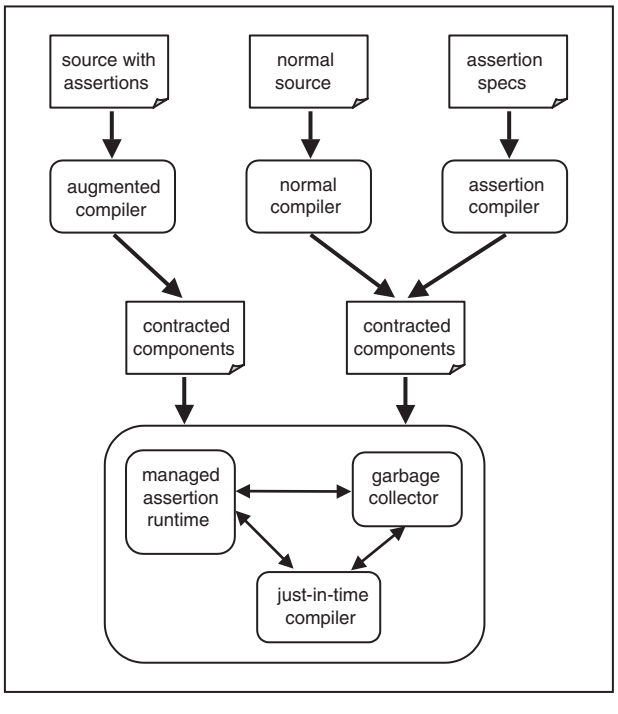

Fig. 1 *Overall architecture of assertions system for the CLI*

2 Assertions in the IL assembler

Our system aims to support the widest scope possible for source-level assertion specification. As long as source compilers and tools can compile specifications for pre-conditions, post-conditions and invariants into corresponding constructs in IL assembler code or IL binary code and metadata, our system understands and can verify them at run-time. Typically, assertions are Boolean expressions or predicates relating to the visible state of an object. However, the formats and expressiveness of source-level assertion specifications are topics of future research and are beyond the scope of this paper. In this section we show how assertions can be specified in IL assembler, while the binary metadata for assertions is the topic of the next section.

2.1 IL assembler assertions example

The following example (Fig. 2) shows fragments of a *contracted* stack implementation in IL assembler. The **.assert** constructs designate method pre-conditions, post-conditions and type invariants. Programmers can write contracted components in IL assembler this way, but the main usage is for compilers and tools that generate IL assembler code.

The **.assert invariant** construct specifies a named invariant for the class Stack. The assertion's name is typically declared by the programmer and would be used in exception handling. It also specifies an evaluation method, whose execution will return a Boolean value indicating if the invariant holds. This evaluation method is marked as special and typically has a name generated from the assertion's name.

Similarly, the **.assert require** and **.assert ensure** constructs specify named pre-conditions and post-conditions, respectively, along with their evaluation methods.

More specially, the **.assert prolog** construct specifies a prolog method to save old values as necessary. The prolog method would be executed just before the execution of the Push method. Old values are saved into the array of objects passed as a parameter. The post-condition method check_height_inc requires access to the old value of the stack height. It gets access to this old value through the same array of object, also passed in as parameter.

```
.class public unicode Stack extends [mscorlib]System.Object {
  // the invariant that stack height is always within valid range
  .assert invariant valid_height {
    .eval instance bool check_valid_height()
  }

  // method to evaluate the invariant
  .method protected hidebysig specialname rtsspecialname instance
          bool check_valid_height()
  {
    // code to check that 0 <= stack height <= max height
  }

  .method public hidebysig instance Push(object x)
  {
    .assert require not_full {
      .eval instance bool check_not_full(object x)
    }
    .assert ensure pushed {
      .eval instance bool check_pushed(object x)
    }
    .assert ensure height_inc {
      .eval instance bool check_height_inc(object x, object[])
    }
    .assert prolog save_olds {
      .eval instance void save_olds(object x, object[])
    }
    .maxstack 8
    // code to do push
  } // end method Push

  // method to evaluate not_full
  .method protected hidebysig specialname rtsspecialname instance
          bool check_not_full(object x)
  {
    // code to check that stack height < max height
  }
  // method to evaluate check_pushed
  .method protected hidebysig specialname rtsspecialname instance
          bool check_pushed(object x)
  {
    // code to check that top of stack is the given item
  }
  // method to evaluate check_pushed_height
  .method protected hidebysig specialname rtsspecialname instance
          bool check_height_inc(object x, object[])
  {
    // code to check that new height = old height + 1
  }
  // method to save old values as necessary
  .method protected hidebysig specialname rtsspecialname instance
          void save_olds(object x, object[])
  {
    // code to save old values as necessary
  }
  // rest of Stack code
}
```

