ELSEVIER

The Journal of Systems and Software 65 (2003) 199–208

The Journal of Systems and Software

www.elsevier.com/locate/jss

Runtime verification of .NET contracts

Mike Barnett *, Wolfram Schulte

Microsoft Research, One Microsoft Way, Redmond, WA 98052-6399, USA

Received 17 March 2001; received in revised form 1 March 2002; accepted 19 April 2002

Abstract

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++.

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.

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. Ideally, this would be done statically, but this is often not feasible. Therefore we propose to do it dynamically; this limits the guarantees, but allows unlimited flexibility and scalability.

Logically, the effect is shown in Fig. 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 obscured by the dominance of pre- and post-conditions.

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, 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 (Gamma et al., 1995) 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*

* Corresponding author. Tel.: +1-425-703-5849; fax: +1-425-936-7329.

E-mail addresses: mbarnett@microsoft.com (M. Barnett), schulte@microsoft.com (W. Schulte).

0164-1212/03/$ - see front matter © 2002 Elsevier Science Inc. All rights reserved.
doi:10.1016/S0164-1212(02)00041-9

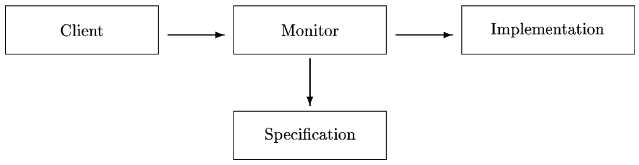

Fig. 1. Logical architecture for runtime verification.

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.

Nondeterministic contracts allow an 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 nondeterminism are two of the outstanding challenges in the runtime verification of model programs. In this paper, we address the latter; a full treatment including the former is described in Barnett and Schulte (2002).

Previously, contracts in the form of pre- and post-conditions have been injected into the class or method to which they apply; we discuss this work in Section 5. Model programs have rarely been used for run-time verification (see Blum and Wasserman (1997) for a discussion of a restricted form of model programs). But as long as model programs are deterministic and do not contain mandatory calls, they are also easy to check. In previous work (Barnett and Schulte, 2001b) 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 (Microsoft Corporation, 2001) 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 specification technology for components and their interaction properties. The specifications are 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. In an accompanying technical report (Barnett and Schulte, 2002), we discuss runtime verification in the general context of software verification.

2. The .NET platform

Here we briefly introduce the platform; a more complete introduction to .NET can be found on the Microsoft web site (Microsoft Corporation, 2001).

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. Assemblies also contain metadata that 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 its types are implemented within the common language subset (CLS). Using the CLS ensures that an 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.

2.1. 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 (Microsoft Corporation, 2001). 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.

All collections can be queried for an enumerator, i.e., they all support the method *GetEnumerator* which returns a reference to the interface *IEnumerator*. 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 multithreaded access.

A collection does not necessarily allow modifications to its elements (once it has been constructed). However, if it does, then once it has executed *GetEnumerator*, any modifications *invalidate* all existing enumerators; most further operations invoked on those enumerators throw an *InvalidOperationException* exception.

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 nonfunctional 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. We present only the parts of the *IDictionary* interface that we need for our presentation, namely the *Item* property. Here we show it in C♯:

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

We leave C♯ modifiers implicit. For instance, our default access is "public"; we also assume that all methods can be overridden. A property is a class member that allows actions to be associated with the reading or writing of the member. Since the *Item* property is marked as both `get` and `set`, it can appear on either side of the assignment operator. It 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 method associated with setting the *Item* property.

3. Contracts

In this section we briefly introduce the specification language AsmL and show its use for specifying components. [1] While AsmL contains a declarative subset, we use its operational character to specify a component by a *model program*. There are several benefits to writing a model program instead of simple pre- and post-conditions:

- Model programs can be executed in isolation, for instance, before the implementation even exists.
- Model programs at the interface level 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.

3.1. The Abstract State Machine Language

Abstract State Machines (ASMs) (Gurevich, 1995) 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 (Huggins, 2001).

AsmL is Microsoft's ASM language. To deal with industrial applications AsmL extends ASMs with submachines, objects, exception handling, bounded genericity and semantic subtypes (Gurevich et al., 2001). The first version of AsmL had native COM connectivity, the current release is .NET aware. AsmL is freely available for noncommercial research or teaching purposes from

[1] Complete documentation can be found on our web site (Foundations of Software Engineering, 2001).

our web site. 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.

