The ABCs of Specification: AsmL, Behavior, and Components

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

Keywords: Specification, Component, Subtyping

Edited by:
Received: **Revised:** **Accepted:**

We show how to use AsmL, an executable specification language, to provide behavioral interfaces for components. This allows clients to fully understand the meaning of an implementation without access to the source code. AsmL implements the concept of behavioral subtyping to ensure the substitutability of components and provides many advanced specification features such as generic types, transactional semantics, invariants and history constraints.

1 Introduction

There is a broad consensus that a specification of a component's interface must include some way of describing its behavior [26, 32, 36, 43]. Current practice tends towards formal specification of the syntax of the interfaces while using informal natural-langugage descriptions for the semantics. Current theory is based on the idea of design by contract [35], generally using pre- and post-conditions. Previous attempts at describing software also have used algebraic specifications [24].

Interfaces, as they are standardized today, for example using IDL [11], are clearly inadequate for the task of specifying components. It is not enough to provide merely the syntax — signature — for each method contained in an interface. A client who wishes to use a component needs to know the semantics — behavior — of each method. In addition, understanding the relationships between the methods contained in an interface is crucial for the effective use of a component supporting that interface.

We follow the specification taxonomy of Beugnard et al. [9]. A specification is a contract for a software component that describes properties on four levels:

1. *basic*: the syntactic properties of method names, number and type of parameters and very simple semantic properties (e.g., in IDL one can specify whether a parameter that is a pointer can ever be null or not).
2. *behavioral*: the properties that can be specified with pre-conditions, post-conditions, and invariants, including history constraints [34].
3. *synchronization*: properties of component interaction.
4. *quantitative*: all non-functional properties, such as quality of service, response times, throughput guarantees, etc.

Our method for specifications covers only the first three levels; however, we use the term *behavioral* to refer also to the synchronization class of specification.

Our group at Microsoft Research, the Foundations of Software Engineering [16], has developed an executable specification language, AsmL, which is based on the theory of Abstract State Machines (ASMs) (see [22] for an introduction to the notion of ASMs). ASMs allow precise, formal, operational specifications of software systems. AsmL has many important features, among which are generic interfaces and classes, and a transaction-based semantics.

In this paper, we use AsmL to specify the behavioral and synchronization properties of component interfaces, in particular method behavior, interface-wide invariants, history properties, and component composition.

In previous work we used AsmL to write component models by reverse-engineering already existing components [6]. The resulting models provided essentially the identical functionality as the components they were models of. In other words, they modeled the classes that implemented the components. Our concern here is the use of AsmL at the design stage by providing models of *interfaces*. We wish to specify interfaces at their most general level: only the required behavior any component implementing them must have is detailed. The rest is left up to the implementer of the component. An interface model allows clients and implementers to understand the behavior of a software component that correctly implements the interface.

We believe that one should implement a component using classes, i.e., using object-oriented programming, but that the specification should be done at the interface level. The key idea connecting a class to its interface is that the class must be a *behavioral subtype* [34] of its interface. An interface specification describes the minimal behavior expected of all of its subtypes: the behavior of a class can be more constrained than that of its interface.

This paper's contribution is to provide a clean layer in which full behavioral specifications can be written. Specification languages should not be tightly coupled to implementation languages. Precise semantics are crucial for a specification language; implementation languages are oriented towards execution efficiency, as indeed they should be. AsmL has a formal semantics which provides a mathematical foundation for the specification effort. It provides features that aid in the refinement process for developing components that correctly implement their specifications.

The paper is organized as follows. Section 2 provides more detail on exactly what an interface specification looks like in AsmL. Section 3 discusses our notion of refinement and provides an example. Then, in Section 4, we show how to handle component creation and parameterization within AsmL. The next two sections explain how to compose specifications: Section 5 for data-linking and behavior-linking and Section 6 for aggregation and delegation. An overview of similar approaches is discussed in Section 7. Section 8 summarizes and presents limitations and future work.

2 Specifications

We write executable specifications of components in AsmL (the Abstract State Machine Language). AsmL is based on the theory of Abstract State Machines [22]. ASMs are transition systems: their states are first order algebras, that is, interpretations of a functional signature. The transition relation is specified by transition rules (in the sequel simply called rule) 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.