Fig. 2 *Contracted stack in IL assembler*

2.2 IL assembler assertion grammar

The grammar extension for the IL assembler, as specified in [19] is shown in Fig. 3. Only the **invariant** attribute is legal for assertions belonging directly to classes as it denotes type invariants. The other attributes are only legal for assertions belonging to methods. The **.eval** construct specifies the evaluation method, which is by convention marked as *specialname* and *rtsspecialname*. The evaluation method must return a Boolean value,

```
<decl> ::=
  | .class <classHead> { <classMember>* }
  | ...
<classMember> ::=
  | ...
  | .method <methodHead> { <methodBodyItem>* }
  | <assertion>
<methodBodyItem> ::=
  | ...
  | <assertion>
<assertion> ::= .assert <assertAttr> <assertName>
                { <assertMember>* }
<assertAttr> ::=
  | invariant
  | require
  | ensure
  | prolog
<assertMember> ::= <dottedName>
<assertMember> ::=
  | .eval <callConv> <type> [<typeSpec> ::]
          <methodName> ( <parameters> )
  | .other <callConv> <type> [<typeSpec> ::]
          <methodName> ( <parameters> )
```

Fig. 3 *IL assembler grammar for assertions*

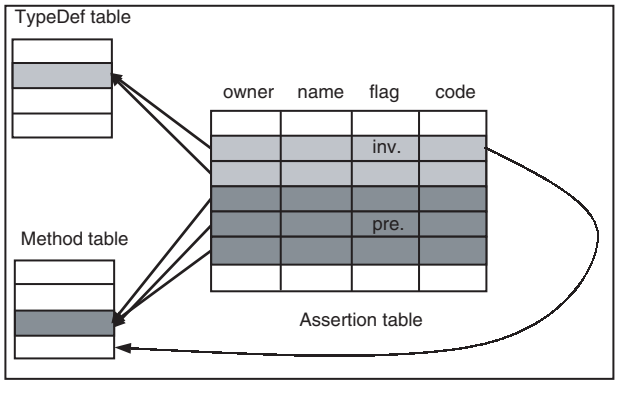

Fig. 4 *Assertion metadata table*

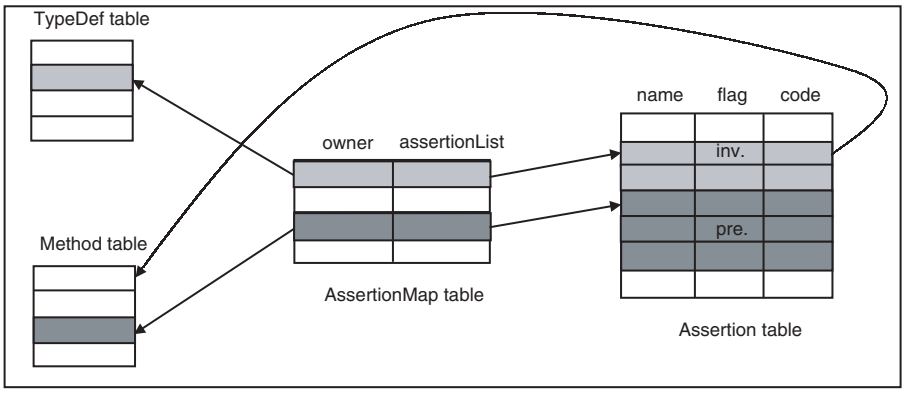

Fig. 5 *Alternative design, Assertion and AssertionMap tables*

except in the case of a prolog construct where its return type is void. The **.other** construct is specified here for possible future extensions.

