Conformance Checking of Components Against Their Non-deterministic Specifications

Mike Barnett Lev Nachmanson
Wolfram Schulte
{mbarnett,levnach,schulte}@microsoft.com

June 2001

Technical Report
MSR-TR-2001-56

Conformance checking of a component is a testing method to see if an implementation and its executable specification are behaviorally equivalent relative to any interactions performed on the implementation. Such checking is complicated by the presence of non-determinism in the specification: the specification may permit a set of possible behaviors. We present a new method to automatically derive a component that manages all of the angelic non-determinism for an arbitrary implementation/specification pair. The new component just plugs in; no instrumentation of any implementation is necessary. Conformance checking thus helps to keep high-level non-determinstic specifications of components and their low-level implementations in sync.

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

1 Introduction

> There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies and the other is to make it so complicated that there are no obvious deficiencies. – C.A.R. Hoare

Software always needs good specifications. A good specification allows software to be understood, in fact, we claim that a software component is only as good as its specification. Unless clients of components understand how to use them, component-oriented programming will never fulfill its promise. We provide tools to create human-understandable, yet nevertheless executable specifications for software components. Using our specification language, AsmL, based on the theory of Abstract State Machines, we have built models of real-world components, like intelligent devices, internet protocols, debuggers and network components.

The central fact about code is that it evolves, and for an executable specification to be useful, it too must be capable of evolving. Indeed, it is the static nature of current specification techniques — primarily human- language descriptions — that leads to the quick obsolescence of such specifications. Given an executable specification, we provide a system that executes the specification in parallel with the real implementation and signals an error whenever they do not agree. This checking is done at the boundary of each interface method; it allows immediate notification that the specification and implementation have diverged. The decision as to which is "right" is something that must be left to the human, but at least they have the automated support for detecting the discord. We term this automatic checking of an implementation against its specification *conformance checking*. Conformance checking of determinstic specifications is known, see Section 5 for related work.

The major complication of conformance checking is *non-determinim in the specification*. Non-determinsm allows the specifier to leave design decisions open; it's up to the implementer to provide detailed data structures and algorithms. Another reason that specifications might be non-determinstic is persistence. The specification must be written relative to an initial state that is not precisely known. Component specifications typically expose both kinds of non-determinism.

The contribution of this paper is that it gives a *detailed, automatizable method* for checking the conformance of components and their specifications in the presence of non-determinism. The central idea is that when running the implementation against the specification, the specification can be used as a test oracle — in fact this test oracle becomes a separate component, a proxy. The proxy maintains a table of correspondances between specification and implementation objects. If a method is called on the proxy, it forwards the calls to the implementation and the specification.

- If both calls return values of primitive type, the result of the implementation can immediately be tested for conformance with the specification.

- If both calls return references to deterministically computed objects, their correspondence is stored; the check whether they conform is postponed. Non-conformance can be found only when calls of the first kind occur on those objects as execution proceeds.

- If the specification non-determinstically returns an object, the correspondence that is stored is between the implementation object and the range of possible specification objects; again the conformance check must deferred until later, when enough evidence has been accumulated to determine the correspondence of the implementation object with one of the possible specification objects.

In previous work, we have constructed a system for automatically checking a COM implementation against a determinstic AsmL specification. It has been used to check real-world industrial components [2]. We curently have a prototype for checking non-determinstic specifications. Although our system is implemented for COM components, it applies to any component technology that uses dynamic linking.

The paper is organized as follows. Section 2 gives an overview of the notation we use. Then in Section 3 we explain how to use a proxy to check the implementation against its determinstic specification. Section 4 presents the extensions needed for checking non-determinstic specifications. An overview of similar approaches is discussed in Section 5 and Section 6 summarizes and presents limitations and future work.

2 Notation

We write executable specifications of components in the Abstract State Machine Language (AsmL). The language is based on the theory of Abstract State Machines [7]. It is currently used within Microsoft for modeling, rapid prototyping, analyzing and checking of APIs, devices and protocols.