To deal with industrial applications, we have extended ASMs with submachines, objects, exception handling [23] and a very powerful type system (as have others, see [2, 8, 10]). AsmL is freely available for non-commercial research or teaching purposes from our web site [16]. It is currently used within Microsoft for modeling, rapid prototyping, analyzing and checking of APIs, devices and protocols.

We introduce AsmL at the same time as we develop the examples. Only a small subset of AsmL will be used. Our first, very small, example is a specification of a counter interface.

```
interface ICounter
  var ct as Integer = 0
  Counter() as Integer
    return ct
  Increment()
    ct := ct + 2
```

To specify components we use interfaces. Stateful interfaces have member variables, which are also called *model variables*. Model variables are not part of the signature of the interface; they are provided only to give meaning to the method bodies. They are accessible only through the methods defined in the containing interface and its subtypes.

Method bodies in an interface are called *model programs*: they specify the effect that any implementation must respect. Method bodies typically refer to member variables. If a method body updates a member variable, it defines an ASM rule. ASM rules are inherently parallel. This synchronous parallelism comes in handy when specifying independent updates. For example to swap two variables you write:

```
swap()
  x := y
  y := x
```

Sequential composition is the unusual case; to discourage its use, we require a "heavy" notation for it. The sequential AsmL specification for swapping the values of two variables uses an ASM *sub-machine*:

```
swap()
  var t = x
  step x := y
  step y := t
```

AsmL also provides exception handling. Combined with synchronous parallelism, this eases specifications: when an exception is thrown all updates that are produced in the protected block are undone.

The simple transition semantics also simplifies the translation of AsmL rules into predicates. For this purpose we use a slight variation of weakest preconditions. This allows the counter also to be specified in more declarative terms.

```
interface ICounter
  var ct as Integer ct = 0
  Counter() as Integer
    require true
    ensure result = ct and ct = ct′
  Increment()
    require true
    ensure ct′ = ct + 2
```

The keywords require and ensure are used for pre- and post-conditions, respectively. Priming (e.g., x') denotes the value in the next state of a run. The keyword result refers to the value returned by the method.

In general, any straight-line method body can be automatically replaced with a pre- and post-condition pair that specifies the same behavior. Loops and recursion require manually-supplied invariants and bounds.

In AsmL a method application changes only those variables that occur in the computed update set; variables not mentioned in the update set are not changed. If a method body is only described by a pre-/post-condition pair one has to specify explicitly which variables change and which retain their values. If no method body and no pre-/post-condition pair is given, the method can do whatever it wants to, except that is has to respect any interface invariants and constraints (as described in Section 3).

Not only do pre- and post-conditions fail to scale with larger specifications [12], but we have found that real users prefer writing executable specifications instead of pre-postcondition pairs. In AsmL, users can use high-level data structures, users can write nondeterministic specifications, users get atomic transition semantics, and users get ease of reasoning due to referential transparency within each step. Furthermore they can immediately execute the written AsmL specifications.

3 Refinement

A specification is useful only in so far as it defines properties that are true for any implementation. In essence, this is Liskov and Wing's notion of behavioral subtyping [34]: a subtype should always be substitutable for a basetype in all contexts. ASMs can be used in a more general theory of refinement (see e.g. [41]), but for our purposes it suffices to restrict our attention to the $1 : n$ refinements possible in the syntactic framework of classes implementing interfaces. That is, any component implementing an interface must support the syntactic interface; it may do less or more work within each method, but the protocol by which a client uses the functionality is fixed by the syntax of the interface.

There is a well-known problem with specifications and behavioral subytping: a subtype might violate properties of its basetype. For example, in the case of the *ICounter* specification, one cannot reason that the value is always even: as specified, a subtype could increment the counter only by one. Likewise, the counter cannot be assumed to always be positive, a subtype might introduce a decrement method. In order to compensate for this, Liskov and Wing require invariants, which are properties of a single state, and constraints, which in AsmL are properties of consecutive states. For instance, to ensure the two above-mentioned properties, we can add to the *ICounter* interface:

```
interface ICounter ...
  invariant even(ct)
  constraint ct ≤ ct′
```

The ellipsis (three dots) is part of our literate programming environment [31]; it indicates that this is a continuation of a previous construct.