3.2. Specifying a component with AsmL

AsmL specifications use abstract statements to operate on the member variables of the interface to effect the results that any implementation is supposed to deliver.

Using our running example, a possible specification for *IDictionary* is shown in Fig. 2.

This example shows several features of AsmL that we have not 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 `forall` indicates that the different calls to *e.Invalidate* will be performed in parallel.

The update sets specified in the model programs say exactly what is modified and what stays the same; we do not explicitly specify frame conditions. We define any method call to another interface (one different from that in which the call occurs) as a *mandatory call*. In order to be faithful to its specification, an implementation must make (at least) all of the mandatory calls contained in its specification. In our example, the calls to *Invalidate* are mandatory calls.

With respect to the subject-view design pattern (Gamma et al., 1995), the dictionary is the Subject and any enumerator that has been returned from the method

```
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)
```

Fig. 2. Model program for *IDictionary*.

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 Fig. 2 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* in Fig. 2:

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 4.2, we show how to generate code so that these properties are checked for. Although this example is simple, consisting of only a uni-directional call, our method can be used for arbitrary communication protocol specifications.

4. Runtime verification

In this section we show how to instrument implementation classes to provide runtime verification. We describe our implementation using the example from the previous section. Section 4.1 explains the simple case of model programs without mandatory calls. Section 4.2 then presents the more complicated translation pattern for the verification of sequencing constraints.

We implement runtime verification by IL code rewriting. 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.

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.
- $s[s1/s2]$ 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 post-values; such transformations are well known and used in many of the existing contract implementations (see Karaorman et al. (1999) 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 Findler and Felleisen (2001)).

4.1. Runtime verification of model programs

Fig. 3 shows the translation for the model program from Fig. 2. Note that the model program is translated into a class which is separate from the implementation class. We use dollar signs in the names to indicate that the definitions have 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 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.)

```
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. 2]] [return e / modelResult = e ]
    } catch (Exception e) { modelResult = e; }
    if (modelResult is Exception)
      ASSERT(implResult.GetType() is modelResult.GetType());
    else ASSERT(implResult == modelResult);
  }
}
```

Fig. 3. *IDictionary* interface model transformed into a contract.

Given the transformed model program, we instrument the code for any class that implements the interface *IDictionary*. All of the constructors are modified to create an instance of *IDictionary*$*Checked*. Each of the methods, f, for which there are corresponding specifications, are modified to call $f$$*Pre* upon entry and $f$$*Post* before returning. The bodies of the methods are inserted into a try/catch block and any return statements are changed to assignment statements just as shown for the model program in Fig. 3. Space reasons prevent us from discussing how to ensure that the initial state of the model program corresponds to the initial state of the implementation, or how to re-synchronize in the presence of uncontracted methods; full details can be found in the accompanying technical report (Barnett and Schulte, 2002).

4.2. Runtime verification of mandatory calls

As long as there are no mandatory calls, we can use the translation scheme outlined in Section 4.1. It executes the implementation and the specification, atomically, one after the other. In order to resolve nondeterminism angelically, we execute the implementation first; its state is then available for choosing the "right" path in the model.

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.

4.2.1. Splitting bodies

The split body becomes an instance method on a new class, *Set$Checked*, shown in Fig. 5, 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 the class *Hashtable* from the System.Collections namespace in the .NET Framework, one of the many classes that implements the interface *IDictionary*. We transform its definition into the one shown in Fig. 4 (we show only the setting of the *Item* property). Instead of calling *Pre* and *Post* as described in Section 4.1, calls to the method *Step* of the class *Set$Checked* are inserted instead. Note that it still contains an instance of *IDictionary$Checked*; that instance still holds all of the member variables from the specification. The code of the set method from Fig. 2 is now encapsulated within *Set$Checked.Step* instead of *Post*. An instance of *Set$Checked* is created for each invocation of setting the *Item* property since it must maintain

```
class Hashtable$Checked : IDictionary {
 ... implementation member fields ...
 IDictionary$Checked dict$contract = new IDictionary$Checked();
 ... inner class definition of Fig. 5 ...
 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;
  }
 ...
```

Fig. 4. Modified implementation for `set`.

```
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. 6 for the general template ... }
}
```

Fig. 5. New inner class *Set$Checked*.

the state of the "concurrent" model program; the CLR's runtime stack maintains the state of the implementation method.

