Contracts, Components, and their Runtime Verification on the .NET Platform

Mike Barnett and Wolfram Schulte
Microsoft Research
One Microsoft Way
Redmond WA, 98052-6399, USA
{mbarnett, schulte}@microsoft.com

April 2002

Technical Report
MSR-TR-2002-38

We propose a method for implementing behavioral interface specifications on the .NET Platform. Our interface specifications are expressed as executable model programs. Model programs can be run either as stand-alone simulations or used as contracts to check the conformance of an implementation class to its specification. We focus on the latter, which we call *runtime verification*. In our framework, model programs are expressed in the new specification language AsmL. We describe how AsmL can be used to describe contracts independently from any implementation language, how AsmL allows properties of component interaction to be specified using *mandatory calls*, and how AsmL is used to check the behavior of a component written in any of the .NET languages, such as VB, C♯, or C++.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

1 Introduction

Component-oriented programming provides an ideal domain for specification technology. Since clients are ignorant of the implementation details, they are forced to rely on a component's specification in order to understand its behavior.

This paper proposes a flexible scheme for attaching stand-alone executable specifications — contracts — to components either statically or dynamically at runtime. Once attached, they monitor the execution of the component and signal any discrepancy in the implementation's behavior relative to its specification. The focus of this paper is on such *runtime verification*. However, there are other uses for executable specifications. They can be used during the testing process to derive test cases and predict how the component should behave. During the design process, they can simulate a design, allowing one to explore its properties before committing to the long development process.

Static vs. Dynamic checking. There is a broad continuum of verification technology for software, ranging from simple program assertions to full mechanically-verified (and sometimes mechanically-generated) formal proofs.

In formal verification, specific, often complex, properties of a given program are proven for all possible input data. State-of-the-art systems are Isabelle [30] and PVS [42]. For distributed system protocols, process algebras, e.g., [26], have been used along with the concept of (bi)simulation. Although the technology has improved dramatically over the years, it is still the case that it is at best semi-automatic. Furthermore, its users need a high level of mathematical expertise and the techniques often do not scale well. Currently the cost/benefit ratio is only acceptable for systems that need utmost care.

Static analysis, which is fully automatic, tries to prove simpler, often generic, properties for any given program, e.g., LCLint [13] and ESC/Java [31]. The difficulty introduced by pointer/object aliases often means that the result of static analysis is not precise, but when it is, static analysis works for all possible input data. Work using model checking [3] also can suffer from the combinatorial explosion of the state space.

In contrast, runtime verification monitors the execution of a component. Its guarantees are limited; it verifies only a single program execution with the specific input data. However, runtime verification is easy to apply and it scales with increased program complexity. These two aspects alone make it appealing for us.

Assertions and Contracts. Properties about a program are often expressed using assertions. Assertions go back to Floyd [16] and Hoare [25]. Since then, there has been an enormous amount of research involving the specification of programs, objects, and more recently, components.

The most well-known work in this area is probably that of Eiffel [36]. Eiffel uses assertions in pre- and post-conditions and class invariants. (When the context makes it clear, we will use *condition* to mean either a pre-condition,

post-condition, or invariant.) "A pre-condition and a post-condition associated with a method describe a contract between the class and its clients. The contract is only binding on the method in as much as calls observe the pre-condition; the method then guarantees the post-condition on return." [36]

A contract describes the behavior of a method independently of its implementation. A contract implies that a specification is expressed as a separate unit from the program or implementation that it describes and that there is some means for enforcing, or checking, the implementation to verify its conformance to the dictates of the specification.

Behavioral Properties and Model Programs. What exactly do contracts specify and check? We are interested primarily in the direct input-output behavior of a component's methods. This is typically expressed using declarative conditions. However, we will show that model programs are often easier to write and understand, in particular for programmers. In addition they allow the specification of component interaction. This is required when moving beyond hierarchical libraries to specify architectures in which components stand in a peer-to-peer relationship. For example, the subject-view pattern [18] is characterized by required calls from the subject to the view and potential callbacks from the view to the subject. Such software architectures require some method for specifying the sequencing of calls. Our model programs use the concept of *mandatory calls*: these are the minimal external communications, via public methods, that a component must make during the execution of the method in which the mandatory calls are located.

Model programs are often nondeterminstic. *Nondeterminism* allows the implementation freedom for a range of behavior. For instance, a specification for a lossy network is one that chooses to forget some messages. Or, consider an enumerator for traversing a collection of items. It is often the case that the order in which the enumerator should return elements is not specified: the actual order is the result of implementation decisions about data structures and index ordering. The latter is an example of external choice; we can observe whatever choice the implementation actually makes. The former type of nondeterminism is an example of internal choice: only the internal state of the implementation describes what choice has been made.

Checking sequencing and nondeterminsim are two of the outstanding challenges in the runtime verification of model programs, which we tackle in this paper.

Runtime Verification. Contracts in the form of conditions allow for their injection into the class or method to which they apply. Implementations we are aware of range from pre-processor source-level transformations [33] to object wrappers [32, 12] to byte-code rewriting [11]. Logically, the effect is shown in Figure 1: the interaction between a client and an implementation is mediated by a monitor that verifies it relative to a specification. The key idea is that the specification is a *parallel* construct to the implementation; this has been

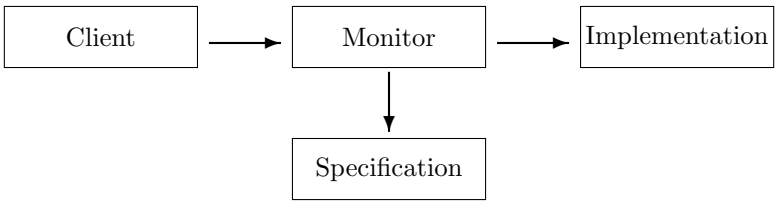

Figure 1: Logical Architecture for Runtime Verification

obscured by the dominance of pre- and post-conditions.

Model programs have rarely been used for run-time verification (see [9] for a discussion of a restricted form of model programs). But as long as model programs are deterministic and don't contain mandatory calls, they are also easy to check. In previous work [6] we presented a method for running restricted model programs and their corresponding implementations side by side, comparing the results at the method boundaries. But as soon as the model programs contain nondeterministic expressions or mandatory calls, we need a tighter integration.

External choice can still be resolved at the method boundaries; we check at runtime whether the implementation made a choice allowed by the specification. Internal choices can be resolved only by using an additional abstraction function. Abstraction functions link the state space of the implementation to that of the specification. Thus when an internal choice is made, first, the state space of the implementation is abstracted into the state space of the model program; next it is checked that the implementation made a choice allowed by the specification.

To check correct sequencing we replace any mandatory call in the specification with a check that the call occurred in the implementation. In addition, we check that any updates in the specification that should occur before a mandatory call are executed before the implementation makes the mandatory call.

To implement this checking our scheme takes advantage of the facilities provided on the .NET Platform [37] to perform intermediate-code rewriting. For each class, we introduce new methods and inner classes. In addition, we insert probes into the beginning and end of each method within the class. For component interaction, we manipulate a separately introduced runtime stack containing the call chain information needed to check the correct sequencing of method calls.

In our work the behavior of a class with contracts is the same as it is without: removing or adding a contract does not change the semantics. That is, a correct program will behave identically whether or not it is being monitored by a contract; but an incorrect program will trigger a runtime violation of the specification.

Overview. In Section 2, we provide an introduction to the relevant parts of the .NET Platform and its class library. Section 3 introduces our specifica-

tion technology for components and their interaction properties. The specifications are all written in our executable specification language, AsmL. Section 4 presents our method for implementing AsmL contracts in the .NET Framework. Section 5 discusses related work; Section 6 summarizes and describes future directions.