AsmL also introduces an alternative construct for an operational specification of the permitted state transitions of any method in any subtype: the *others* clause. For instance, to ensure the even stronger property that any other method can increment *ct* only by a multiple of two in the range from 0 to 20 one can write:

```
interface ICounter ...
  others(...)
    choose i in {0, 2..20}
      ct := ct + i
```

Any additional method defined in any subtype of *ICounter* will inherit the derived post-condition from the others method.

Our notion of refinement for synchronization properties depends on the concept of a *mandatory call.* Certain method calls in the model programs are identified as communications that any implementation must make during the execution of the corresponding method. All calls to non-local public interface methods are mandatory calls. This includes constructors, see Section 4 for an example. Note that it is the call site that is mandatory, not the method definition. An implementation is free to make additional calls; the model indicates the minimal behavior that must be observed. Thus, we say that an AsmL specification provides a *minimal model* for any implementation.

Classes that implement an interface must be a behavioral subtype of the latter. But the implementation typically chooses a different representation of its fields. Contrary to Liskov and Wing's formulation, we do not require that the class defines an abstraction function (see also Hoare [28]) which relates the concrete state of the class to the abstract state of the interface. In other work [6] we outline a scheme that provides for run-time checking of the subtype relationship without an abstraction function.

However, providing an abstraction function allows for a higher level of verification; AsmL allows a class to define one with the abstraction construct. Suppose that the class that implements the *ICounter* uses a "successor" representation for a counter. Then the abstraction function is just two times its successor representation.

```
class CCounter implements ICounter
  var succ as Integer = 0
  abstraction
    ICounter.ct = 2 * succ
  Counter() as Integer
    return 2 * succ
  Increment()
    succ := succ + 1
```

In this particular example, it is obvious how *CCounter* fulfills the obligations it inherits when implementing the *ICounter* interface. However, in general, abstractions can be much more complicated.

There is no requirement that an AsmL specification be implemented in AsmL. AsmL provides native COM connectivity (as well as COM Automation) and so can be used directly with a component implemented in any programming language.

One interface may also refine another interface, either by extension (see Sections 5 and 6) or implementation. Again, the former interface must be a behavioral subytpe of the latter interface.

To simplify rapid prototyping, i.e., executing of specifications, AsmL classes don't have to provide their own definitions. As long as interface methods are specified by method bodies, interfaces are executable exactly as written. Thus a class can *reuse* the definitions of the interfaces. The simplest implementation for *ICounter* then becomes:

```
class CCounter reuses ICounter
```

Thus it is often sufficient to close a specification by merely providing a class that reuses the specification.

4 Creation and Parameterization

In this section, we consider two prerequisites for composing interfaces. First, there must exist a way to specify the *creation* of a reference to an interface. An interface is merely a view on a component (namely a particular subset of the component's functionality): what does it mean to have a new reference to one? Second, an interface can be dependent on external values (and/or objects); a completely closed interface is not particularly interesting. The simplest forms of dependency are ones required for *parameterizing* an interface: by type and by value.

Creation. At the interface level there are only interfaces, not components. So if one wishes to access a new interface, where does it come from?

One solution would be to parameterize all interfaces by a *factory interface* that can be used to request the desired interface. A factory interface contains a method which will deliver an interface reference upon request, given some sort of identifier for the interface. But this merely pushes the problem back one level: where does the specification of the factory interface get the interface reference to return? What exactly are the properties of the returned interface?

While factory interfaces are very useful at the implementation level in order to decouple component creation and allow subclassing [17], AsmL interfaces are already expressed at the abstract level. A clearer picture of the desired properties is needed.

When a component is created, there are several assumptions about the resulting reference. Abstracting from the specifics of implementation issues, such as storage allocation, leaves us with the following properties: the component supporting the requested interface

1. should have a unique identity,
2. should not be aliased, and
3. should provide the requested interface in one of its initial states.

