Contracts: Specifying Behavioral Compositions in Object-Oriented Systems

Richard Helm[†], Ian M. Holland[‡] and Dipayan Gangopadhyay[†]

[†]I.B.M. Thomas J. Watson Research Center,
P.O. Box 704, Yorktown Heights, NY 10598

[‡]College of Computer Science, Northeastern University,
360 Huntington Ave., Boston MA 02115

Abstract

Behavioral compositions, groups of interdependent objects cooperating to accomplish tasks, are an important feature of object-oriented systems. This paper introduces *Contracts*, a new technique for specifying behavioral compositions and the obligations on participating objects. Refinement and composition of contracts allows for the creation of large grain abstractions based on behavior, orthogonal to those provided by existing class constructs. Using contracts thus provides a basis and vocabulary for *Interaction-Oriented* design which greatly facilitates the early identification, abstraction and reuse of patterns of behavior in programs. Contracts differ from previous work in that they capture explicitly and abstractly the behavioral dependencies amongst cooperating objects. By explicitly stating these dependencies, contracts also provide an effective aid for program understanding and reuse.

1 Introduction

> "... *no object is an island.*
> *All objects stand in relationship to others,*
> *on whom they rely for services and control*"
> Beck & Cunningham 1989[2].

Indeed, not only are objects not islands, but within an object-oriented system, groups of related objects will often cooperate to perform some task or maintain some invariant. We call such groups of cooperating objects behavioral compositions. Some examples are: a set of Radio Buttons coordinating with each other to ensure that only one of the buttons is "on" at any time; a Scrollbar coordinating with a Viewport to maintain a View of a window consistent with the shape and location of the Scrollbar's elevators; a parent window coordinating resizing and reshaping operations with its child windows to ensure that it always properly surrounds them. Patterns of communication within a behavioral composition are often repeated throughout a system with different participating objects. They represent a reusable domain protocol or programming paradigm. For the purposes of extending, modifying or re-using programs, it is clearly important to understand these behavioral compositions and the inter-object dependencies they imply.

While the recent literature recognizes the importance of inter-object behavior (expressed in terms of collaboration graphs [25], responsibilities [26], mechanisms [3], or views [20], for example), there is surprisingly little language support for its specification and abstraction. This means the existence of behavioral compositions in a system, and in particular the *behavioral dependencies* that they imply, cannot be easily inferred; they are spread across many class definitions in method implementations. This causes subsequent problems in the design, understanding and reuse of object-oriented software.

Addressing the issues of specifying the behavior of object-oriented systems, this paper proposes *Contracts*: a construct for the explicit specification of behavioral compositions. Contracts aim to formalize the collaboration and behavioral relationships between objects, thereby making precise the intuition "*no object is an island*". A contract defines a set of communicating *participants* and their *contractual obligations*. Contractual obligations extend the usual type signatures to include constraints on behavior which capture the behavioral dependencies between objects. As an example, the MVC paradigm[13] essentially defines behavioral composition in which Model, View and Controller objects participate to ensure that the View always reflects the state of the Model. The statement "a View displays itself in response to the message *update:* from its Model," defines a behavioral dependency between these participants. A contract also defines preconditions on participants required to establish the contract, and the *invariant* to be maintained by these participants.

Two important operations on contracts are *refinement* and *inclusion*. These provide two distinct means to express complex behavior in terms of simpler behavior. *Refinement* allows for the specialization of contractual obligations and invariants of contracts. *Inclusion* allows contracts to be composed from simpler sub-contracts. Through these mechanisms, contracts enable us to create and reuse large grain abstractions based on behavior, orthogonal to those based on structure defined by class constructs.

The specification of how a particular class implementation meets a participating object's obligations is declared via a *conformance declaration*. Conformance declarations define how classes can participate as "Black Boxes" in a contract. For abstract classes, conformance declarations describe how a participants obligations are apportioned between that class and its subclasses, thus describing how abstract classes can be reused as "White Boxes". Using contracts to specify behavioral compositions, and conformance declarations to specify how abstract classes implement the corresponding obligations, allows for the explicit representation of application frameworks [6] [12], which are considered important for reuse.

Behavioral compositions are created through the *instantiation* of contracts. This binds concrete objects, which satisfy the required contractual obligations to participants. Through new language constructs for the instantiation of contracts, the existence of behavioral compositions in a system can be made explicit, greatly assisting the comprehension of its architecture.