2 The .NET Platform

Microsoft's .NET Platform represents a new way to build distributed desktop and mobile applications for the Windows family of operating systems, and possibly non-Microsoft operating systems in the future.

From a programmer's point of view the .NET Platform can be understood as a new runtime environment and a common base class library. The runtime is referred to as the Common Language Runtime (CLR). Its primary role is to load, execute and manage .NET types. A .NET type contains code in the form of IL, an intermediate language that all .NET languages are compiled into. It is only at execution time that IL is compiled into native binary machine code. The CLR takes care of a number of low-level details such as memory management and language integration. The CLR also supports a simplified deployment of binary units, called assemblies. Assemblies contain .NET types. The CLR allows multiple versions of the same assembly to exist in parallel on a single machine. Assemblies are the components of the .NET world. We will specify the .NET types contained in an assembly.

Another building block of the .NET platform is the Common Type System (CTS). The CTS describes all possible types supported by the runtime, specifies how those types can interact with each other and how they are represented in the .NET metadata format. Metadata describes source level information like names or signatures. Metadata information is essential to link model programs to their implementations.

A language is called .NET aware if it uses a subset of the CTS called the Common Language Subset (CLS). If an assembly uses only the CLS, then it ensures that this assembly can be used seamlessly across all languages targeting the .NET platform. AsmL, our executable specification language, stays within the CLS. This guarantees that all .NET aware languages (of which there are currently more then 20, ranging from C♯ and VB to SML) are able to make use of contracts expressed in AsmL.

A more complete introduction to .NET can be found on the Microsoft web site [37].

The .NET Base Class Library: Collections. We illustrate our techniques through specifications for the .NET Framework Class Library, specifically, portions of the *System.Collections* namespace [38]. This namespace contains interfaces, classes, and structures that provide functionality for collections of elements.

The most basic property of a collection is being able to enumerate, in some order, the elements contained within it. Here is a partial interface for the interface *IEnumerator* (written here in C♯) [37]:

```
interface IEnumerator {
  object Current { get; };
  bool MoveNext();
  }
```

We leave C♯ modifiers implicit. For instance, our default access is "public"; we also assume that all methods can be overridden.

A collection does not necessarily allow modifications to its elements (once it has been constructed). However, if it does, then once it has "handed out" an enumerator, any modifications *invalidate* the enumerator; most further operations invoked on the enumerator throw an *InvalidOperationException* exception.

Clients use the method *MoveNext* to sequence through the elements; the value of the property *Current* takes on the value of each element in turn. A property is a function of zero arguments that syntactically appears as a simple variable. *Current* is a read-only property; so it can appear as an *r-value*, for instance on the right-hand side of an assignment statement. Writable properties, which are allowed to appear as *l-values*, that is on the left-hand side of an assignment are marked with set.

All collections can be queried for an enumerator. Other than that, collections are characterized by the number of elements they contain, a way to store all of the elements in an array and synchronization control for multi-threaded access. In the sequel we do not use these features; but we need the type. Thus we only show an empty interface *ICollection* :

```
interface IEnumerable {
  IEnumerator GetEnumerator();
  }
interface ICollection : IEnumerable { ... }
```

Extending this rudimentary functionality, there are two main types of collection: lists (sequences) and dictionaries, both allowing indexed access to the collection. In both cases, the indexing permits random access to individual elements. (The implication is that such access should be constant-time or almost constant-time. Such non-functional properties are beyond the scope of our specification methods.) The interfaces differ in the type of the indices: a list uses natural numbers as indices, while a dictionary allows the use of an arbitrary type. Again, we present only the parts of the *IDictionary* interface that we need for our presentation, namely the *Item* property:

```
interface IDictionary : ICollection {
  object this[ object key ] { get; set; }    }
```

The *Item* property allows the indexing of the collection with the familiar array access syntax, i.e., square brackets. In the rest of the paper, we focus on the *Item* property and the *GetEnumerator* method; they illustrate all of the features contained in all of the other properties and methods.

3 Contracts

In this section we use the specification language AsmL to specify components, progressing up the abstraction levels from code specification to component specification by providing examples. We defer an introduction to AsmL until Section 3.3. The first few specifications should be clear even without knowing the language.

3.1 Class Specifications

Assemblies can contain classes and interfaces. This subsection discusses how to specify classes; Section 3.2 discusses how to specify interfaces.

As our running example we use the class *Hashtable*, which is one of the library classes implementing the *IDictionary* interface. How could *Hashtable* have been specified? For an external client, the only assertions that are of interest are the pre-condition and post-condition of each method, along with any class invariant that each instance maintains.

Suppose that the class *Hashtable* implements the *IDictionary* interface via a pair of arrays: one to hold the keys and one to hold the values. This would be the corresponding C♯ code.

```
class Hashtable : IDictionary {
  protected object[ ] keys;
  protected object[ ] values;
  ... implementation of IDictionary methods ...
}
```

In such an implementation the clashing of hash values could be resolved by linear probing.

Class specifications don't abstract from the representation of the implementation. Figure 2 shows such a low-level specification using AsmL. AsmL uses a VB-like syntax.

The AsmL *Hashtable_Contract* class extends the C♯ class *Hashtable*. This allows AsmL to access its protected and public fields. Next follows the class invariant. AsmL's equality operator '=' denotes the application of the *Equals* method; *Equals* is inherited from the root type *System.Object*. Then the contracts for the set and get operations of the *Item* property are specified. Contracts must have the same signature as the method they specify. The keywords require and ensure introduce pre- and post-conditions, respectively. If one of them is not given it defaults to true. In our example pre-conditions are not given since most

```
class Hashtable_Contract extends Hashtable
  invariant
    (keys ≠ null and values ≠ null) ⇒
    let indices = {0..keys.Length − 1}
    keys.Length = values.Length and
    (forall i, j in indices
       holds (keys[i] ≠ null and keys[i] = keys[j]) ⇒ i = j)
  me(key as object) as object // Item property
    set ensure
       (key = null and thrown is ArgumentNullException) orelse
       (exists i in {0..keys'.Length − 1} holds keys'[i] = key and
                                                values'[i] = value and
                                                result = value)
    get ensure
        let indices = {0..keys.Length − 1}
        (key = null and thrown is ArgumentNullException) orelse
        (forall i in indices holds keys[i] ≠ key and result = null) or
        (exists i in indices holds keys[i] = key and result = values[i])
```

Figure 2: Class Contract for *Hashtable*.

library methods are total; they accept calls in any state. The post-condition uses result and thrown to refer to the result or exception returned by a method, respectively. Likewise it can use primed variables. A primed variable like x' denotes the (presumably updated) value of x in the state when the method terminates. The keyword value is an implicit parameter denoting the value on the right-hand side of the assignment statement when writing to a property. Note that we do not specify any *frame conditions*, for instance the post-condition for setting the *Item* property does *not* mention that all of the other keys and values have remained unchanged. One can certainly use AsmL to specify frame conditions, but they are costly to express and check.

This specification shows already several omissions of the pre-release documentation: for instance there is no provision in the post-condition of setting the *Item* property for the case that the key-value pair could not be installed into the dictionary because of some resource constraint, e.g., the two arrays could not be grown any larger. Another ambiguity is that the value of the *Item* property is supposed to be "the value associated with the key", but it doesn't say whether, if a key is already in the table, it is the old value that should be returned or the new value that is replacing it. Yet another ambiguity is which exception should be thrown when more than one can be!