Such an interface is guaranteed to be private to the component that is requesting it, unless it explicitly decides to share the reference either by creating aliases or by passing the reference to a third party. For this concept, we use the keyword new with an interface:

```
interface IHistory
  var s as ICounter = new ICounter
```

However, it is important to note that the use of new does not necessarily imply the creation of an object as it would when used on a class. As long as properties 1–3 are ensured for s, then it does not matter if a new class object is created by actually calling a constructor or not.

The above example specifies that within the interface *IHistory*, the name s refers to an interface *ICounter* on *some* component. Only *IHistory* has a reference to this component. Furthermore, this component is in its initial state, i.e., $s.ct$ is equal to zero, and will remain so until changed by a call from within *IHistory*. The fact that the component has a unique identity will be utilized in Section 6.

Sometimes a new interface is requested on an already referenced component, i.e., an existing interface reference. In AsmL that is modeled by a type cast:

```
// ... i is an interface reference to IA ...
let j = i as IB
...
```

This corresponds to using the COM method *QueryInterface* [11]. When the type cast is successful, the requested interface is *not* necessarily in its initial state.

Parameterization. An interface can be dependent on a type, i.e., it can be a generic interface. A generic interface specifies a family of interfaces all of whom instantiate the generic parameter for some particular type. A typical example for a generic interface is the *IState* specification:

```
interface IState⟨T⟩
  private var value as T
  Set(v as T)
    value := v
  Get() as T
    return value
```

The *IState* specification says nothing about its initial state; it is also dependent on a value of type T that must be supplied to the *constructor* when an instance of *IState* is created. AsmL provides a default constructor that has the same name as the interface. The default constructor takes a parameter for each of the uninitialized member variables:

```
interface IState⟨T⟩ ...
  IState(v as T)
    value = v
```

In order to be instantiated, the interface *IState* is dependent on both the type parameter T and supplied argument for *value*. Note that it is just a coincidence that the type of *value* is itself T. Multiple constructors with different parameter lists are also allowed.

The visibility attribute private on *value* means that it may not be modified by a method within any subtype. Therefore the only way to modify *value* is to call *Set*. This guarantees the property that once a client calls *Set*, *value* will remain unchanged until the next call to *Set*. In other words, any component implementing *IState* will act like a program variable.

5 Linking Specifications

While it is important to be able to specify interfaces in isolation, true component-oriented programming can be realized only when sub-units are composed to make larger units. This implies that we must be able to compose interfaces as well, since the specification for the composition of two components should be the composition of their individual specifications.

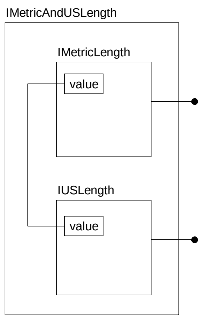

Figure 1: Linking two interfaces by shared data

5.1 Data Linkage

Linking two specifications through shared data — state-coupled specifications — allows for multiple viewpoints on the same component, while ensuring that the component stays in a consistent state. This represents a common pattern; our example uses the idea of different units for a single measurement [20]. For instance, suppose there are two interfaces.

```
interface IMetricLength extends IState⟨Integer⟩
  IMetricLength() extends IState(0)
interface IUSLength extends IState⟨Integer⟩
  IUSLength() extends IState(0)
```

The specification for *IMetricLength* implicitly keeps *value* in metric units, e.g., centimeters. Meanwhile, the specification for *IUSLength* implicitly keeps *value* in inches. Note that neither interface is parameterized: the generic parameter T from *IState* has been instantiated to *Integer*. Also, the explicit constructors take no arguments. But they call the constructors of the interface they are extending; the initial state is thus fully determined.

Suppose we would like to specify a component that provides *both* interfaces with a consistent shared value. Whatever changes are made through one interface should be reflected in the other interface. This is easily specified via a *linking invariant* which constrains any implementation to meet this condition. Fig. 1 shows the structure of the composition.

```
interface IMetricAndUSLength
    extends IMetricLength and IUSLength
  invariant
    IMetricLength.value * 2.54 = IUSLength.value
```

A crucial feature of AsmL is that all methods and member variables from inherited interfaces are kept distinct. The interface *IMetricAndUSLength* does not identify the methods *Get* and *Set* from the two interfaces; the combined interface has all four methods. AsmL, just as C# [25], does *not* fold methods with the same name and signature when extending multiple interfaces. This is especially important for generic interfaces. Java [19], for instance, is unable to keep the methods distinct.