The key aspects which distinguish AsmL from other related specification languages are:

- it is executable,

- it uses the ASM approach for dealing with state,

- it has a full-fledged object and component system,

- it supports writing non-deterministic specifications.

See `http://www.research.microsoft.com/fse` for a detailed description.

Because AsmL has native COM (and .NET) connectivity, one can not only specify components in AsmL and simulate them but also substitute low-level implementations by high- level specifications. This substition allows heterogeneous systems to be built, partly developed using standard programming languages

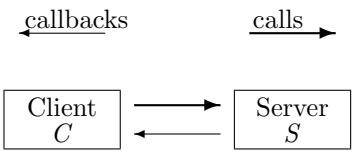

Figure 1: A client-server architecture

and partly using executable specifications. It is also crucial for implementing conformance checking without the need for instrumenting the implementation.

Non-determinism is one of the key features of AsmL, that allows designers to leave room for implementations decisions. However in AsmL non-determinsm is restricted, you can choose or quantify only over bounded sets [4].

In the sequel we use AsmL as the notation for specifications as well as for imlementations, although the latter are generally written in languages such as C++. Note that all of the formal descriptions are executable as is.

3 Deterministic Specifications

First, we explain our implementation of conformance execution in the context of deterministic specifications. In Section 4, we extend our method for non-deterministic specifications. We start with a system whose architecture is shown in Figure 1. It comprises a client, C, and a server, S, that the client accesses.

Both are *components*: their functionality can be accessed only by calling *methods* (i.e., functions) that are grouped into sets called *interfaces*. In addition, each method call is attached to a specific object reference which is an implicit argument (*this* in C++, or `me` in AsmL e.g.) to the method.

The internal details of the implementation of an object are never known, or needed, only that the object provides responses via the methods of the interface that it supports (which may involve returning a reference to another object that supports a different interface). However, we rely on the fact that the object references are stable and can be used as identifiers that can be tested for equality.

In the following we assume that we have not only implementations of the client and the server but also an executable model, M, of the server. M is also a component; it provides the same interfaces as S. M step-for-step simulates the behavior of S at the method level of granularity. Conformance checking means that from the client's point of view, the observational behavior of the model is indistinguishable from that of the server, i.e., they are *observationally equivalent*.

```
interface ICanvas
  createFigure(...) as IFigure
  createInternalFigure(...)
  minimized() as Boolean
  getFigEnum() as IEnumFigure

interface IFigure
  getColor() as Color
  setColor(c as Color)
  getBorder() as IBorder

interface IBorder
  getWidth() as Integer
  setWidth(i as Integer)

interface IEnumFigure
  Next() as IFigure
```

Figure 2: Example Interfaces

3.1 Example

Figure 2 presents a small example that we use throughout. It is typical for COM interfaces, but is not COM-specific. The example provides interfaces for a component-oriented drawing program: a client interacts with a root interface, *ICanvas*, to create and manipulate geometric figures, which support the interface *IFigure*.

There are two ways to create figures. The method *createFigure* returns a reference to the *IFigure* interface on the figure that is created. Alternatively, the method *createInternalFigure* creates a figure, but returns nothing to the client. When the latter method is used, the client retrieves figures by first getting a reference to the *IEnumFigure* interface from the *ICanvas* interface (by calling *getFigEnum*), and then using the *Next* method from the returned interface to access individual figures. Each figure has a nested object, a border, which supports the interface *IBorder*.

To make our method easier to explain, we assume that each method either reads the state of the server and returns a value or writes a value into the state of the server, but not both. (The method *createFigure* is a special case.) We call the former methods observers, the latter ones are called modifiers. Of course, in reality, a method can have an arbitrary number of outputs and may both read and write values; we support this in our implementation. We leave implicit the fact that COM methods also return a status value in addition to whatever other values they return.