Class contracts provide several benefits. They summarize the properties of the class so clients can understand them without having to access the source code. This immediately provides guidance for generating test cases. However, it

```
class IDictionary_Contract implements IDictionary
  var map as Map of object to object
  invariant null notin domain(map)
  me(key as object) as object // Item property
    set ensure
      (key = null and thrown is ArgumentNullException) orelse
      (map(key)' = value and result = value)
    get ensure
      (key = null and thrown is ArgumentNullException) orelse
      ((key notin domain(map) and result = null) or
        (key in domain(map) and result = map(key)))
```

Figure 3: Interface Contract for *IDictionary*.

restricts the level of abstraction at which the specifications can be expressed; all properties are in terms of the implementation data structures. One consequence is that the class invariant is long and fairly complicated: it must span the gap between the implementation level and the logical properties that the class encapsulates. Also, in component-oriented programming, methods are often defined to either have parameters of an interface type, or to return an interface type. When only class contracts are used, these methods cannot be specified. For components, class level specifications are often too low level.

3.2 Interface Specifications

Component-oriented programming is all about clients interacting with objects solely through interfaces. In this section, we describe *interface contracts*, i.e., contracts that describe the behavior of any class implementing the interface.

In current practice, interfaces list only signatures of the methods any implementing class must support. They don't have state variables. AsmL allows interface specification using "abstract" classes. For instance, in Figure 3, we specify the *IDictionary* interface as operating on a finite map of type object to object. Finite maps are part of AsmL's library of pure mathematical data structures. Using mathematical data structures lifts the specification level so that it is not dependent on the particular data structures any implementation happens to choose. The essential properties are much clearer and more comprehensible.

But where does the value of *map* come from? In this section we require that the values of interface variables are set only by evaluating an abstraction function. Abstraction functions translate an implementation-specific instance into an equivalent instance that lives in the abstract specification level of the interface.

Therefore, in order for the contracts to be checked, an implementation class must provide an abstraction function. It is executed before and after every

method call on the interface.

The abstraction function translates the (C♯) implementation state to the (AsmL) model state. To access the implementation state it must be written as part of the Hashtable contract.

```
class Hashtable_Contract extends Hashtable
  abstraction () as IDictionary_Contract
    return new IDictionary_Contract(
      {keys(i) ↦ values(i) |
          i in {0..keys.Length − 1} where keys(i) ≠ null} );
```

AsmL provides a default class constructor for *IDictionary_Contract* that takes a map as argument. Here the map is given in the form of a map-comprehension. Its value is a map where each *maplet* (domain-range pair) is constructed from parallel slots in the arrays *keys* and *values*. Since AsmL's mathematical data structures are .NET aware, this is no problem at all. When the class doesn't provide access to all of the necessary information, the abstraction function must be implemented within the class scope. This is possible, however we prefer to separate implementation and contract if possible.

3.3 Stand-alone Specifications

Up until now, contracts could be executed only in connection with some particular implementation. In addition, model variables required an abstraction function to link them to the implementation. In this section, we introduce contracts as model programs. There are several benefits for doing so:

- Model programs can be executed in isolation, that is, before the implementation exists.
- Model programs do not require an abstraction function, in fact sometimes it is not even possible to define an abstraction function.
- Model programs allow component interaction to be specified.

We write model programs in the Abstract State Machine Language (AsmL).

3.3.1 The Abstract State Machine Language

Abstract State Machines (ASMs) [22] provide the foundation for AsmL. We briefly review their semantics here. ASMs are based on transition systems; their states are first order algebras, that is, interpretations of a functional signature. The transition relation is specified by transition rules describing the modification from one state to the next, namely in the form of guarded updates, i.e., assignment statements that are executed if a boolean condition holds. A sequential run of an ASM program P is a finite or infinite sequence of states $S_0, S_1, \ldots$ where each S_i, $i > 0$, is obtained from S_{i-1} by executing the updates of P at

S_{i-1}. The updates generated in a particular step are called the *update set* for the step. For a wealth of ASM-related literature see the Michigan Website [28].

AsmL is Microsoft's ASM language. To deal with industrial applications AsmL extends ASMs with submachines, objects, exception handling, bounded genericity and semantic subtypes [23]. The first version of AsmL had native COM connectivity, the next release (to be available in Spring 2002) is .NET aware. AsmL is freely available for non-commercial research or teaching purposes from our web site [17]. It is currently used within Microsoft for the modeling, rapid prototyping, analysis, semi-automatic test-case generation, and checking of APIs, devices and protocols.

ASMs are a perfect fit for the operational specification of stateful components. Here is a simplified view of the correspondence: The class or interface fields are ASM functions. Method bodies describe guarded updates (i.e., statements); when executed they compute an update set. When the method terminates the update set is committed, i.e., a step is performed. A run is defined by a sequence of method calls. The next sections explore this in more detail.

3.3.2 Removing the Abstraction Function

AsmL specifications can use abstract statements to operate on the member variables of the interface to effect the results that any implementation is supposed to deliver. This allows us to get rid of the abstraction function.

Using our running example, a possible specification for *IDictionary* is shown in Figure 4. This example shows several features of AsmL that we haven't yet mentioned. AsmL is inherently parallel: to indicate sequential ordering one must use the keyword step. Within each step, all updates (assignment statements) are collected and transacted as one atomic transition. The keyword choose selects an arbitrary element from a collection which meets the associated predicate. When present, the ifnone clause acts as an else-construct: either the set was empty or the predicate failed on all members of the set. In either case, the ifnone clause is executed. When it is not present, the empty choice is equivalent to skip. The keyword forall indicates that the different calls to *e.Invalidate* will be performed in parallel. Thus, all of the calls to *Invalidate* are performed with the state of *map* the same, regardless of any actions an enumerator might take to assign a new value to it. So an intrinsic property of this specification is that every enumerator is updated with the collection holding the same value for *map*.

The update sets specified in the model programs say exactly what is modified and what stays the same. This is why we ignored the specification of frame conditions in the earlier examples.

What good is an interface contract that is not directly connected to a class? We already mentioned that they can be used for simulation and test. However we haven't yet mentioned other issues, for instance that of reuse of specifications. Reuse is simply done via calls. Usability issues are another reason to favor model programs. It is almost certainly easier to get working programmers to specify their systems using model programs than it is to get them comfortable with predicate calculus. Finally, model programs allow dealing with specifications for

```
class IDictionary_Contract implements IDictionary
  var map as Map of object to object
  var enums as Set of IEnumerator
  invariant null notin domain(map) and null notin enums

  me(key as object) as object
    set step if key = null
                 throw new ArgumentNullException()
             map(key) := value
         step forall e in enums
                 e.Invalidate()
         step return map(key)

    get if key = null
          throw new ArgumentNullException()
        else choose k in domain(map) where k = key
               return map(key)
             ifnone
               return null
```

Figure 4: Model Program for *IDictionary*.

which an abstraction function cannot even be given. For instance we recently modeled the ANSI Stream IO library. The "file open" call takes a file name and a file mode. The file mode specifies whether the file is opened for reading or writing. The passed file mode is important for the correct behavior of subsequent calls: for instance a client can only write onto a file that was successfully opened for writing. The library however does not provide accessors for file name and file mode. Thus there are implementations (in particular for embedded devices) that do not store file modes; therefore an abstraction function cannot be defined. For model programs this is no problem; model variables are updated in parallel with the running implementation.

Section 4.2.1 explores in more detail the role abstraction functions will play for runtime verification.