The behavior of any component implementing the interface *IMetricAndUSLength* must respect the invariant (as well as the individual behaviors specified in each interface). How it does so is left up to the component; one way is to keep *value* in one unit and converting it for the other interface:

```
class CMetricAndUSLength
      implements IMetricAndUSLength
  var metricValue as Integer = 0
  abstraction
    IMetricLength.value = metricValue
    IUSLength.value = metricValue / 2.54
  IMetricLength.Set(v as Integer)
    metricValue := v
  IMetricLength.Get() as Integer
    return metricValue
  IUSLength.Set(v as Integer)
    metricValue := v * 2.54
  IUSLength.Get() as Integer
    return metricValue / 2.54
```

This example differs from the traditional Observer pattern [17] in that both of the original interfaces are *peers*; neither is distinguished as the subject holding the "correct" value (although the component decided to implement it that way).

5.2 Behavior Linkage

In this section, we present an example of two components that are coupled through their interacting behaviors instead of through shared state. We use the Observer pattern [17] which involves two components: a subject and a set of observers, called *views*.

The *IView* interface is trivial: it contains a method *Update* that is to be called by the subject, and a method for registering the subject with the view so it has access to the subject.

```
interface IView⟨T⟩
  var subject as ISubject⟨T⟩
  Update()
    // behavior goes here, to be defined by subtype
  SetSubject(s as ISubject⟨T⟩ )
    subject := s
```

The subject holds some state; whenever the state is changed, each view is notified. This is a generalization of the Reader/Writer paradigm. The specification of a subject is an extension of *IState* that has methods for adding, removing, and alerting views:

```
interface ISubject⟨T⟩ extends IState⟨T⟩
  var views as Set⟨IView⟨T⟩⟩ = {}
  Set(val as T)
    step base.Set(val)
    step Notify()
  private Notify()
    forall v in views
      v.Update()
  Attach(v as IView⟨T⟩)
    views += { v }
  Detach(v as IView⟨T⟩)
    views −= { v }
  others(...)
    ensure value = value′
```

There are three interesting properties that this specification prescribes for any implementation:

1. A subject calls the *Update* method of each view whenever its *Set* method is called. Because *Update* is a public interface method, this call is a mandatory call. An implementation is free to call *Update* more than once, perhaps for fault-tolerance purposes.
2. Views are synchronized with subjects. That is, all views receive a notification with the subject in the same state. This is because the forall loop used within *Notify* is a parallel loop.
3. A view can perform any behavior within its *Update* method. Obviously, it would be unwise to call the subject's *Set* method: allowing the state to change during a callback is known to create problems [43]. The specification can easily be modified to disallow it.

Because of the others clause, no subtype of *ISubject* is allowed to add a method that alters *value* other than by calling *ISubject.Set*. This may be too restrictive; one can specify instead a constraint that connects state changes to *value* with calls to *Update* for each view.

The method *Notify* is marked private to emphasize that it is not a mandatory call. It is only the call to *Update* during the execution of *Set* that must be observable.

6 Aggregating Specifications

In addition to linking interfaces, we use AsmL to define interfaces that re-use existing behaviors to create new functionality. This explores another way of composing specifications which can be seen as aggregation or delegation depending on the details of how it is specified.

We take the example of a radio button group in a graphical user interface from [26]. A radio button group is a set of radio buttons that operate in a mutually-exclusive manner. At most one of them can be selected at any one time; selecting one radio button unselects all of the others in the group. Each button in the group must display itself appropriately as either selected or not.

A radio button group can be seen as an example of reusing the Subject/View contract (i.e., the Observer pattern [17]): each radio button is a view on a subject whose state reflects which button is currently selected. To begin the specification, we first specify the behavior of buttons in general.