Note that the interface definitions allow us to distinguish between data values

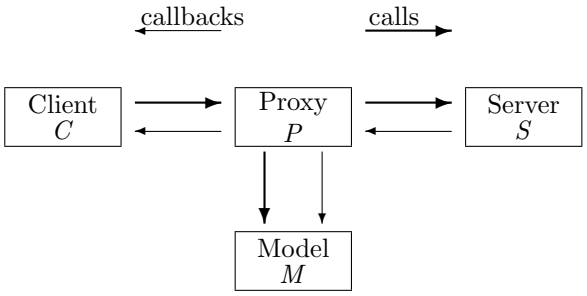

Figure 3: Proxy Architecture

and references to interfaces.

The method *minimized* is used only in Section 4.1. The use of *createInternalFigure* and the *IEnumFigure* interface will be left to Section 4.2. We defer the discussion of modifiers until Section 4.4; until then we discuss conformance checking for observers only.

3.2 Checking

We implement conformance checking for arbitrary clients by using a fourth component, P, which operates as a proxy, as shown in Figure 3. Using a proxy allows the interaction of the client C and the server S to be observed without having to instrument (i.e., modify) either component.

From now on, we use the letters C, S, M, and P to refer to the client, server, model, and proxy, respectively.

3.2.1 Architecture

All method calls between C and S are intercepted by P. As far as C is concerned, it is accessing the functionality provided by S and is unaware of either P or M. P manages the concurrent execution of M and S; it forks every call so that they are delivered to M as well as S. P compares the results from both components, checking at each interface call that they agree in terms of success/failure as well as any return values. (In our examples, we do not explicitly show the checks for the success or failure of the methods.) As long as they are the same, the results are delivered to C. Otherwise S and M are not behaviorally equivalent; the discrepancy is made evident to an observer of the system.

P also passes on any callbacks from S to C (and returns the results to S). Callbacks are delivered to M as well; it acts as a passive sink on the communication channel between C and S.

We create P automatically from the definition of the interfaces that are used between C and S. In the following we concentrate on (non-deterministic)

```
map as Map of Object to (Object * Object)

class PCanvas implements ICanvas
  createFigure(...) as IFigure =
    let (M, S) = map(me)
    let m = M.createFigure(...)
    let s = S.createFigure(...)
    let p = new PFigure()
    if (m = nothing) or (s = nothing) then
      throw ConformanceException(...)
    else
      map(p) := (m, s)
      return p
```

Figure 4: Deterministic *PCanvas.createFigure*

specifications for servers. The specification of callbacks has been explained in more detail in [3].

3.2.2 Tracking Interface References

P's ability to monitor all of the communication between C and S is enabled by its awareness of all interface references that are passed through any of the methods. Any object that is created in either C or S and which is made available to the other component is accessible only through some interface that it supports. The object's interface reference must first be passed through some method, at which point P can intercept and *spoof* it.

Methods *createFigure* and *getBorder* return interface references; since there are no callbacks in our example, no interface references are passed from C to S.

Figure 4 illustrates the proxy object *PCanvas* that implements the *ICanvas* interface. P maintains a global table *map*, which stores object references created in P to pairs of corresponding model and server object references. Initially this table just contains one entry: the reference of the root object of P to the roots of S and M. This entry is created when C first connects to P.

When *createFigure* on *PCanvas* is called, it forwards the call to the server and model objects M and S, respectively. M and S will return different objects representing figures, say m and s, respectively. If only one of the *createFigure* methods fails to return an object there is already a discrepancy. When both methods succeed, P creates a new object of type *PFigure* (for *P*roxy *Figure*), updates the global correspondence *map* and returns a reference to the new proxy figure p. The original interface reference s has been spoofed by p. (This is the standard way marshalling proxies are created for remote interfaces in COM [6].)

Figure 5 and Figure 6 show the implementation of the *Figure* proxy. Its methods are called by C, using the interface reference returned by P from

```
class PFigure implements IFigure
  getColor() as Color =
    let (M, S) = map(me)
    let m = M.getColor()
    let s = S.getColor()
    if s ≠ m then
      throw ConformanceException(...)
    else
      return s
```

Figure 5: Deterministic *PFigure.getColor*

createFigure.