3.3.3 Introducing Nondeterminism

In general, we do not believe in components being nondeterministic; in most sequential systems nondeterminism is a bug. But specifications should be nondeterministic, either through nondeterministic data structures (such as unordered data types like sets) or through nondeterministic control structures. Nondeterministic specifications allow an implementer the freedom to make the appropriate time-space tradeoffs. Without nondeterminism, specifications degenerate into high-level prototypes (which may be useful, but are distinct from specifications).

We have already seen an example of AsmL's control nondeterminism, the choose operator in Figure 4. However, this particular choose statement is, in reality, deterministic: a key can occur at most once in the domain of a map. Enumerators in the .NET Class Library show the need for real nondeterminism. Every collection in the *Systems.Collection* namespace suports the interface *IEnumerable* which contains a single method: *GetEnumerator*. The method returns an interface which a client uses to sequentially retrieve elements from a collection. For instance, the method *GetEnumerator* in the *IDictionary* interface returns a value of type *IDictionaryEnumerator*. This enumerator returns elements of type *DictionaryEntry*. A specification of the behavior of *IDictionaryEnumerator* is shown in Figure 5. (The member variable *valid* and its use will be described in Section 3.3.4.)

```
struct DictionaryEntry
  key as object
  val as object
class IDictionaryEnumerator_Contract implements IDictionaryEnumerator
  var unvisited as Set of DictionaryEntry
  var current as DictionaryEntry = null
  var valid as Boolean = true

  Current as DictionaryEntry
    get if current = null
          throw new InvalidOperationException()
        else return current

  MoveNext() as Boolean
    if not valid
      throw new InvalidOperationException()
    else choose e in unvisited
          unvisited := unvisited - { e }
          current := e
          return true
        ifnone
          current := null
          return false

  Invalidate()
    valid := false
```

Figure 5: Model Program for *IDictionaryEnumerator*.

At the interface level, it doesn't matter in what order the elements are enumerated. Some classes that implement this interface are ordered types: their behavior is completely deterministic. However, for many classes, such as *Hashtable*, the decision is best left to the implementation; the latter's choice of data structures will dictate the most efficient way to perform the enumeration. Thus, at

the specification level, we use the choose operation to indicate the freedom that must be narrowed down by the implementation in the *MoveNext* method.

Section 4.2.2 will show how runtime verification can cope with checking nondeterministic model programs against an implementation.

3.3.4 Component Interaction

Pre- and post-conditions assume that methods are atomic units. However, there are properties involving component interaction that cannot be specified solely with conditions. The canonical example in the component literature is the subject-view design pattern [18]. In the subject-view pattern, a Subject is required to make a call to another component, a View, as part of its behavior. When the View is called, it is free to call back to the Subject, thus recursively re-entering the Subject and thereby exposing the Subject's intermediate state.

Although this example is simple, consisting of only a uni-directional call, our method can be used for arbitrary communication protocol specifications.

Model programs allow properties of component interaction to be specified: they can be executed in steps, which breaks up the atomicity of methods. But what is a reasonable step size? At which intermediate points can we synchronize model and implementation? Obviously these must be the calls to other interfaces; they can call back into the object and then model and implementation must agree. We define any method call to another interface (one different from that in which the call occurs) as a *mandatory call*. These are the call patterns that any implementation must be observed to make in order to faithfully implement its specification.

In our example, the dictionary is the Subject and any enumerator that has been returned from the method *GetEnumerator* is a View. As discussed in Section 2, .NET Class Library enumerators are read-only views on a collection; if the collection is modified during an enumeration, the enumerator becomes *invalid* and most further operations will throw an exception. This is why setting the *Item* property in Figure 4 calls the method *Invalidate* on each enumerator. (Unfortunately, *Invalidate* was not included in the actual .NET design of the interface.)

Here are some of the interaction properties that hold for the dynamic relationship between setting the *Item* property and *Invalidate*. (The corresponding AsmL specifications are given in Figure 4 and Figure 5).

1. Dictionaries update their internal state before notifying the enumerators of the change.

2. A dictionary calls *Invalidate* for each registered enumerator; the call is made whenever the dictionary is updated, even if the new value stored is identical to the old value. The order of the calls is implementation dependent.

3. Enumerators are synchronized with dictionaries. That is, all enumerators receive a notification with the dictionary in the same state. For instance,

if each enumerator calls back to the dictionary during the execution of its *Invalidate* method, all of the calls will see the same state. This is because the forall loop is a parallel loop.

In Section 4.3 we will see how to generate code so that these properties are checked for.

4 Runtime Verification

We have now explored all of the properties needed in a specification language to provide full behavioral specifications for components.

In this section we show how to instrument implementation classes to provide runtime verification. We describe our implementation in a parallel development to the previous section. Section 4.1 shows how to instrument classes with conditions. Next, Section 4.2 explains how to verify implementations against model programs. Finally, Section 4.3 presents the translation patterns for the verification of sequencing constraints.

We implement runtime verification by IL code rewriting. As outlined in Section 2, all .NET languages are compiled into assemblies. Assemblies contain .NET types; they contain IL and meta-data. The latter is meant to allow security checking, but it also provides us with important information about the types of the names (variables and methods) that are contained in the code. When the CLR loads a .NET type, its IL is compiled into native code. We intercept this process; we insert the IL of the contracts into the IL of the implementation and then give the resulting IL back to the CLR, which finally compiles it into executable code. This scheme allows us to immediately add contracts to any .NET language, without having to write or modify a parser for any one particular language. All that is needed is the ability to parse IL and restructure it.

For ease of reading we will use C♯ instead of IL to show the result of the code injection. As a simplifying assumption, we will treat exceptions as values that are only thrown, never returned. That way we can use a single object to hold either the value returned by the implementation or the exception thrown by it. We use the following notations in the transformed code:

- $[[s_1, s_2, \ldots, s_n]]$ represents the translation of the n statements (or expressions) from AsmL into IL. As part of the translation, we introduce new variables for holding the pre-values of expressions that are being compared to their post-values in the post-condition.
- $[[e]]v$ represents the translation of expression e from AsmL into IL, but with the value stored into the variable v.
- $s[s1/s2]$ represents the substitution of $s2$ for $s1$ in s. Substitution occurs within the same language, e.g., from AsmL code to AsmL code.

In the sequel we concentrate on how and where to inject probes. We do *not* show the extra code needed to copy the pre-values for expressions that are compared to their primed versions; such transformations are well known and used in many of the existing contract implementations (see [32] for a good explanation of the mechanism). Likewise we do *not* explain how to distribute conditions to check for behavioral subtyping constraints (for recent work in this area see [14]).

4.1 Runtime Verification of Conditions

We translate conditions into methods that are inserted into the transformed implementation class. The methods either terminate with no effect or else, when the implementation is incorrect relative to the specification, abort. The resulting methods have the appendix $\$Pre$, $\$Post$, and $\$Invariant$. We append $\$Checked$ to the name of the transformed class if it has checking code attached to it. We use a dollar sign in the name to stress that it is mechanically-generated code that we create and load into the CLR. However, remember that we are not creating a new type, but modifying the definition of the type before it has been loaded into the CLR. Thus, as far as clients are concerned, instances of the class are of the original implementation type.

We illustrate the transformation with the class contract for *Hashtable* from Section 3.1.

```
class Hashtable$Checked : IDictionary {
  void Hashtable$Invariant() { ASSERT([[invariant of Fig 2 ]]); }
  void set$Pre (object key, object val) { ASSERT(true); }
  void set$Post (object key, object val, object result) {
    ASSERT([[ensure of set in Fig. 2 ]]);
  }
  ... same for getting the Item property ...
}
```

ASSERT checks the value of its argument; if it is false, it throws a run-time violation exception; otherwise it just returns. Note that the implicit parameter value for setting the property is made explicit here. The parameter list for the pre-condition is the same as the method in the interface, but for the post-condition, one extra parameter has been added: *result*. It holds either the result computed by the implementation or the exception thrown by the implementation.