As can be seen from the example and the grammar, the assertions are first class constructs in the IL program, providing explicit designation of contract roles and not relying on method naming conventions. The assertion construct is very similar to the property and event constructs in the existing CLI specification [19]. However, unlike properties and events, which are essentially syntactic sugar at source code and IL level, assertions are first class constructs in the run-time as well, as will be illustrated in following Sections.

3 Assertions in the binary components

In this Section, we firstly present and compare two designs for explicit metadata representation of assertions. We then briefly discuss the alternative of using custom attributes. Our current prototype implements the first design.

3.1 Assertion metadata tables

In the first design, corresponding to the **.assert** construct in the IL assembler is an Assertion metadata table. This is an extension to the existing binary metadata logical format [19]. The Assertion metadata table contains all assertions declared in a module. Figure 4 illustrates this logical design.

The TypeDef and Method tables are specified by the existing metadata format. As their names imply, a TypeDef table contains type definitions of a module and a Method table contains method definitions of a module.

In our Assertion table, each row represents a named assertion, consisting of an owner field, a name field, a flags field and a code field. The owner field is an index into the TypeDef or Method table. The name field is an index into the string heap. The flags field contains a bit-mask value, indicating the kind of the assertion, i.e. pre-condition, post-condition, invariant or prolog. The code field is an index into the Method table to indicate the evaluation method of this assertion. The table is sorted so that assertions owned by the same type or method lie in consecutive rows. Owner fields of invariants are indices into the TypeDef table, while owner fields of other assertions are indices into the Method table.

The second design is slightly different and involves two tables as illustrated in Fig. 5. The AssertionMap table essentially acts as an extension column to the TypeDef and Method tables. Its owner field provides this link. Its assertion list field contains the index into the Assertion table, marking the first of a contiguous run of assertions owned by the corresponding type or method. The Assertion table is still sorted in the same manner, but does not include an owner field anymore.

Both designs do not involve any modification to existing tables. This is to maximise compatibility with CLI implementations that do not understand the new tables by ignoring them. As a result, contracted components should still work in existing CLI implementations.

The second design is slightly more complex to implement but is potentially slightly more efficient. Regarding space, on average each owner (a type or a method) needs $3n + 2$ fields to hold its assertion information, where n is the average number of assertions per owner. This could be less than the $4n$ fields required by the first design if on average each owner has more than two assertions. Regarding speed, the typical run-time operation on the Assertion metadata is searching for the first assertion of a given type or method. The first design requires a linear search of a p-row Assertion table, where p is the number of assertions. The second design requires a linear search of a q-row AssertionMap table, where q is the number of types and methods, plus an index operation. In good coding practices where assertions are frequently used, the number of assertions should exceed the number of types and methods. In such cases, the second design works faster.

3.2 Custom attribute alternative

A more markedly different design is to use custom attributes, which are metadata elements whose contents are defined by custom managed classes [Note 4]. Custom attributes can be applied to almost every language construct in the CLI, including types and methods.

It is possible to add a custom attribute class to the system library to represent an equivalent assertion construct. An instance of this class would hold the name of an assertion, its kind, and the name of its evaluation method. However, it cannot easily hold an index into the Method metadata table as the index is only computed when the executable file is generated, while any parameter to the construction of a custom attribute must be known at the source code or IL assembler level. In addition, a metadata table run-time check is faster compared to custom attributes as stated by [19]. Therefore the design using custom attributes is significantly less efficient than explicit Assertion tables. Moreover, a solution using explicit metadata tables constitutes more in-built support for assertions, which is our aim.