Explicitly specifying inter-object relationships is not entirely new. Class invariants in Eiffel [19] or "relations" in DSM [21] define constraints among class instances and methods. The contributions of contracts are: firstly, to generalize these to multi-object dependencies; secondly, to capture the *behavioral dependencies* between cooperating objects; and thirdly, provide a formalism for their abstraction. Other constructs to support generalized patterns of communication among multiple processes have also been proposed in the area of distributed computing and reactive systems [7, 11, 8]. Our experience in using contracts to design and specify a number of object-oriented applications suggests that an initial shift away from class-based design to one based on contracts greatly facilitates the early identification, abstraction and subsequent reuse of patterns of interactions between objects. Behavioral compositions expressed via contracts thus provide a rich vocabulary for *Interaction-Oriented Design.*

This paper is organized as follows. Section 2 introduces contracts and their composition and refinement through a number of examples, drawn from the domain of window-based user interfaces. Section 3 describes the binding of classes to participants via conformance declarations. Section 4 describes how contract instantiation makes explicit the behavioral compositions in applications. Finally, in section 5 we briefly describe how behavioral compositions and contracts may be used as the basis for an Interaction-Oriented design methodology. We also discuss some future research directions.

2 Contract Specification

A *contract* defines the behavioral composition of a set of communicating *participants*. Each contract specifies the following important aspects of behavioral compositions. Firstly, it identifies the participants in the behavioral composition and their contractual obligations. *Contractual obligations* consist of *type obligations*, where the participant must support certain variables and external interface, and *causal obligations*, where the participant must perform an ordered sequence of actions and make certain conditions true in response to these messages. Through causal obligations, contracts capture the behavioral dependencies between objects. Secondly, the contract defines invariants that participants cooperate to maintain. It also defines what actions should be initiated to resatisfy the the invariant, which as a matter of course during program execution will become false. Lastly, the contract specifies preconditions on participants to establish the contract and the methods which instantiate the contract.

As an example, consider the behavioral composition in which a *Subject* object, containing some data, and a collection of *View* objects, which represent that data graphically, say as a dial, histogram and counter (see Figure 1), cooperate so that at all times each *View* always reflects the current value of the *Subject*. This behavioral composition is described by the contract *SubjectView* shown in Figure 2.

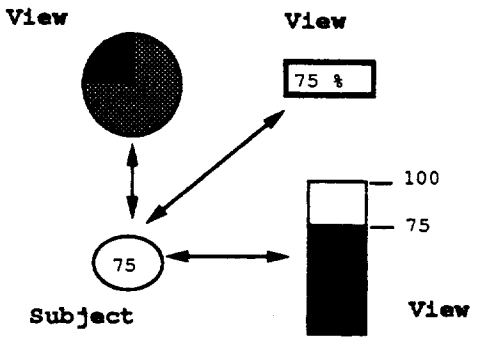

Figure 1: Subject and Views

```
contract SubjectView
    Subject supports [
        value : Value
        SetValue(val:Value) ↦ Δvalue {value = val}; Notify()
        GetValue() : Value ↦ return value
        Notify() ↦ ⟨|| v : v ∈ Views : v→Update() ⟩

        AttachView(v:View) ↦ {v ∈ Views}
        DetachView(v:View) ↦ {v ∉ Views}
        ]
    Views : Set(View) where each View supports [
        Update() ↦ Draw()
        Draw() ↦ Subject→GetValue() {View reflects Subject.value}
        SetSubject(s:Subject) ↦ {Subject = s}
        ]
    invariant
        Subject.SetValue(val) ↦ ⟨∀v : v ∈ Views : v reflects Subject.value⟩
    instantiation
        ⟨|| v : v ∈ Views : ⟨Subject→AttachView(v) || v→SetSubject(Subject)⟩ ⟩
end contract
```

Figure 2: Contract SubjectView

Contract *SubjectView* requires certain obligations of its participants *Subject* and the set of *Views*. Type obligations require a certain interface and data: *Subject*, for example, is required to support a variable of the unspecified type *Value* and the method *Notify()*. Causal obligations require that receipt of a message *leads to* certain behavior. For example, the expression $Draw() \mapsto Subject \rightarrow GetValue()$ specifies that each *View*, on receipt of a *Draw()* message, is required to behave in a way which leads to the sending of a *GetValue()* message to *Subject*.