6.1 Buttons

We model a button as a user-interface element that has a text label and allows the user to select it, say by clicking on it with the mouse.

```
interface IButton
  var label as String
  var chosen as Boolean
  GetLabel() as String
    return label
  SetLabel(s as String)
    label := s
  Select()
    choose b in { false, true }
      chosen := b
```

Of course, the interface would have additional methods relating to its size, color, etc.

A checkbox button acts as a toggle: clicking it reverses its current state.

```
interface ICheckBoxButton extends IButton
  Select()
    chosen := not chosen
```

A radio button, by way of contrast, is idempotent: clicking on it sets it to true. The only way to unselect it is to select another radio button in the same group.

```
interface IRadioButton extends IButton
  Select()
    chosen := true
```

A single radio button may seem useless, but could be used for signing a document or some other irreversible operation.

6.2 ButtonView

A radio button, as specified in *IRadioButton*, is not immediately composable into a group. As stated, the interface does not provide any functionality for synchronizing its state with other buttons in the same group. This clearly separate behavior can be added in a modular fashion.

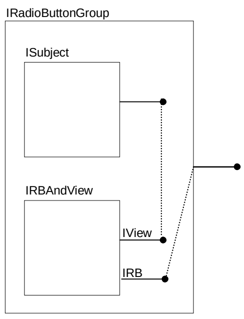

Figure 2: The *IRadioButtonGroup* interface

We *extend* the *IRadioButton* interface with an *IView* interface. It describes a button that just behaves like a radio button, but responds to a new input notifying it that some state has changed somewehere else.

```
interface IRadioButtonView extends IRadioButton
        and IView⟨IRadioButton or Undef⟩
  // explicit constructor
  IRadioButtonView(s as String)
    extends IRadioButton(s, false) and IView(Undef)
  Update()
    chosen := subject.Get() = me
    // redraw appropriately ...
  Select()
    base.Select()
    subject.Set(me)
```

The explicit constructor initializes the button to be unselected. In addition it initializes the view's subject to be undefined. Note that in AsmL reference types don't contain a null value. But disjunctive types, here exemplified by the type *IRadioButton* or *Undef* give you the flexibility to add *Undef* when needed. The keyword base refers to the immediate supertype, in this case *IRadioButton*.

Given this interface, it is now easy to define the behavior of a radio button group.

6.3 ButtonGroup

The requisite behavior of having the radio buttons be mutually exclusive is achieved by wiring the augmented radio buttons from Section 6.2 together with a component that implements the *ISubject* interface into a Subject/View relationship.

```
interface IRadioButtonGroup
  var bs as Set⟨IRadioButtonView⟩
  var subj as ISubject⟨IRadioButton or Undef⟩
```

Figure 2 shows the structure of the interface, although not the multiplicity of views.

A radio button group has its own interface: there are operations that make sense for a button contained in a group, but not for the collection as a whole and vice versa. Selection of one button in the group is specified by delegating the *Select* call to the appropriate button.

```
interface IRadioButtonGroup ...
  Select(s as String)
    choose b in bs where b.GetLabel() = s
      b.Select()
  ... other methods ...
```

The constructor for *IRadioButtonGroup* takes a set of labels for the radio buttons to be generated, generates the sub-components and wires them together.

```
interface IRadioButtonGroup ...
  IRadioButtonGroup(ls as Set⟨String⟩)
    subj = new ISubject(undef)
    bs = { new IRadioButtonView(l) | l in ls }
    forall b in bs
      b.SetSubject(subj)
      subj.Attach(b)
```

Informal Reasoning. Now that the composition has been created, it can be reasoned about. Here we give an informal outline of *IRadioButtonGroup*'s correctness for maintaining the mutual exclusion of a selected button.

The only external action that can cause an update is for one of the radio buttons b_1 to b_n in the group to have their *Select* method called. Without loss of generality let's assume that b_1 is selected. This, in turn, will cause b_1.*IRadioButtonView.Select* to be called, which will call b_1.*IRadioButton.Select*. So b_1.*chosen* becomes true.