Note 4: While using custom attributes for source-level assertion specifications might be possible, by having special compiler treatment, it is not strictly related to our discussion here as we leave source-level choices open.

4 Rules for assertions

The design of our assertion system is also characterised by a number of rules that source-level tools and our own run-time service must follow. These include translation rules, which are most important for source-level tools, exception handling rule, object re-entrance rule, sub-typing and assertion inheritance rule, and old values handling rule.

4.1 Assertion translation

Source-level tools can generate assertion constructs in either IL assembler format or binary metadata format directly. Recall that we do not restrict the format of the source-level assertion notations, which can be language neutral or specific to a programming language. A pre-condition assertion is compiled into a pre-condition method whose IL code evaluates the assertion. The name of the method is insignificant because we are not relying on naming conventions. A pre-condition method returns a Boolean value, and takes exactly the same set of parameters as the method for which it is a pre-condition. A pre-condition method is static if its owner is static, and is instance if its owner is instance. However, it is never abstract or virtual as there must always be code to evaluate a condition, even for an abstract method, and assertion inheritance is handled specially. Code within a pre-condition method has the same access to the object state as the owner method, including arguments but excluding the owner's local variables.

Similar rules apply to post-condition methods, prolog methods and invariant methods, with two differences. Firstly, a post-condition method may and a prolog method always take an extra parameter as the last parameter to hold the array of objects containing any necessary old values. Secondly, an invariant method is always an instance method.

For tools that generate binary code directly, our extensions to the metadata APIs of the CLI help with generating the metadata tables. The tools must still perform their own assertion method generation, of course. Those tools that generate IL assembler code must also generate the new assertion directives themselves.

4.2 Exception handling

With run-time assertion monitoring, an assertion failure will result in an exception of a designated type.

A pre-condition failure will result in a pre-condition violation exception being raised to the caller. This is because it is the caller's responsibility to ensure pre-condition assertions hold.

Similarly, a post-condition failure will result in a post-condition violation exception being raised to the caller. While it is the callee's responsibility to uphold post-condition assertions, the callee would have finished its execution by the time of the failure, and handling such an exception at that point is not good programming practice [6]. A post-condition exception indicates to the caller the failure of the callee.

An invariant failure can indicate one of two possibilities. A failure upon method return indicates a fault on the callee's part, similar to a post-condition failure. This is because, ignoring possible concurrent modifications of object state addressed by the invariant, upon method entry that very invariant must have been held. A failure upon method entry indicates a fault that occurred earlier in the lifetime of that object, most possibly due to a fault in a method that was called internally.

4.3 Object re-entrance

Our system checks invariants (when turned on by configuration) only for external calls, i.e. calls that cross object boundaries. 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 then re-establish its invariants [1].

4.4 Sub-typing and assertion inheritance

Following the DbC methodology [6], assertions are inherited. A subtype only specifies the extra assertions it requires in addition to those it inherits. The specified assertions are called 'partial conditions' as they will be combined with those inherited to form full conditions. Within a single type, multiple assertions of the same kind for the same owner are also combined, always with logical conjunction.

In addition, the behavioural sub-typing rules [23–25] are observed so that a subtype can only require no more and provide no less compared to its base type. By definition, the full pre-condition of a method in a subtype is the disjunction of its own specific partial pre-condition and all partial pre-conditions of overridden methods in base types. The full post-condition of a method in a subtype is the conjunction of its own specific partial post-condition and all partial post-conditions of overridden methods in base types. Specification of behavioural contracts in this manner is *always* correct with regard to behavioural subtyping. It is a 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 [14].

4.5 Handling old values

In using method post-conditions, it is often desirable to refer to the values of various entities, such as object public fields and method parameters, *before* method execution. Typically, these old values need to be saved upon method entry. Copying these values can potentially cause side-effects if the assertion notation allows old value expressions with side-effects. This risk is also inherent in evaluating any assertion, however. The general rule, which is not easy to strictly enforce, is to use only side-effect free expressions in assertion specifications [6].