Now each method of the class is modified: return statements are replaced with assignments to a new local variable *result* (and an unconditional jump to the label *END*), the body of the method is wrapped in a try-catch block, and calls to the conditions are inserted. Figure 6 shows the resulting code. The translation scheme for interface contracts is exactly the same as that for class contracts, but an abstraction function is needed to link the implementation program to the interface contract. The abstraction function is used to repeatedly

```
class Hashtable$Checked : IDictionary {
  protected object[ ] keys; protected object[ ] values;

  object me[object key] { // Item property
    set {
        object result;
        Hashtable$Invariant();
        set$Pre(key, value);
        try {
          body of set from implementation [return e / result = e; break END; ]
        } catch (Exception e) {
          result = e;
        }
      END :
        set$Post(key, value, result);
        Hashtable$Invariant();
        if (result is Exception) throw result; else return result;
    }
  ...
}
```

Figure 6: *Hashtable* with Embedded Contract

create instances of the contract which are then checked in the conditions. Since abstraction functions are also needed for model programs, we concentrate on how *set*$*Pre* and *set*$*Post* differ for model programs and defer the discussion of declarative interface contracts until the end of Section 4.2.1. In the sequel, we use ellipses (...) for those code snippets that stay unchanged from the already presented examples.

4.2 Runtime Verification of Model Programs

In Figure 7, we show the translation for the model program from Figure 4. Note that the model program is translated into a class which is separate from the implementation class. Again, we use the dollar sign to indicate that this definition has been automatically generated.

The type *Map* is exactly the AsmL type; all of the AsmL types are implemented within the CLS. All of the statements from setting the *Item* property are put into *set*$*Post*. The return statement has been modified: it now assigns the model's return value to a newly introduced variable. The assertion at the end of the method checks to see that this variable is the same as the result from the implementation. Similarly, it tests that any exceptions thrown by the implementation are a subtype of the exceptions thrown by the model. What has been

```
class IDictionary$Checked {
  AsmL.Map map = new AsmL.Map();
  AsmL.Set enums = new AsmL.Set();
  void set$Pre (object key, object val) { ASSERT(true); }
  void set$Post (object key, object val, object implResult) {
    object modelResult;
    try {
      [[body of set in Fig. 4]] [return e / modelResult = e ]
    } catch (Exception e) { modelResult = e; }
    if (modelResult is Exception)
      ASSERT(implResult.GetType() is modelResult.GetType());
    else
      ASSERT(implResult == modelResult);
  }
  ... same for getting the Item property ...
}
```

Figure 7: *IDictionary* Model Interface Transformed into a Contract

accomplished is that the operational AsmL method has been translated into an assertion. (In AsmL, a return statement is restricted to be the last statement in a method, so we do not need to branch immediately after the newly introduced assignment statement.)

4.2.1 How to Keep Model State Up-to-Date?

Because the updates from the model program are executed as part of evaluating the $\$Post$ method, the state of the specification evolves alongside that of the implementation. However, runtime verification depends upon keeping the two states synchronized, at least from the perspective of any client that is utilizing the implementation's functionality.

Lockstep Execution. If the initial state of the model is equivalent to the initial state of the implementation, then we just add a new member variable to the implementation that holds the model instance. Then the only difference from Section 4.1 is that the calls are now based on a different instance variable (compare this code to that of Figure 6):

```
class Hashtable$Checked : IDictionary {
  IDictionary$Checked contract = new IDictionary$Checked();
  object me[object key] {
    set {
      ... contract.set$Pre(key, value);
      ...
      ... contract.set$Post(key, value, result); ...
    }
}
```

But there are two situations where such lockstep execution is not feasible. First, the initial state of the model may not match that of the implementation. Second, the implementation may make *silent transitions*, i.e., execute methods that change the implementation's state, but do not have a corresponding specification. We tackle these problems next.

Initial State Undetermined. Suppose the contents of the hashtable are persistent: i.e., when an instance of the hashtable is created, it initializes itself from an external source that is not part of the model. For instance, if the values stored in the hashtable are files, then the initial state depends on the state of the file system. In such a case, the contract doesn't change at all. Instead, the implementation is required to provide an initialization function, similar to the abstraction function from Section 3.2.

```
class Hashtable$Checked : IDictionary {
  IDictionary$Checked initialization() { ... }
}
```

The function must generate an instance of the model program; it is then called by code inserted into the constructor(s) of *Hashtable*.

```
class Hashtable$Checked : IDictionary {
  IDictionary$Checked contract;
  Hashtable(...) {
    ... rest of original constructor ...
    contract = initialization(); // inserted code
  }
}
```

Silent Transitions. Another problem occurs when an implementation provides methods that are not in the specified interface. These extra methods can change the state of the implementation without corresponding operations occuring in the model. In this case, we require an abstraction function from the implementation (as in Section 3.2) and call it if re-synchronization is required.

Calls to the conditions (here shown for the pre-condition only) are changed as follows:

$$(contract == null?contract \ = \ abstraction() : contract).set\$Pre(key, \mathsf{value});$$

Each extraneous method sets the value of *contract* to *null*. Let f be such a method.

```
class Hashtable$Checked : IDictionary {
  f() { // extra method
    contract = null; // inserted code
    ... rest of f without modification ...
  }
}
```

Note we have to potentially recalculate the abstraction before each call to the contract methods since methods executed in the interim could have voided the correspondence between the implementation and the specification.

Back to Conditions. Now we can revisit the interface specification from Figure 3. In that example, the specification was in terms of "abstract" types. To attach such a contract just means always re-creating the state of the specification via the abstraction function before checking any of the conditions. For example the call to the pre-condition is:

$$abstraction().set\$Pre(key, \mathsf{value}, result);$$

If the abstraction is always called, the field *contract* is not needed in the implementation class.

4.2.2 How to Resolve Nondeterminism?

As mentioned in Section 3.3.3, when the model programs make use of nondeterminism, then we must make sure that the choices made are angelic: if the behavior of the implementation can be matched, then that is the correct choice to make. Instead of performing an expensive search and the attendant problems of back-tracking, we enforce enough restrictions to solve the problem simply. We take advantage of the one-point rule [20]:

$$\exists(x \in S)(P(x) \wedge x = y) \quad \equiv \quad y \in S \wedge P(y)$$

We use a value from the implementation as the witness y. Ideally, we would like to be able to automatically identify the value, but when that is not possible, then we need a mechanism for the programmer to provide it.

Resolving External Choice. When the value being chosen is also being returned as the value of the method, then we can automatically resolve the

nondeterminism. Suppose we change the specification of setting the *Item* property to explicitly allow the implementation the choice of returning either the old value or the new value when a key already exists in the hashtable. The AsmL specification could be written as:

```
me(key as object) as object
  set if key = null
        throw ArgumentNullException
      else
        map(key) := value
        if key in domain(map)
          choose r in { map(key), value }
            return r
        else
          return value
```

The set being chosen from contains the old value that was already in the map and the value that will overwrite it. (The value of $map(key)$ in the set is the old value: AsmL's parallel semantics means that the assignment to $map(key)$ is not committed until the method terminates. If the method contains more than one step, then it is committed when the step terminates.)