Fig. 6 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 Fig. 4 at the beginning of the set method.

```
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. 3 but with the body
    // of the last step instead of the whole method
    return;
  } } }
```

Fig. 6. Generic "Interpreter" for contract methods.

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 mandatory calls to be made. This case is executed at the end of the set method as shown in Fig. 4. Any nondeterminism is angelically resolved using the result of the implementation (see Barnett and Schulte (2002) for a full treatment of nondeterminism).

The structure of the code in Fig. 6 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 explicit 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 4.2.2. 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 4.2.2. 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 callback to the interpreter (case $3i + 2$) evaluates the assertion of the strongest postcondition derived from the previous step in the model program. In our example of invalidating 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 callback (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 nondeterminism 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 nondeterminism, but does contain mandatory calls, then it can be translated into one of the pre-*END* triples of cases.

4.2.2. 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 Fig. 7 which would be part of the specification for the *IEnumerator* interface.

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,

```
void Invalidate() {
  object result;
  // EntryCode
  Call c = mandStack.SelectCall(this, "Invalidate", null);
  if (c != null) c.Caller.Step();
  try {
    body of implementation [return e / result = e; goto END; ]
  } catch (Exeption e){ result = e; }
END :
  //ExitCode
  if (c != null) {
    // c.Caller.Result = result; // if this method returned a value
    c.Caller.Step();
    mandStack.Remove(c);
  } }
```

Fig. 7. Modified code of a mandatory call.

but does not pop the mandatory stack. *Invalidate* cannot just unconditionally jump back. Our models are *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 (Barnett and Schulte, 2001b), 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 (Barnett and Schulte, 2001a). Other uses of AsmL are described in papers available from our web site (Foundations of Software Engineering, 2001).

Helm et al. (1990) and Holland (1992) 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 (Back and von Wright, 1998); in particular Büchi and Weck (1999) 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 (Mikhajlova and Sekerinski, 1999) 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.

Arguably, the most well-known system for attaching contracts to components is Eiffel (Meyer, 1992). 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 (Man Machine Systems, 2002) monitors classes that have been annotated with conditions by creating methods from the conditions and calling them based on run-time interception. iContract (Kramer, 1998) uses an assertion language that is compatible with OCL (IBM, 2001) and a source code pre-processor in order to attach the contracts. It can not handle any of the generic extensions to Java. Handshake (Duncan and Hölze, 1998) 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 does not catch any exceptions that are thrown within the original method. jContractor (Karaorman et al., 1999) 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 (Bartetzko et al., 2001) 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 (Fischer, 2000) 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 (Findler and Felleisen, 2001) 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 (Müller et al., 1999), is mostly concerned with the specification of frame properties. JML (Leavens

et al., 2000), which is very similar to our work, can be used to check Java programs (Bhorkar, 2000) 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 (2001) 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 (2001) 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, (Ball and Rajamani, 2002) instrument C programs to monitor certain temporal properties.

There are type systems that impose restrictions to address the frame problem (see for example Müller et al. (2001)), but in our opinion, they force too many programmer annotations to be feasible in our setting. It also does not directly address the issues of runtime verification for behavioral conformance.

Gannod and Cheng (1996) explore the derivation of predicate-style specifications from program statements showing one way to unite declarative specifications and model programs.

The specification language B (Abrial, 1996) 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 (Warmer and Kleppe, 1999), 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 Grieskamp et al. (2001) 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, which is freely available for noncommercial 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 nonconformant 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.

Acknowledgement

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

References

Abrial, J.-R., 1996. The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge, MA.

Back, R.-J., von Wright, J., 1998. Refinement Calculus: A Systematic Introduction. Springer-Verlag, Berlin.

Ball, T., Rajamani, S., 2002. SLIC: A specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research. Available from http://research.microsoft.com/pubs.

Barnett, M., Schulte, W., 2001a. The ABCs of specification: AsmL, behavior, and components. Informatica 25 (4), 517–526.

Barnett, M., Schulte, W., 2001b. Spying on components: A runtime verification technique. In Leavens, G.T., Sitaraman, M., Giannakopoulou, D., (Eds.), Workshop on Specification and Verification of Component-Based Systems. Published as Iowa State Technical Report 01-09a.

Barnett, M., Schulte, W., 2002. Contracts, components and their runtime verification on the .NET platform. Technical Report MSR-TR-2002-38, Microsoft Research. Available from http://research.microsoft.com/pubs.