The *getColor* and *getBorder* methods have the same prelude as *createFigure*. First, we look up the reference to the model M and the server S in the global *map*; next we call the appropriate methods. Once the results have been returned from M and S, P checks to make sure they agree. If the results are deterministic data (as in case of *getColor*), they must be the same. In case the results are determinstic object references (as in case of *getBorder*), they must stand in correspondence. That is, either they do not exist in the map, or else they must occur together as a pair that is indexed by a previously created proxy object that spoofs them. (Actually, the same checks are made in *createFigure*, but for this paper they have been removed since we assume that it creates a new figure that has never been returned before.)

4 Adding Non-determinism

The proxy P as described in Section 3.2 provides a mechanism to ensure that M can monitor the behavior of S as it is visible to C through the published interfaces. As shown, the ability to track object correspondence allows monitoring behavior in the presence of objects whose behavior becomes observable at an arbitrarily later time than their creation, and even for nested objects.

However, the approach is limited to deterministic specifications. When the specification is non-deterministic, M is allowed to provide any of a range of results. As long as S returns one of the allowed results, it is in conformance with its specification.

But if we use a non-deterministic specification for conformance checking, how can M make the right choice, so that S and M agree (if possible at all)? Or stated in other words, how can M exhibit *angelic choice*?

We describe our approach in a layered fashion: Section 4.1 explains how data non-determinism is handled, then Section 4.2 extends the solution to incorporate object non-determinism. Coping with nested objects is discussed in Section 4.3,

```
class PFigure implements IFigure
  getBorder() as IBorder =
    let (M, S) = map(me)
    let m = M.getBorder()
    let s = S.getBorder()
      throw ConformanceException(...)
    if ∃ p where map(p) = (m, s) then
      return p
    elseif ∃ p where
          first(map(p)) = m
            or second(map(p)) = s then
      throw ConformanceException(...)
    else
      let p = new PBorder()
      map(p) := (m, s)
      return p
```

Figure 6: Deterministic *PFigure.getBorder*

```
class MCanvas specifies ICanvas
  minimized() as Boolean = choose from {true, false}

class SCanvas implements ICanvas
  minimized() as Boolean = true
```

Figure 7: Components M and S for Data Non-determinism

and finally, Section 4.4 introduces the solution for methods that update the state of an object.

4.1 Data Non-determinism

We illustrate data non-determinism with the *minimized* method in the *ICanvas* interface. Its meaning is that the canvas, when created, is either displayed in a minimized or maximized view. Its specification, captured in the model *MCanvas*, is trivial: it either returns *true* or *false*. It is up to the implementation of S, here *SCanvas*, to make a choice, Figure 7.

Data non-determinism can easily be handled. Technically, the non-determinism is controlled by the introduction of an extra component: *ND* (for Non-Deterministic), a layer which is inserted in between P and M, as shown in Figure 8. *ND* contains interfaces and methods that are derived from M. As

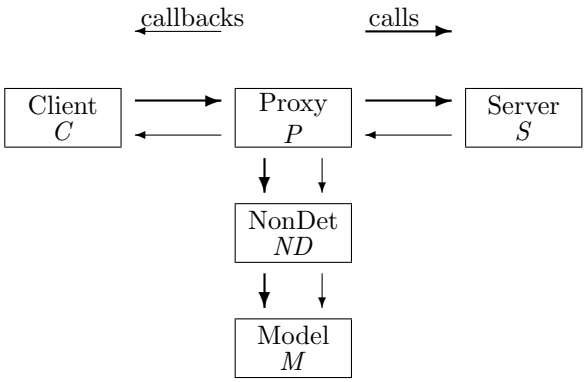

Figure 8: Non-deterministic Proxy Architecture

```
class NDCanvas specifies ICanvas'
  minimized() as Set of Boolean = {true, false}

class PCanvas implements ICanvas
  minimized() as Boolean =
    let (ND, S) = map(me)
    let m = ND.minimized()
    let s = S.minimized()
    if s ∉ m then
      throw ConformanceException(...)
    else
      return s
```