The general substitution is to replace any occurrences of

```
choose x in S where p(x)
  R(x)
ifnone
  Q
```

with the following:

```
let x = result
if x in S and p(x)
  R(x)
else
  Q
```

and then, as before, any return statements are substituted by tests.

Resolving Internal Choice. When the nondeterminism cannot be automatically resolved, then we again fall back on forcing the implementer to provide an abstraction function. But it does not need to be a complete abstraction function; it needs only to provide enough information from the implementation to enable the choice to be made in the specification. For instance, in Figure 5, the method *MoveNext* must choose which element to set as the value of *current*, but the implementation returns a boolean value, not the value of the element. The only change to our substitution is that instead of binding the local variable

x to *result*, we instead bind it to the value returned by the provided abstraction function. For this simple example, the abstraction function can be defined as:

```
class DictionaryEnumerator : IDictionaryEnumerator {
  object MoveNext_Choose() { return this.Current; }
}
```

4.3 Runtime Verification of Mandatory Calls

As long as there are no mandatory calls, our translation scheme as presented has been able to maintain the fiction that the implementation and the specification can be executed atomically, one after the other. In order to resolve nondeterminism angelically, we executed the implementation first. However, when an implementation method f makes a mandatory call, the receiving component can call back into the implementation. That callback will, we assume, be to a method g with an attached contract. But the state of the model program will not be correct because the part of the model method for f will not have been executed yet.

Logically, what is required is for the AsmL specification to execute concurrently (i.e., in an interleaved manner) with the implementation; each mandatory call and the interface method boundaries are the synchronization points.

In order to implement this, we split the body of each model method into a set of blocks that can be executed piece by piece. We insert triggers into the implementation, both in the contracted method and in the mandatory calls, to cause each piece to be executed at the "correct time"; this is made precise in the following. When the implementation behaves according to its specification, then the combined effect is as if the model program had been executed as one atomic procedure call. An implementation that exhibits incorrect sequencing will throw a runtime exception, just as a failed condition does.

4.3.1 Splitting Bodies

The split body becomes an instance method on a new class, *Set$Checked*, shown in Figure 9, so that it can retain its state between invocations. It has member variables for the parameters of the call, for any local variables of the method, the return value from the mandatory call, and a reference to the contract of the contracted class. The new class implements an interface *ISteppable* which represents a method which can be suspended and resumed, just like a co-routine.

```
interface ISteppable {
  void Step();
  object result { set; }
  int pc { set; }
}
```

For example, consider setting the *Item* property as specified in Figure 4.

We call the method *Step* of the class *Set$Checked*, in place of calling *Pre* and *Post* as before. Note that the hashtable in Figure 8 still contains an instance of *IDictionary$Checked*; that instance still holds all of the member variables from the specification. The code of the set method is now encapsulated within *Set$Checked.Step* instead of *Post*. There is an instance of *Set$Checked* created for each invocation of setting the *Item* property since it must maintain the state of the "concurrent" model program; the CLR's runtime stack maintains the state of the implementation method.

```
class Hashtable$Checked : IDictionary {
 ... implementation member fields ...
 IDictionary$Checked dict$contract = new IDictionary$Checked();
 ... inner class definition of Fig. 9 ...
 object me[object key] {
  set {
   object result;
   Set$Checked set$Contract = new Set$Checked(dict$contract, key, value);
   set$Checked.Step(); // takes the place of Pre
   try {
    body of set from implementation [return e / result = e; goto END; ]
   } catch (Exception e) { result = e; }
 END :
   set$Contract.implResult = result; // replaces the parameter
   set$Contract.pc = END;
   set$Contract.Step(); // takes the place of Post
   if (result is Exception) throw result; else return result;
  }
 ...
```

Figure 8: Modified Implementation for set when it contains Mandatory Calls.

Figure 10 shows the generic template for *Step*, which acts as an interpreter for the model program. It has an outer loop that continually dispatches to the code for the current value of the program counter, *pc*. The very first "instruction" (case 0) checks any pre-condition from the specification (if present) and then pushes an empty frame onto a global stack, called the *mandatory call stack*, that we use to record the sequencing of the mandatory calls. It is executed by the first call to *Step* that is made in Figure 8 at the beginning of the set method. Each frame contains a set of objects; each object represents one mandatory call that must be made in the current step. (Remember that AsmL's parallel semantics means that all of the calls to *Invalidate* occur in the same step without any state changes happening in the model.)

The last instruction (case *END*) contains the final step and the check that used to be in the *Post* method. It also checks that there are no outstanding

```
class Hashtable$Checked.Set$Checked : ISteppable {
  object key; object val; IDictionary$Checked contract;
  int pc = 0; object implResult;
  Set$Checked(IDictionary$Checked contract, object key, object val) {
    this.contract = contract; this.key = key; this.val = val;
  }
  void Step() { ... see Fig. 10 for the general template ... }
}
```

Figure 9: New Inner Class *Set$Checked*.

mandatory calls to be made. This case is executed at the end of the set method as shown in Figure 8. Any non-determinism is angelically resolved using the result of the implementation, as shown in Section 4.2.2.

The structure of the code in Figure 10 is based on there being three "instructions" per step, i, of the original model program. The first instruction (case $3i + 1$) computes any updates that were in the corresponding step of the model program. It is very important to realize that any updates performed in this case are not committed: they are updates to AsmL variables. The *Step* method is a sequential implementation of the parallel AsmL model program, so an explicot commit is required to make any updates visible. Also, the AsmL model program has been transformed into a normal form where all steps are at the outermost level. Our example is already in that form. As mentioned earlier, all return statements are required to be in the last step, so the only substitutions for them are in case *END*. Furthermore, all variable references are prefixed with the instance *contract* because the members of the model program live in the contract object and not in each invocation of a contracted method.

The important difference is that mandatory calls are replaced with code that adds the information about the call (the callee and the parameters) into the set of mandatory calls in the current stack frame as an object of type *Call*. That is because the actual call is to be made by the implementation; the model program monitors the calls as shown in Section 4.3.2. The test on *NoOfCalls* is to see if there really are any mandatory calls in this step, if not, the program counter is incremented to jump around the handling of the mandatory call. But if there is a mandatory call, then the interpreter returns, suspending its execution. When it is resumed, it is because it is called (indirectly) by a mandatory call; we show this part of the scheme in Section 4.3.2. If any exceptions are thrown during the evaluation of case $3i + 1$, the program counter is set to *END* and the co-routine is suspended.

Mandatory calls come back to the interpreter twice: once before their body executes and once just before they terminate. The first call back to the interpreter (case $3i + 2$) evaluates the assertion of the strongest post-condition derived from the previous step in the model program. In our example of inval-

```
Step() {
 object modelResult;
 while (true) {
  switch (pc) {
   case 0 :     mandStack.PushEmptyFrame(); break;
   ...
   case 3i + 1 :
    try { [[ (step i
           [ var / contract.var ]
           [ e0.m(e1, ..., en) /
              mandStack.Add(new Call(this, e0, m, e1, ..., en))]
           ) ]];
           if (mandStack.NoOfCalls > 0) { pc += 1; return; } pc += 2; break;
         } catch (Exception e) {
           modelResult = e; pc = END; return;
         }
   case 3i + 2 : ASSERT(SP of step i − 1); pc += 1; return;
   case 3i + 3 : // Store result of the mandatory call
           if (mandStack.NoOfCalls > 1) { pc −= 1; return; }
           // Commit updates generated in case pc − 2
           pc += 1;
           if i + 1 is the last step return; else break;
   ...
   case END :
    ASSERT(mandStack.NoOfCalls() == 0);
    mandStack.PopFrame();
    // same as Fig. 7 but with the body
    // of the last step instead of the whole method
    return;
  } } }
```

Figure 10: Generic "Interpreter" for Contract Methods

idating enumerators, it is this check that makes sure the value of $map(key)$ is still the same, i.e., that the model state has not been incorrectly changed.