Bartetzko, D., Fischer, C., Möller, M., Wehrheim, H., 2001. Jass—Java with assertions. In Havelund, K., Rosu, G., (Eds.), Proceedings of the First Workshop on Runtime Verification (RV'01), vol. 55 of Electronic Notes in Theoretical Computer Science. Elsevier Science.

Bhorkar, A., 2000. 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. Available by anonymous ftp from ftp.cs.iastate.edu or by e-mail from almanac@cs.iastate.edu.

Blum, M., Wasserman, H., 1997. Software reliability via run-time result-checking. Journal of the ACM 44 (6), 826–849.

Büchi, M., Weck, W., (1999). The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Centre for Computer Science. Available from www.tucs.abo.fi at /publications/techreports/TR297.html.

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

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

Findler, R.B., Felleisen, M., 2001. Contract soundness for object-oriented languages. In OOPSLA 2001, ACM SIGPLAN, p. 1–15.

Fischer, C., 2000. Combination and Implementation of Processes and Data: from CSP-OZ to Java. Ph.D. thesis, University of Oldenburg.

Foundations of Software Engineering, M.R., (2001). http://research.microsoft.com/fse.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Mass.

Gannod, G.C., Cheng, B., 1996. Strongest postcondition semantics as the formal basis for reverse engineering. Journal of Automated Software Engineering 3 (1/2), 139–164.

Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M., 2001. Conformance testing with abstract state machines. Technical Report MSR-TR-2001-97, Microsoft Research. Available from http://research.microsoft.com/pubs.

Gurevich, Y., 1995. Evolving Algebras 1993: Lipari Guide. In: Börger, E. (Ed.), Specification and Validation Methods. Oxford University Press, Oxford, pp. 9–36.

Gurevich, Y., Schulte, W., Veanes, M., 2001. Toward industrial strength abstract state machines. Technical Report MSR-TR-2001-98, Microsoft Research. Available from http://research.microsoft.com/pubs.

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

Holland, I.M., 1992. Specifying reusable components using contracts. In Madsen, O.L., (Ed.), ECOOP '92, European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, vol. 615. Utrecht, The Netherlands, Springer-Verlag, New York, NY, pp. 287–308.

Huggins, J., 2001. Abstract State Machines. http://www.eecs.umich.edu/gasm.

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

Karaorman, M., Holzle, U., Bruno, J., 1999. jContractor: A reflective Java library to support design by contract. Technical Report TRCS98-31, University of California, Santa Barbara. Computer Science.

Kramer, R., 1998. 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.

Leavens, G.T., Baker, A.L., Ruby, C., 2000. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06o, Iowa State University, Department of Computer Science. See www.cs.iastate.edu/~leavens/JML.html.

Man Machine Systems 2002. JMSAssert. Available from http://www.mmsindia.com/JMSAssert.html.

Meyer, B., 1992. Eiffel: The Language. Object-Oriented Series. Prentice Hall, New York, NY.

Microsoft Corporation 2001. NET documentation. http://www.microsoft.com/net/.

Mikhajlova, A., Sekerinski, E., 1999. Ensuring correctness of Java Frameworks: A formal look at JCF. Technical Report TUCS-TR-250, TUCS - Turku Centre for Computer Science.

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

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

Soundarajan, N., Tyler, B., 2001. Testing components. In Workshop on Specification and Verification of Component-Based Systems, OOPSLA 2001, Published as Iowa State Technical Report #01-09a, pp. 1–6.

Warmer, J., Kleppe, A., 1999. The Object Constraint Language: Precise Modeling with UML. Addison-Wesley Longman, Reading, Mass.

Michael Barnett received his Ph.D. from the University of Texas at Austin in 1992. He was an Assistant Professor in the Computer Science Department at the University of Idaho from 1992 to 1995. Since 1995 he has been a Research Software Design Engineer at Microsoft Research, first in the Natural Language Processing group and currently with the Foundations of Software Engineering group.

Wolfram Schulte is a researcher at Microsoft Corporation, Redmond, since 1999. He received his Masters (1987) and Ph.D. (1992) in Computer Science from Technical University of Berlin. He has been a software engineer for Software Design and Management (1992–1993), and was Assistant Professor of Computer Science at University of Ulm (1993–1999) where he also got his Habilitation (2000). He worked for and collaborated with several large corporations, including Daimler–Chrysler and Rhode und Schwarz. He has broad interests in several areas of software construction, including development methods, specification and programming languages, automatic analysis of designs and implementations, and automatic testing.