Figure 9: Components *ND* and *P* for Data Non-Determinism

will be seen in the examples, the interfaces are changed only in the return types of the methods. We use I' to denote the modified version of interface I from M. For any non-deterministic method f in M we introduce a corresponding method f in *ND*. Where $M.f$ chooses a particular element and returns it, $ND.f$ returns the range of possible values that $M.f$ chose from. P searches among the set of returned results from *ND* to determine if M allows a behavior that agrees with the result from S. The component *ND* and the adaption to the proxy P for our running example are shown in Figure 9. Note that the only difference between *minimized* in Figure 9 and *getColor* in Figure 5 is the implementation of the conformance test after the results from M and S are gathered: the latter checks for set membership, while the former required equality.

The non-deterministic layer, *ND*, completely captures the non-determinism; *ND* is derived automatically from the interface definitions and the definition of

```
class MCanvas specifies ICanvas
  getFigEnum() as IEnumFigure =
    return new MEnumFigure(figs)

class MEnumFigure(fs) specifies IEnumFigure
  var collection as Set of IFigure = fs
  Next() as IFigure =
    choose f ∈ collection do
      collection −= {f}
      return f
    ifnone // collection = ∅
      return nothing
```

Figure 10: The component M for Collections

M.

4.2 Object Non-determinism: Collections

When the specification non-deterministically chooses an object to return to the client, conformance testing becomes more complicated than for the data non-determinism.

We extend the solution from Section 4.1, using as an example a *figure enumerator*. Suppose C no longer uses the method *createFigure*, but instead *createInternalFigure*. The latter method does not return a reference to the *IFigure* interface on the created figure. Instead, after C has created (possibly) several figures, it then requests an enumerator to retrieve (interfaces to) the individual figures. Consequently, the proxy will not have seen their interface references, and so there is no correspondence between them and the figures created in the model.

We use the naming conventions of COM, and call the interface for iterating over collections of objects/interfaces *IEnumFigure*. It is a simplified version of a COM enumerator: *Next* returns one element (or nothing), rather than a list (array) of elements, and there are no methods to skip elements, to reset the enumerator, or to clone the enumerator.

Typically, the order in which the elements are enumerated is not specified; M and S may use different criteria for determining which figure to return.

Reflecting the lack of a specified order, M uses internal choice for picking which figure to return. The layer *ND* provides the angelic non-determinism by creating a non-deterministic figure, nd_f, that is to say, an object in which the collection of objects from M is embedded, as shown in Figure 11. The object nd_f has the capability to act *in the future* as any element in its embedded collection. We call this collection the object's *constraint set*.

```
class NDCanvas implements ICanvas
  var figs as Set of IFigure
  getFigEnum() as IEnumFigure =
    new NDEnumFigure(figs)

class NDEnumFigure(fs) implements IEnumFigure
  var collection as Set of IFigure′ =
    { new NDFigure(fs) | i ∈ {1..size(fs)} }
  Next() as IFigure =
    choose f ∈ collection do
      collection −= {f}
      return f
    ifnone // collection = ∅
      return nothing

class NDFigure implements IFigure′
  getColor() as SET of IFigure * Color =
    { (f, f.getColor()) | f ∈ constraints }
```

Figure 11: The component *ND* for Collections

As in Section 3.2.2, we use the ability of P to track the correspondence between objects, but now between objects from S and *ND*. At each call when an object from S is observed to behave in a particular fashion, somehow the object nd_f must behave the same way, as long as there is some figure in its constraint set that could behave in the same fashion.

Corresponding to the method *getColor* in the interface (*IFigure*) in M, there exists a method *getColor* in *ND*, as shown in Figure 11. When called, it probes its constraint set by calling the corresponding method of each object and returns the set of results to P. In addition to maintaining the correspondence between nd_f and the object returned from S, the constraint set of nd_f is *pruned* by P to remove all objects from M that cannot expose the behavior of (the object from) S. Figure 12 illustrates the modified version of the method (compare it to Figure 5).