The second call back (case $3i+3$) would store the result from the mandatory call, if it returns a value. In our example, *Invalidate* does not return any value. Since there may be several mandatory calls in this same step, due to AsmL's parallel semantics, the current stack frame is inspected to see if there are any more calls to observe. (The epilogue code inserted into the mandatory call removes the *Call* object from the stack frame.) When there are more mandatory calls to be made during step i, then *pc* is decremented and control is returned to the mandatory call. If there are no more mandatory calls, then step i has been completed and its updates are committed. When step $i+1$ is the last step of the specification, then control is returned to the mandatory call. All other steps, however, execute *break* and continue on to begin the execution of the next step, setting up for whatever mandatory calls are made in those steps.

The need to execute the last step of the contract following the implementation conflicts with the lockstep synchronization used to monitor the mandatory calls. When the specification contains non-determinism that is to be resolved automatically, it can be accommodated only as long as it occurs in the final step and the final step does *not* contain any mandatory calls. If the last step does not contain any non-determinism, but does contain mandatory calls, then it can be translated into one of the pre-*END* triples of cases.

4.3.2 Tracking Calls

The object for each mandatory call that is put onto the mandatory call stack contains enough information for the mandatory call to recognize itself and to call back to the method that made the call. That is, it calls back to the model's method, in particular it causes the next step to occur in the piece-wise implementation of the model method.

```
class Call{
  ISteppable Caller;
  object Callee;
  object MethodReference;
  object[ ] MethodParameters;
}
```

Then, in each method that could potentially be a mandatory call, we insert two triggers that call back to the model. In our running example, the method *Invalidate* is a mandatory call that should be made when setting the *Item* property; its modified form is shown in Figure 11.

The method *SelectCall* looks through the set of calls in the current mandatory stack frame, and if an object is in the set where the last three members match the arguments to the call, returns a reference to the object. *Remove* just takes the call out of the current stack frame, but doesn't pop the mandatory stack. *Invalidate* cannot just unconditionally jump back. Our models are

```
class DictionaryEnumerator : IDictionaryEnumerator
  Dictionary dict;
  bool valid = true;
  DictionaryEnumerator(Dictionary d) { ... }
  Invalidate() {
    // EntryCode
    Call c = mandStack.SelectCall(this, "Invalidate", null);
    if (c != null) c.Caller.Step();
    try {
      // body
      [[valid := false]]
    } catch (Exeption e){ result = e; }
    finally {
      //ExitCode
      if (c != null) {
        // c.Caller.Result = ...; // if this method returned a value
        c.Caller.Step();
        mandStack.Remove(c);
    } } } }
```

Figure 11: Modified Code of a Mandatory Call

minimal models; an implementation may perform more externally visible communications than specified. We want to ensure only that it makes at least the calls that occur in the model program.

5 Related Work

The field of specification and component technology is vast. Here we concern ourselves only with work specifically in the area of attaching specifications as contracts to component-based software. We restrict ourselves to sequential systems: the issue of concurrency is yet another entire area.

Previously, we had used a more limited technique for attaching AsmL models to COM components [6], but it did not provide for nondeterministic specifications or for mandatory calls. We have also produced a more general description of using AsmL for component specification [5]. Other uses of AsmL are described in papers available from our web site [17].

Helm et al. [24, 27] were among the first to use model programs as contractual specifications, but do not present a method for the automatic conformance monitoring.

The Turku school has explored component specification in the context of the refinement calculus [2]; in particular Büchi and Weck [10] have proposed the use

of operational specifications to capture sequencing constraints. However they analyze the specifications statically, not at runtime. A case study of proving the correctness of Java Collections Frameworks [39] uses interface contracts with abstraction functions, but it does not address the issue of runtime checking for monitoring an implementation's conformance.

Almost all other work that we know of allows only *conditions*, i.e., contracts specified only as pre- and post-conditions and class/interface invariants.

As stated previously, the most well-known system for attaching contracts to components is Eiffel [36]. It allows conditions to be added to classes; conditions must be written in Eiffel as well. There are also plans to provide Eiffel contracts for .NET components. The intended implementation uses object wrappers to do so.

There are many systems for attaching contracts to Java components. Generally they provide for class specifications, not interface specifications. There are a mix of systems from commercial and academic sources. JMSAssert [35] monitors classes that have been annotated with conditions by creating methods from the conditions and calling them based on run-time interception. iContract [33] uses an assertion language that is compatible with OCL [29] and a source code pre-processor in order to attach the contracts. It can not handle any of the generic extensions to Java. Handshake [11] performs class modification at load time, but is limited to conditions. It intercepts the call from the JVM to the OS asking for a class file, instruments the file and returns the modified file to the JVM. So it does not modify the JVM itself or the class loader. The original functions are renamed inside of the modified class; the newly created methods check the conditions and call the original function, trapping the return value to compare against the specification. However, it doesn't catch any exceptions that are thrown within the original method. jContractor [32] uses a modified class loader to perform essentially the same functions as Handshake, but it does catch exceptions and allows for exception specification in addition to simple contracts. Jass [7] is a similar system, but adds enough bookkeeping so that it does not check assertions in methods that are called in conditions located in other methods. It also has just added trace assertions [15] as class invariants to specify valid method call sequences. Trace assertions are written separately from the conditions. They can express repetition, disjunction, and conjunction of traces and even control and data dependence on arbitrary predicates of the program state. Contract Java [14] tries to lift conditions to the interface level; they discuss how component-oriented programming prevents meaningful contracts when restricted to the class level since many parameters are of an interface type. But their interfaces do not have model variables and so the conditions are restricted to predicates on parameters. They also point out that many contract-checking systems do not correctly enforce behavioral subtyping. JISL, the Java Interface Specification Language [40], is mostly concerned with the specification of frame properties. JML [34], which is very similar to our work, can be used to check Java programs [8] via a source-to-source transformation. The current implementation allows only pre-conditions that do not contain quantifiers. Although the tool is limited, JML itself goes beyond simple contracts: it can be used to

specify interface contracts which can contain model variables and even model programs. But it tends to use model programs only for interaction properties and continues to use conditions for everything else.

There are also other systems for runtime verification that are not specifically targeted at Java. Edwards [12] uses specifications for components to generate wrapper components that check the pre- and post-conditions. An abstraction function is required because the conditions are expressed in terms of abstract values. But without model programs, synchronization properties cannot be specified. Soundarajan and Tyler [43] use trace variables in specifications to record method calls in order to reason incrementally about subtypes. Their trace variables are similar to our mandatory calls, but they also do not have model programs. Using a simple specification language, [4] instrument C programs to monitor certain temporal properties.

There are type systems that impose restrictions to address the frame problem [41], but in our opinion, they force too many programmer annotations to be feasible in our setting. It also doesn't directly address the issues of runtime verification for behavioral conformance.

Gannod and Cheng [19] explore the derivation of predicate-style specifications from program statements showing one way to unite declarative specifications and model programs.

The specification language B [1] is similar in many respects to AsmL, but while it is object-based, it is not object-oriented. Also it is targeted at static verification which limits its scalability. OCL [44], an industry-standard specification language used with UML, is restricted to conditions; it cannot be used to describe model programs.