The button b_1 will also call *Set*(b_1) on the shared subject. *First*, its *value* becomes b_1. *Next*, the shared subject will call back to every button in the group via *IRadioButtonView.Update*. For every button the *Get* method called on the shared subject will return b_1 — this is the value that was just stored. For b_1 this generates another update of b_1.*chosen* to true; this is a non-conflicting update. In contrast, the *chosen* field of the buttons $b_2 \ldots b_n$, become false, since b_1 is different from any of $b_2 \ldots b_n$.

As a result, we are guaranteed that the group makes an atomic step which preserves the property that at most one button can be selected at any time.

Given AsmL's transactional semantics, it is possible for two buttons to execute their *Select* methods in the same step. Each method will cause an update in the subject for two different values (the value of me for each of the buttons). These updates are *conflicting*; AsmL's runtime checks for different values being assigned to the same location at the end of each step and will signal an exception.

7 Related Work

As long as there have been programmers, there has been concern with the meaning of the artifacts they create by formally specifying the programming process, e.g. [27]. Here, we concentrate specifically on work involving components.

There is a long tradition within the object-oriented community that is concerned with specification, whether formal or not. Meyer, of course, is famous for his ideas on *design by contract* [35]. Over a decade ago, Helm et al. [26] pointed out the necessity for *contracts* and how they can be used as a structuring concept for specifications, but their contracts were a) not executable, and b) confused the *wiring* of components with the specification of their *interaction*. America [1] did some of the early work on behavioral subtyping. The most standard formulation of behavioral subtyping follows that of Liskov and Wing [34]. Most of this work used only pre- and post-conditions for methods, or did not consider using a separate specification language.

Leavens and Dhara provide specifications for Java components using a language called JML [33]. Like us, they insist on behavioral subtyping as a refinement notion and also use *model programs* in addition to pre- and post-conditions. However, their work is limited to Java programs; AsmL can be used in conjunction with any implementation language. They make the distinction between *strong* and *weak* subtyping; we restrict our attention to strong subtyping since AsmL does not prohibit aliasing.

Besides JML, there has been a lot of work on using assertions to specify Java interfaces, e.g., Contract Java [15], iContract [13], jContractor [30], and Jass [7] all implement various schemes to implement design by contract for Java programs. JISL, the Java Interface Specification Language [40], translates and inserts specifications into Java programs. It uses pre- and post-conditions and is used to primarily specify and check frame properties.

Edwards [14] 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 [42] 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.

Jonkers [29] has interface specifications that are not executable; he also does not insist on absolute rigor in a specification. But his ideas of how to specify interfaces are very similar to ours.

The theoretical background for component specification is mostly based on the refinement calculus by Back and Wright [4] and Morgan [39]. Constructs for object-oriented programming are added to a notation for sequential computing and class refinement is defined such that it respects supertype behavior [3]. To declare a class as a subtype of another means to do a proof in the refinement calculus that the predicate transformer semantics of the class hold the correct relationship with those of the superclass. However, there does not seem to be a concern with directly executing specifications. Sekerinski et al. have explored the restrictions on component-oriented programming that are needed in order to be able to prove refinement in the presence of recursive re-entrance [37]. They have also done a small case study of proving the correctness of Java Collections Frameworks [38]. They extend Java with a specification language and claim that it has a formal mathematical foundation: "every executable statement of the Java language... that we use has a precise mathematical meaning". We take that to mean that only a subset of Java is used.

8 Conclusions

The need for behavioral specifications is widely recognized, especially in component-oriented programming. AsmL provides an industrial-strength tool for writing such specifications. It provides all of the features necessary to express the properties needed for behavorial subtyping.

AsmL is agnostic with regards to verification technology. An AsmL specification can be subjected to analysis with a variety of formal methods, for instance, a refinement calculus proof.

The executability of AsmL specifications opens possibilities that go beyond those traditionally associated with specification languages. A formal specification is the boundary between the informal understanding of a system and its digital incarnation. At the design stage, exploration of the specification provides insight and feedback about the appropriateness of the formalization. During the coding process, the specification can be used, in special domains, to derive test cases and perform conformance testing [18, 21]. An executable specification allows conformance checking, i.e., assertion monitoring, to ensure that an implementation's behavior is allowed by the specification [5, 6]. Furthermore, AsmL's COM connectivity means that it can be used in a language-neutral setting: any language can be used to implement the specification.