Old values are handled by separate methods through parameter passing where a variable of type Array of Object is used to hold the old values required for a method's post-conditions. The run-time monitoring service is responsible for declaring the variable, and passing it to appropriate methods. The source-level tool generates a prolog method to save necessary old values into a passed in variable. Post-condition methods requiring access to old values would take an extra parameter that is the array of old values. The same source-level tool is responsible for both the prolog and post-condition methods. Therefore, it knows how to encode access to individual elements in the array in a consistent way. While this solution is not the most efficient, it is relatively simple to operate at the IL code level with separate methods for checking post-conditions.

A possibly more efficient but certainly more complicated alternative is to use local variables in both prolog and

post-condition methods to hold old values. The source-level tool has complete control over both methods and thereby is able to map between two sets of local variables. The run-time service, however, must map the two sets of local variables into one by inlining both the prolog and post-condition methods into a common body. This inlining also requires modifications to instructions accessing those local variables as they now have different indices. If any short form instruction is to be changed to its long form, offset changes occur and in turn may require changes to branching instructions and exception handling tables.

5 Run-time monitoring

Dynamic run-time monitoring, where monitoring policies are checked and assertion code integrated on demand at run-time, is the ideal strategy for our system. It is firstly almost a necessity because assertions are first class constructs in the representation, separate from program code proper. In addition, the decision to integrate checking code at run-time significantly improves flexibility and efficiency. Various levels of monitoring can be supported whereby only the necessary assertions are checked to reduce run-time overhead. Moreover, our monitoring service can take advantage of the managed execution scheme, as opposed to the traditional compile-link-execute scheme, to efficiently handle subtyping with dynamic dispatch, exception handling and object re-entrance. Reference [22] contains detailed discussions on the benefits of dynamic monitoring and separate representation for assertions.

5.1 Alternatives for dynamic code integration

There are various strategies for integrating checking code at run-time. The strategy of inserting checking code into callers at call-sites can be easily ruled out. This is because of the required duplicated effort at every call-site and the difficulty in dealing with dynamic dispatch. With dynamic dispatch, it is not always easily deducible before run-time exactly which method is called at a call-site. There remain the alternatives of providing wrapper methods at IL or native code level, modifying the JIT compiler to handle return and various call instructions in special ways, and inserting checking code into bodies of callee methods at IL or native code level.

Native code wrappers, special JIT treatment for return and call instructions, and inserting native code into callee methods lead to essentially the same result. They all end up with a special native code stack frame around a callee stack frame. The special wrapper frame executes the assertion checking code. This is essentially the native stub mechanism that Rotor uses for various mechanisms such as code access security and remoting interception. All these strategies, however, involve native code generation at run-time, which is necessarily platform dependent. We prefer to operate in a platform-neutral manner.

Providing wrapper methods at IL level appears to be an attractive solution. It is relatively simple to create a wrapper method. However, the complication lies in adding extra methods to a running system. To avoid having to redirect every call-site, a wrapper method should have the same name and replace the wrapped method in the method table. The wrapped method must then be moved to a new slot. The work involved in changing the method table this way can be quite complicated.

It seems that avoiding the extra method by inlining the wrapped method into the wrapper would be a good solution. We call this inline wrapping. This is essentially the same as weaving checking code into the callee method. We describe inline wrapping in more detail in the next subsection.

5.2 Inline wrapping IL methods

We perform inline wrapping of a method, whose assertions need to be checked, on demand by replacing its IL code body with a new one just before the method is JIT-compiled. We do this by leveraging the facilities provided to profilers. Just before a method is JIT-compiled, our system checks if its assertions are to be monitored and carries out inline wrapping as necessary.

The general algorithm is as follows. Firstly, the system walks the typing hierarchy as necessary to collect all relevant assertion methods. Then a new method body is created in which code to execute invariant methods is inserted first, if invariants of the enclosing type are to be checked. This can be done by inlining the invariant methods, or calling them. Then code to execute pre-condition methods is inserted. If a prolog method is present, code to execute it is inserted, too, after a local variable of type array of object has been declared and passed to it. Then code from the old method body is inlined. This is followed by code to execute post-condition methods and invariant methods as necessary.