6 Conclusions

Our work occurs within the context of specifying and checking software components. In particular, we are interested in the behavioral specification of interfaces. That means we require a mechanism for specifying the abstract behavior of any implementation of an interface; additionally, component interaction must be taken into account.

By using model programs, we are able to specify all of the traditional design-by-contract concepts of pre- and post-conditions and invariants. Model programs go further in that they can be used to specify properties of component interaction through the concept of mandatory calls. They also lend themselves to more uses than runtime verification of an implementation, although this paper has focused on that aspect. (See [21] for other uses of AsmL.)

The desirability of having higher-order data types and control structures in specifications leads us to propose the use of a new specification language, AsmL [17], which is freely available for non-commercial use from our web site. It contains the essential feature of nondeterministic choice, which allows specifications to prescribe behavior while still giving the implementor the freedom to choose efficient data structures and algorithms.

We have demonstrated a feasible method for performing runtime verification by using the facilities provided on the .NET Platform. Because our method operates at the level of IL, our specifications can be applied to components written in any .NET language. Under reasonable restrictions, and with limited human intervention, our method can cope with nondeterministic specifications. Using abstraction functions, the state correspondence between an implementation and its specification can be made arbitrarily precise. The more effort that a user puts into writing an abstraction function, the sooner any divergence is discovered. However, even without an abstraction function, our method detects any non-conformant behavior as soon as it is visible to a client of the implementation.

We have performed some pilot projects within Microsoft with earlier versions of our scheme. Within those scenarios, we did not find the need for any difficult abstraction functions; our method appears to be quite practical for industrial use.

The single most important missing aspect is concurrency. Our specifications do not explicitly deal with multi-threaded components. We would like to address this topic in the future.

We would like to thank the anonymous reviewers for their helpful comments.

References

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

[2] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[3] Thomas Ball and Sriram Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, February 2000. Available from `http://research.microsoft.com/pubs`.

[4] Thomas Ball and Sriram Rajamani. SLIC: A specification language for inteface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, January 2002. Available from `http://research.microsoft.com/pubs`.

[5] Mike Barnett and Wolfram Schulte. The ABCs of specification: AsmL, behavior, and components. *Informatica*, 25(4), Nov. 2001.

[6] Mike Barnett and Wolfram Schulte. Spying on components: A runtime verification technique. In Gary T. Leavens, Murali Sitaraman, and Dimitra Giannakopoulou, editors, *Workshop on Specification and Verification of Component-Based Systems*, Oct. 2001. Published as Iowa State Technical Report 01-09a.

[7] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass - Java with assertions. In Klaus Havelund and Grigore Rosu, editors, *Proceedings of the First Workshop on Runtime Verification (RV'01)*, volume 55 of Electronic Notes in Theoretical Computer Science. Elsevier Science, July 2001.

[8] Abhay Bhorkar. A run-time assertion checker for Java using JML. Technical Report 00-08, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, May 2000. Available by anonymous ftp from ftp.cs.iastate.edu or by e-mail from almanac@cs.iastate.edu.

[9] Manuel Blum and Hal Wasserman. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, November 1997.

[10] Martin Büchi and Wolfgang Weck. The greybox approach: When black-box specifications hide too much. Technical Report 297, Turku Centre for Computer Science, August 1999. Available from `www.tucs.abo.fi` at `/publications/techreports/TR297.html`.

[11] A. Duncan and U. Hölze. Adding contracts to Java with Handshake. Technical Report TRCS98-32, University of California at Santa Barbara, December 1998.

[12] Stephen H. Edwards. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, 11(2), 2001.

[13] David Evans, John Guttag, Jim Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Symposium on the Foundations of Software Engineering*. SIGSOFT, December 1994.

[14] Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *OOPSLA 2001*, pages 1–15. ACM SIGPLAN, September 2001.

[15] Clemens Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, University of Oldenburg, January 2000.

[16] R. W. Floyd. Assigning meanings to programs. *Proceedings Symposium on Applied Mathematics*, 19:19–31, 1967.

[17] Microsoft Research Foundations of Software Engineering, 2001. `http://research.microsoft.com/fse`.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[19] Gerald C. Gannod and Betty Cheng. Strongest postcondition semantics as the formal basis for reverse engineering. *Journal of Automated Software Engineering*, 3(1/2), 1996.

[20] D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math.* Springer-Verlag, Berlin, 1993.

[21] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Conformance testing with abstract state machines. Technical Report MSR-TR-2001-97, Microsoft Research, October 2001. Available from `http://research.microsoft.com/pubs`.

[22] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[23] Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Toward industrial strength abstract state machines. Technical Report MSR-TR-2001-98, Microsoft Research, October 2001. Available from `http://research.microsoft.com/pubs`.

[24] R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented system. *ACM SIGPLAN Notices*, 25(10):169–180, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[25] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[26] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1985.

[27] Ian M. Holland. Specifying reusable components using contracts. In Ole Lehrmann Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 287–308. Springer-Verlag, New York, NY, 1992.

[28] Jim Huggins. Abstract State Machines, 2001. `http://www.eecs.umich.edu/gasm`.

[29] IBM. Object constraint language, 2001. `http://www-4.ibm.com/software/ad/library/standards/ocl.html`.

[30] Isabelle. The Isabelle theorem prover, 2001. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`.

[31] Raymie Stata K. Rustan M. Leino, James B. Saxe. Checking Java programs via guarded commands. Technical Note 1999-002, Compaq Systems Research Center, Palo Alto, CA, May 1999.

[32] Murat Karaorman, Urs Holzle, and John Bruno. jContractor: A reflective Java library to support design by contract. Technical Report TRCS98-31, University of California, Santa Barbara. Computer Science., January 19, 1999.

[33] Reto Kramer. iContract—the Java Designs by Contract tool. In *Proc. Technology of Object-Oriented Languages and Systems, TOOLS 26, Santa Barbara/CA, USA*. IEEE CS Press, Los Alamitos, 1998.

[34] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06o, Iowa State University, Department of Computer Science, May 2000. See `www.cs.iastate.edu/~leavens/JML.html`.

[35] Man Machine Systems. JMSAssert, 2002. Available from `http://www.mmsindia.com/JMSAssert.html`.

[36] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[37] Microsoft Corporation. .NET documentation, 2001. `http://www.microsoft.com/net/`.

[38] Microsoft Corporation. System.Collections documentation, 2001. Available at `http://msdn.microsoft.com/library` at `/en-us/cpref/html/frlrfsystemcollections.asp`.

[39] Anna Mikhajlova and Emil Sekerinski. Ensuring correctness of Java Frameworks: A formal look at JCF. Technical Report TUCS-TR-250, TUCS - Turku Centre for Computer Science, March 1999.

[40] P. Müller, J. Meyer, and A. Poetzsch-Heffter. Making executable interface specifications more expressive. In C. H. Cap, editor, *JIT '99 Java-Informations-Tage 1999*, Informatik Aktuell. Springer-Verlag, 1999. Available from `http://www.informatik.fernuni-hagen.de` at `/pi5/publications.html`.

[41] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in jml. Technical Report 01-03, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, April 2001. To appear in the Formal Techniques for Java Programs 2001 workshop at ECOOP 2001. Also available from archives.cs.iastate.edu.

[42] PVS. The PVS specification and verification system, 2001. `http://pvs.csl.sri.com/`.

[43] Neelam Soundarajan and Benjamin Tyler. Testing components. In *Workshop on Specification and Verification of Component-Based Systems, OOPSLA 2001*, pages 1–6. Published as Iowa State Technical Report #01-09a, October 2001.

[44] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley Longman, Reading, Mass., 1999.