There are many areas that need to addressed in future work. For example, adding automatic support to enforce the kind of restrictions needed for refinement proofs [37] or other proof tools.

Acknowledgements

We wish to thank the rest of our research group in Microsoft Research, FSE. Clemens Szyperski, Egon Börger, and Crispin Goswell reviewed earlier drafts of this work and made many useful suggestions.

References

[1] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bezivin et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 234–242, New York, NY, June 1987. Springer-Verlag. Lecture Notes in Computer Science, Volume 276.

[2] M. Anlauff. XASM – An Extensible, Component-Based Abstract State Machines Language. In Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, editor, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 69–90. Springer-Verlag, 2000.

[3] Ralph Back, Anna Mikhajlova, and Joakim von Wright. Class refinement as semantics of correct subclassing. Technical Report 147, Turku Centre for Computer Science, December 1997. Available from `www.tucs.abo.fi` at `/publications/techreports/TR147.html`.

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

[5] Mike Barnett, Lev Nachmanson, and Wolfram Schulte. Conformance checking of components against their non-deterministic specifications. Technical Report MSR-TR-2001-56, Microsoft Research, June 2001. Available from `http://research.microsoft.com/pubs`.

[6] Mike Barnett and Wolfram Schulte. Spying on components: A runtime verification technique. In *Workshop on Specification and Verification of Component-Based Systems, OOPSLA 2001*, pages 7–13. Published as Iowa State Technical Report #01-09a, October 2001.

[7] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass — Java with Assertions. Available from `http://semantik.informatik.uni-oldenburg.de` at `~jass/doc/index.html`.

[8] H. Baumeister and A. Zamulin. State-Based Extension of CASL. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Integrated Formal Methods (Proceedings of IFM 2000)*, volume 1945 of *LNCS*, pages 3–24. Springer, 2000.

[9] Antoine Beugnard, Jean-Marc Jézéquel, Nöel Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–44, July 1999.

[10] E. Börger and J. Schmid. Composition and Submachine Concepts for Sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *LNCS*, pages 41–60. Springer, 2000.

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

[12] Martin Büchi and Emil Sekerinski. Formal methods for component software: The refinement calculus perspective. In *Proceedings of the Second Workshop on Component-Oriented Programming (WCOP)*, June 1997. Available from `ftp://ftp.abo.fi` at `/pub/cs/papers/mbuechi/FMforCS.ps.gz`.

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

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

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

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

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

[18] Uwe Glässer, Yuri Gurevich, and Margus Veanes. Universal plug and play machine models. Technical Report MSR-TR-2001-59, Microsoft Research, June 2001. Available from `http://research.microsoft.com/pubs/`.

[19] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[20] Crispin Goswell, 2001. Personal communication.

[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] Yuri Gurevich. Evolving algebra 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, Oxford, UK, 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] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.

[25] Anders Hejlsberg and Scott Wiltamuth. C# language specification, version 0.22. Available at `http://msdn.microsoft.com/library/default.asp`.

[26] 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).

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

[28] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[29] 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.

[30] 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.

[31] Donald E. Knuth. Literate programming. *Computer Journal*, 27(2):97–111, May 1984.

[32] Gary T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, July 1991.

[33] Gary T. Leavens and Krishna Kishore Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, 2000.

[34] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

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

[36] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[37] Leonid Mikhajlov, Emil Sekerinski, and Linas Laibinis. Developing components in the presence of re-entrance. Technical Report TUCS-TR-239, TUCS - Turku Centre for Computer Science, February 1999.

[38] 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.

[39] Carroll Morgan. *Programming from Specifications*. Prentice Hall International, Hempstead, UK, 1990.

[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] G. Schellhorn. *Verification of Abstract State Machines*. PhD thesis, Universität Ulm, Ulm, Germany, 1999. Available from `http://www.informatik.uni-ulm.de` at `/pm/mitarbeiter/gerhard/`.

[42] 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.

[43] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.