Code to execute assertion methods (excluding a prolog method) also checks their returned Boolean values. This also takes into account behavioral sub-typing rules as appropriate. If an assertion failure occurs, execution jumps to code that raises appropriate exceptions. A local variable is added to hold the current assertion being checked so that the assertion that failed can be identified.

If the configuration dictates that invariants are to be checked for a method call, the system inserts code to execute invariant methods wrapped inside a dynamic check. This dynamic check is a call to a helper function that inspects the current and previous stack frames to check if the current call is crossing object boundaries. If the two frames have the same ‘this’ object, the invariant methods will not be executed. With this dynamic check, we can uphold the object re-entrance rule as specified in Section 4.3 without having to create two versions of a method.

Local variables extra to those of the wrapped method are necessary to hold temporary values such as the current assertion, the returned Boolean values. They are added after the existing local variables. This avoids the need to modify existing instructions accessing local variables. The signature blob of the final set of local variables is added to the StandAloneSig metadata table [19] and the new method header is set to point to the new signature blob.

The old method body is inlined as follows. Every instruction is copied unchanged except return and branch instructions (including **leave** and **leave.s**). Every return instruction is replaced with an assignment and an unconditional branch. The assignment is necessary to store a return value if present. The CLI control flow constraints demand that the stack be empty at the instruction following an unconditional branch if it is not the target of an earlier forward branch. In general this can be the case and an analysis to ascertain whether storing away the return value can be avoided seems unjustified. The unconditional branch transfers execution to the point where post-conditions are evaluated. This replacement results in code offset changes (a return is one byte while an assignment and a jump combined are multiple bytes). Consequently, all branch instructions may need modification. To avoid recursive passes to adjust branching offsets, we convert all existing short form branch instructions to their long form

equivalence. Their offset changes are made according to the following two-pass algorithm.

The first pass scans through the old code body, constructing a list of structures representing branch instructions. Each item of the list contains an instruction position and its target offset.

The second pass through the old code body performs instructions copying and offsets adjustment.

- An instruction not being a return or a short branch is firstly copied as is.
- A return or a short branch is turned into a long branch. Each time this occurs, the list of structures is also processed item-by-item.
  - If the item's position is greater than the current position in the new instruction stream (not counting assertion checking code) then it is increased by the difference between the sizes of the long branch and the old instruction it replaces.
  - If the current position value lies between the item's position and its target, the item's target offset is increased in absolute value.
  - If the current position is beyond both the item's position and its target then the corresponding instruction in the new stream is fixed up and the item removed from the list.

After the last instruction of the old body is copied, the list of structures is gone through to fix up any remaining branch instruction.

Adjustment to offsets used in any existing exception handling table is done in similar fashion to ensure continued correct behaviour. Moreover, this also ensures that any assertion exception raised by checking code is not caught by mistake by existing exception handlers.

While it is tempting to seek a solution that works by directly manipulating the IL buffer without changes to code offsets, there seems to be none readily workable. All the return instructions must be redirected to one point where post-conditions are checked. There is no same-size instruction that can replace the return and achieves this purpose. Wrapping a try-finally construct around the whole body would not work at the binary IL code level because return instructions are illegal within a try block.

An alternative is to reverse engineer the IL code back to abstract syntax trees or similar structures, or even higher levels, and apply changes there before re-generating IL code. The Common Compiler Infrastructure (CCI) for .NET [Note 5] is a recent technology that facilitates this alternative implementation technique.

6 Related work

There has been much work done adding contracts to programs and supporting their run-time monitoring. This includes work on model checking and extended static checking, both of which are fairly different to our domain of dynamic run-time assertion monitoring. Typical model checkers, such as SPIN [26], work by logical analysis to verify that finite state models of real systems satisfy their temporal logic specifications. Extended static checkers, such as those for Modula-3 [17] and Java [18], perform static check of program code against assertions by dataflow analysis or theorem proving. Of the existing work in dynamic run-time assertion monitoring, however, most has been tied to specific programming languages and assertion notations.