Once a constraint set is pruned, all of the constraint sets in the collection of non-deterministic figures are checked to ensure that a feasible assignment exists of objects in *ND* to objects in M. For example, if a figure's constraint set has been pruned to the empty set, then that is clearly an infeasible situation: no object in M is able to behave according to the observed behavior of S. For the general case, consider the scenario where C has created three figures, f_1, f_2, and f_3. The enumerator will then be created with three non-deterministic figures, nd_i for $i \in \{1..3\}$, each of which will initially have the following constraint sets:

$$nd_1 \quad = \quad \{f_1, f_2, f_3\}$$

```
class PFigure implements IFigure
  getColor() as Color =
    let (ND, S) = map(me)
    let m = ND.getColor()
    let s = S.getColor()
    let pruned = { f | (f, v) ∈ m where v = s }
    M.setConstraints(pruned)
    let setOfConstraints =
      { f.constraints | f ∈ NDCanvas.figs }
    if not feasible(setOfConstraints)
      throw ConformanceException(...)
    else
      return s
```

Figure 12: The component P for Collections

$$\begin{aligned} nd_2 &= \{f_1, f_2, f_3\} \\ nd_3 &= \{f_1, f_2, f_3\} \end{aligned}$$

This reflects the fact that until at least one behavior is observed, each of the non-deterministic figures can act as any of the figures from M. Obviously at this point, a feasible assignment exists. Suppose that the client peforms sufficient method calls so that the constraint sets become:

$$\begin{aligned} nd_1 &= \{f_1, \mathbf{f_2}\} \\ nd_2 &= \{\mathbf{f_1}, f_2\} \\ nd_3 &= \{f_1, \mathbf{f_3}\} \end{aligned}$$

That is, there was a method call on nd_1 such that the result returned by f_3 did not match the result from S, and similarly for nd_2 and nd_3. A feasible assignment could be $nd_1 = f_2, nd_2 = f_1, nd_3 = f_3$ (as indicated by the bold face type in the constraint sets). Note that it would not be feasible for both nd_1 and nd_3 to be assigned f_1, the matching must be complete and unique. A bipartite-matching algorithm that computes a matching between objects in ND and objects in M is shown in Figure 13; it is called from *getColor* in Figure 12.

As long as a feasible assignment exists, P returns the same results as S. C and S then continue their mediated interaction. Otherwise, an exception is raised to indicate that S and M are no longer conformant.

Our feasibility analysis depends on knowing the number of underlying objects that exist in the model. In our example this was provided by the enumerator being provided the set of objects at its creation.

```
feasible(sets as Set of (Set of Object)) as Boolean =
  if size(sets) = 0 then
    return false
  elseif size(sets) = 1 then
    return size(choose from (sets)) = 1
  else
     let s = choose from sets
     let rest = sets − {s }
     return ∃ f ∈ s
        where feasible({s′ − {f} | s′ ∈ rest})
```

Figure 13: Feasibility checking

```
class NDFigure implements IFigure
  var children as Set of Object
  var border as IBorder′ =
     new NDBorder(me,
          { f.getBorder() | f ∈ constraints })
  getBorder() as IBorder′ = border

class NDBorder(p, constraints) implements IBorder′
  var parent as Object = p
  getWidth() as Set of IBorder ∗ Integer =
     { (b, b.getWidth()) | b ∈ constraints }
```

Figure 14: Allowing nested objects

4.3 Object Non-determinism: Nested Objects

A further complication arises when an object has nested objects, i.e., objects contained within it. In our example, figures have borders which support the interface *IBorder*; it contains a single method that returns the width (in some unit) of the border it represents.

Because figures are represented by non-deterministic objects, any nested objects must be also (and transitively for any objects nested within them). Consider the implementation of non-deterministic figures, *NDFigure*, as shown in Figure 14. It must have a member variable *border* to return from *getBorder*, but it is unable to return an interface to a particular border from M; instead, it must return a non-deterministic border, nd_b, which has its own constraint set. As depicted in Figure 15, nd_b's constraint set is the set of borders corresponding to the value of *f.getBorder()* for each f in the constraint set of nd_f.