Closest to our work is AsmL .NET by Barnett and Schulte [27]. 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 is not much concerned with specific source level notations. Also, no work has argued for the need to represent assertions as first class constructs in the binary format and at run-time. The lack of such common ground in most existing work prevents assertion-based contracts from interoperating.

AsmL .NET represents assertions and performs run-time 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 assertions 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 [5, 6]. Our assertion constructs are styled on DbC, originally developed for Eiffel. DbC in Eiffel remains a one-language feature, however. At present, Eiffel programs can be compiled as .NET components [28]. If an Eiffel .NET component was compiled with assertion monitoring turned on, an assertion failure in that component will be indicated by a .NET exception. Otherwise, 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 [7] 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 [8] 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 [9] is a pre-processor system that allows assertion 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.

Note 5: https://faculty.university.microsoft.com/2003/uploads/7496_115_Rainier_CCI Quick Overview_Adams.ppt http://research.microsoft.com/Collaboration/University/Europse/events/dotnetcc/Version2/Crash Course.ppt

Jass [13] 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 [14]. The programmer 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 [14] adds behavioural contracts to Java, based on a rigorous contract soundness theorem to ensure proper behavioural subtyping. Following this theorem, contract checking at run-time also checks if the behavioural subtyping rules hold by walking back up 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 assertions obey behavioural sub-typing rules by definition. With our approach, checking contracts is more efficient and there are no hierarchy errors.

JML [11, 12] 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 run-time assertion checker tool [29] has been built to operate as a source level translator.

7 Conclusion

This paper has presented the design and implementation of our system to support DbC-style assertions in the CLI. Assertions are first class constructs in the IL assembler, the binary metadata format and in the run-time. Our design is flexible and powerful enough to leave source-level assertion specification choices open while allowing uniform treatment in the run-time. Therefore the system should be able to support interoperability between useful ranges of specification technologies and over different programming languages. For example, DbC assertions written for Eiffel programs, if compiled to our extended CLI, should be usable by components written in other CLI languages.

8 Future work

Indeed, a possible further investigation is to get the Eiffel .NET compiler to generate assertions in our format. This will make the assertions available to other components. It would be interesting to compare performance levels of assertion checking of our scheme and of native Eiffel .NET. Another interesting source language experiment would be to add DbC-style assertion support to C#. A case study of language interoperability for assertions would be another interesting work in this direction.

Of immediate interest to us is the completion of our implementation by measuring performance overheads of assertion support. This can be done by monitoring some example runs of programs with and without assertions to evaluate overhead of assertions with no run-time checks versus assertions with run-time hits versus no assertions at all.

In a further future, we aim to experiment with assertion specification notations of increasing expressive power. AsmL and other more formal specification notations would be interesting examples.

9 Acknowledgment

We thank our colleague Bertrand Meyer for useful discussions on the ideas presented here. We also thank the anonymous reviewers whose comments and suggestions help improved the paper greatly. 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 Rotor award.