Note that *getWidth* for *NDBorder* is implemented in the same way as *get-*

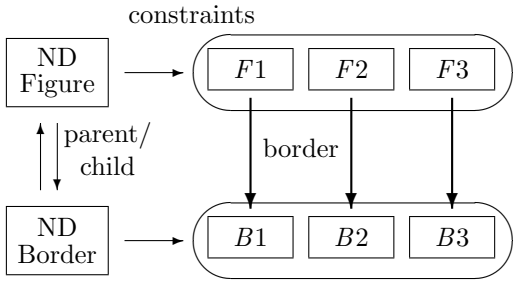

Figure 15: Nested Objects

Color was for *NDFigure*. Having nested objects means that the pruning of the constraint set of either nd_f or nd_b induces a pruning of the other object's constraint set. These induced updates of the constraint sets must propagate throughout the tree of any objects in *ND* which either contain the updated object, or are nested within the updated object. For instance, consider the situation depicted in Figure 15, and suppose that *getWidth* is invoked on nd_b. Furthermore, suppose that as a result of the value returned by S, the constraint set of nd_b is pruned to remove border $B2$. Then nd_f's constraint set must have figure $F2$ removed from it. On the other hand, suppose that *getShape* is invoked on nd_f and that as a result of the value returned by S, the constraint set of nd_f is pruned to remove figure $F3$. Then nd_b's constraint set must have border $B3$ removed from it. To effect the propagation of the prunings, non-deterministic objects are connected by parent and child(ren) references.

4.4 Modifying Object State

The final feature that we consider is to allow methods which update the state of an object. We do assume, however, that no global variables are updated and that no externally visible I/O (or any other covert channels of communication) is performed by the methods. These are not unreasonable restrictions for component-oriented programming. From the original interfaces in Figure 2, we now consider *setColor* and *setWidth*.

We modify the constraint sets so that they contain *copies* of the objects from M. The methods that update state are replicated in *ND* where they become updates of every object within that object's constraint set, as shown in Figure 16. Performing the updates on an independent copy guarantees that any state changes do not affect any other object.

The complication is that each constraint set now holds unique objects, so the equality test used as part of computing a feasible matching for collections must now compare the object of which each element is a copy of. To allow that, each copy must have a back reference to the original object.

```
class NDEnumFigure(fs) implements IEnumFigure
  var collection as Set of IFigure′ =
    { new NDFigure({ f.copy() | f ∈ fs })
      | i ∈ {1..size(fs)} }

class NDFigure implements IFigure′
  setColor(c as Color) =
    forall f ∈ constraints do
      f.setColor(c)

class NDBorder implements IBorder′
  setWidth(i as Integer) =
    forall b ∈ constraints do
      b.setWidth(i)
```

Figure 16: Allowing updates

5 Related Work

The need to specify and check components is widely recognized (cf. [12]). However there is still no standard way how to specify components nor any standard to check an implementation for conformance with the specification. We will shortly review both areas.

In a recent book, Leavens and Sitaraman [10] summarize the current approaches for specifying components formally. But only Müller and Poetzsch-Heffter's [11] article is really targeted towards the specification of interfaces. They propose to use pre/postconditions to specify interfaces. Of course pre/postcondition pairs can be seen as non-determinstic specifications. But these specifications are not executable in isolation; they can only be used for assertion checking. However, normally pre/postconditions do not allow the level of abstraction to vary; the data structures are fixed by the programming language. However, we believe that it is important to specify interfaces independently from implementation (and their data structures).

Jonkers, working at Phillips, is working on interface specifications, too [9]. In their work on Inspect, they use transition systems to provide the semantics for interface specifications. However they don't try to execute the model in isolation or run it in parallel with the implementation. Instead they want to generate black-box tests.

Closer to our work on conformance checking is the work on program checking as proposed by Blum and Wasserman [5]. They argue that it is often much easier to write a program that checks whether a result is correct, than to prove the algorithm correct that produces the result. For example, it is difficult to factor an integer, but, given x and y, it is trivial to determine whether or not y is a

factor of x. In our case the checker is the specification.

Using this idea, Antoy and Hamlet [1] propose the use of algebraic specifications to specify software. Algebraic specifications use high level data structures — thus solving one of the aforementioned problems of pre/postconditions — The price is that when checking the implementation against the specification one needs abstraction. Their system is able to run the executable specification (in fact it is a rewrite system) in parallel with the implementation in C; similar to our framework, they check the results on the method boundaries. They include a comprehensive review of similar work; we do not repeat it here. But due to the restricted nature of algebraic specifications, they cannot deal with state or with object identities (without a lot of coding), and they don't consider non-determinstic specifications at all. But we consider non-determinstic specifications absolutely essential.

Another similar project is the SLAM project by Herranz-Nieva and Moreno-Navarro [8]. They developed a new specification language and define class operations with pre/postconditions. The resulting specifications are tranlated to C++; part of the pre/postconditions are compiled to Prolog. Using a bridge between C++ and Prolog, the Prolog clauses are used as assertions during run-time. Results are speculative, since the project is in the early stages of development.

6 Conclusions

As far as we know, result checking of non-deterministic executable specifications of components, without instrumenting the components at all, is a new contribution.

To be useful in real-world applications, it is important that the management of the angelic non-determinism be automatically generated so that the specification writer is not burdened with all of the bookkeeping details it entails. We have a complete implementation for automatically creating the proxy for deterministic specifications. We currently have a prototype implementation for automatically creating the non-deterministic layer. However, it still requires a human to write the model...

The benefits of non-deterministic specifications do not come without cost: the checking can suffer from the same sort of exponential explosion that always results from reducing non-determinism to determinism, e.g., in reducing an NFA to a DFA. Whether this will occur often in practice will need to be studied.

We have used our methods to model two medium-sized components within Microsoft (each over 100K LOC), and performed conformance checking during user scenarios as well as in the context of testing using an automated test suite. Both times we have been able to find discrepencies between the actual component and its specification.

We believe that conformance checking shows promise in providing automated support for keeping a specification alive and for ensuring that an implementation correctly implements its non-determinsitic specification.

References

[1] Sergio Antoy and Richard G. Hamlet. Automatically checking an implementation against its formal specification. *Software Engineering*, 26(1):55–69, 2000.

[2] Mike Barnett, Egon Börger, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Using Abstract State Machines at Microsoft: A case study. In *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, page ??, Berlin, Germany, March 2000. Springer-Verlag.

[3] Mike Barnett, Colin Campbell, Wolfram Schulte, and Margus Veanes. Specification, simulation and testing of COM components using Abstract State Machines. In *Formal Methods and Tools for Computer Science, Eurocast 2001*, pages 266–270. IUCTC Universidad de Las Palmas de Gran Canaria, February 2001.

[4] Andreas Blass, Yuri Gurevich, and Saharon Shelah. Choiceless polynomial time. *Annals of Pure and Applied Logic*, 100:141–187, 1999.

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

[6] Don Box. *Essential COM*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1998.

[7] Yuri Gurevich. Evolving algebra 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, Oxford, UK, 1995.

[8] Angel Herranz-Nieva and Juan Jose Moreno-Navarro. Generation of and debugging with logical pre and post-conditions. http://lml.ls.fi.upm.es/slam/.

[9] H.B. Jonker. Ispec: Towards practical and sound interface specifications. In *IFM'2000*, volume 1954 of *LNCS*, pages 116–135, Berlin, Germany, November 1999. Springer-Verlag.

[10] G. T. Leavens and M. Sitaraman (eds.). *Foundations of Component-Based Systems*. Cambridge University Press, New York, NY, 2000.

[11] P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In *Foundations of Component-Based Systems* [10], pages 137–160.

[12] Clemens Szyperski. *Component Software*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.