10 References

1 Szyperski, C., Gruntz, D., and Murer, S.: 'Component software: beyond object-oriented programming' (Addison-Wesley/ACM Press, London, 2002, 2nd edn.)
2 Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., and Wallnau, K.: 'Technical concepts of component-based software engineering'. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, 2000
3 Meyer, B.: 'Contracts for components', *Soft. Dev.*, 2000, **8**, (7), pp. 51–53
4 Szyperski, C.: 'Components and contracts', *Soft. Dev.*, 2000, **8**, (5), pp. 55–57
5 Meyer, B.: 'Eiffel: the language' (Prentice Hall, New York, 1992)
6 Meyer, B.: 'Object oriented software construction' (Prentice Hall, Upper Saddle River, 1997, 2nd edn.)
7 Duncan, A., and Hölzle, U.: 'Adding contracts to Java with Handshake'. Technical Report TRCS98-32, University of California, Santa Barbara, 1998
8 Karaorman, M., Hölzle, U., and Bruno, J.: 'jContractor: reflective Java library to support design-by-contract'. Technical Report TRCS98-31, University of California, Santa Barbara, 1998
9 Kramer, R.: 'iContract – the Java design by contract tool'. Proc. 26th Conf. on Technology of Object - Oriented Languages (TOOLS), 3–7 August 1998, Santa Barbara, California, pp. 295–307
10 Cicalese, C.D.T., and Rotenstreich, S.: 'Behavioral specification of distributed software component interfaces', *Computer*, 1999, **22**, (7), pp. 46–53
11 Leavens, G.T., Leino, K.R.M., Poll, E., Ruby, C., and Jacobs, B.: 'JML: notations and tools supporting detailed design in Java'. OOPSLA 2000 Companion, 2000, pp. 105–106
12 Leavens, G.T., Baker, A.L., and Ruby, C.: 'Preliminary design of JML: a behavioral interface specification language for Java'. Technical Report 98-06r, Department of Computer Science, Iowa State University, August 2002
13 Bartetzko, D., Fischer, C., Moller, M., and Wehrheim, H.: 'Jass - Java with Assertions', *Electron. Notes Theor. Comput. Sci.*, **55**, (2)
14 Findler, R.B., and Felleisen, M.: 'Contract soundness for object-oriented languages'. Proc. Conf. on Object Oriented Programming, Systems, Languages and Applications (OOPSLA), 2001, pp. 1–15
15 GNU Nana Project, Available at http://www.gnu.org/manual/nana/, accessed 15 September 2002
16 Object Management Group, 'OMG unified modeling language specification', Version 1.4, 2001
17 Detlefs, D.L., Leino, K.R.M., Nelson, G., and Saxe, J.B.: 'Extended static checking'. Research Report 159, Compaq Systems Research Center, December 1998
18 Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., and Stata, R.: 'Extended static checking for Java'. PLDI '02: Proc. ACM SIGPLAN 2002 Conf. on Programming Language Design and Implementation, Berlin, Germany, June 2002, pp. 234–245
19 ECMA, Standard ECMA-335: The Common Language Infrastructure, December 2001
20 Stutz, D., Neward, T., and Shilling, G.: 'Shared source CLI essentials' (O'Reilly, Sebastopol, CA, 2003)
21 Meyer, B.: 'Polyglot programming', *Soft. Dev.*, 2002, **8**, (5), pp. 68–71
22 Tran, N., Mingins, C., and Abramson, D.: 'Managed assertions for component contracts'. To be presented at Conf. on Integrated Design and Process Technology (IDPT), Austin, December 2003
23 America, P.: 'Designing an object-oriented programming language with behavioral subtyping', *Lec. Notes Comput. Sci.*, 1991, **489**, pp. 60–90
24 Liskov, B.H., and Wing, J.M.: 'A behavioral notion of subtyping', *ACM Trans. Program. Lang. Syst.*, 1994, **16**, (6), pp. 1811–1841
25 Liskov, B.H., and Wing, J.M.: 'Behavioral subtyping using invariants and constraints'. Technical Report CMU-CS-99-156, School of Computer Science, Carnegie Mellon University, July 1999
26 Holzmann, G.J.: 'The model checker SPIN', *IEEE Trans. Softw. Eng.*, 1997, **23**, (5), pp. 279–295
27 Barnett, M., and Schulte, W.: 'Contracts, components and their run-time verification on the .NET platform'. Technical Report MS-TR-2002-38, Microsoft Research, April 2002
28 Simon, R., Stapf, E., and Meyer, B.: 'Full Eiffel on the .NET framework'. MSDN Library, July 2002. Available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/pdc_eiffel.asp, accessed 15 September 2002
29 Cheon, G., and Leavens, G.T.: 'A run-time assertion checker for the Java modeling language'. Technical Report 02-05, Department of Computer Science, Iowa State University, March 2002