Causal obligations are the essential feature of behavioral compositions. Through them we can infer that sending *Subject* the *Notify()* message will lead to the sending of a *Update()* message to each *View*. Each *View*, upon receipt of an *Update()* message, is required to behave in a way which leads to the sending of *Draw()* to itself. The *View* is then required behave in way which leads to the condition *View* **reflects** *Subject.value*[1] becoming true. Thus the contract, via the causal obligations, makes explicit the behavioral dependency between *Subject* and each *View*.

Participants in behavioral compositions often cooperate to maintain some invariant. However, during execution, this invariant can become false, requiring its resatisfaction. The invariant along with the actions which lead to its satisfaction appears in the the contract in the **invariant** section. For *SubjectView*, the expression $Subject.SetValue(val)$ preceding the $\mapsto$ symbol signifies the action that will lead to the satisfaction of the invariant "for each *View* in *Views*, *View* **reflects** *Subject.value*".

Finally, the **instantiation** statement specifies that to initiate this contract between a set of *View* objects and a *Subject* object, the methods *AttachView()* and *SetSubject()* must be executed with the appropriate arguments.

Contracts are defined in a high-level language which allows abstract description of behavior in terms of ordered sequences of actions to be performed and conditions to be made true. The language only supports the actions of sending a message, denoted by $P \rightarrow M$, and the setting of an instance variable v, denoted by Δv. The ordering of actions can be explicitly given by the operator ";", an **if-then-else** construct, or be left unspecified by the operator ||. The language also provides the construct $\langle o\ v : c : e \rangle$ for the repetition of an expression e separated by the operator o for all variables v which satisfy c. For example ⟨||v: v ∈ Views : v→Update()⟩ from $Notify()$ above, which should be read as for $v_1, v_2, v_3, ... \in Views$, $v_1 \rightarrow Update() || v_2 \rightarrow Update() || v_3 \rightarrow ...$. Conditions which participants are obliged to make true appear in parentheses {...} and are expressed as logical formula over the signatures of participants.

Note that each participant can refer to other participants

[1] reflects is some predicate over values of variables of *Subject* and *View*, the details of which do not concern us here.

in the contract. For example, in the *Notify()* body, messages are sent to *Views* and in the *Draw()* body a message is sent to *Subject*. The contract does not specify how these objects come into the scope of the action bodies. We assume that each participant maintains a local reference to each of the other participants. How such a reference is resolved, via an instance variable, parameter or global data structure is left to the implementation.

2.1 Contract Refinement and Inclusion

Contracts provide constructs for the refinement and inclusion of behavior defined in other contracts. *Refinement* allows for the specialization of contractual obligations and invariants of other contracts. *Inclusion* allows contracts to be composed from simpler contracts. These constructs provide two distinct means to specify complex behavioral compositions in terms of simpler ones.

2.1.1 Contract Refinement

Contracts are refined by either specializing the type of a participant, extending its actions, or deriving a new invariant which implies the old. Refinement of a contract essentially defines a more specialized behavioral composition. Refinement is expressed in a contract by the **refines** statement. Obligations which are refined in a contract, override those from the contract identified in the **refines** statement. All other obligations are inherited from this contract.

Consider the contract *ButtonGroup* in Figure 3 which specifies how a 'Radio Button' collection might be implemented. Each button in the collection independently reflects the value of the current setting, which we call the *State*. Depending whether or not the value represented by a button equals that of the *State*, the button would be visually represented as 'on' or 'off'. The relationship between the *State* and each button is basically a *SubjectView* relationship refined with the semantics specific to radio buttons, i.e. only one button can be on at any time and each button represents a distinct value.

Contract *ButtonGroup* refines contract *SubjectView* in a number of ways. Firstly, the definition of *Update()* is more specific than and overides that in *SubjectView*. A call to *Update()* will still lead to a call to *Draw()*, as prescribed in *SubjectView*, however other methods, specific to radio button semantics, are called in between. Secondly, additional obligations, for example the new method *Select()*, are required. Thirdly, the **instantiation** clause has also been extended by a precondition on the set of buttons. This ensures that each button in the set represents a unique value and thus only one button is "on" at a time. Lastly, the invariant implied by *ButtonGroup* implies that specified in *SubjectView*, but includes more conditions. Participant *State* inherits all its obligations from *Subject* in *SubjectView*. They are not refined further in *ButtonGroup*.

Specialization of behavioral compositions appear to be common in practice. Consider a contract *ParentChild* where a *Parent* window cooperates with a collection of *Child* windows to ensure that at all times the *Parent* displays an arbitrary layout of its *Child* windows. Possible specializations of contract *ParentChild* could require that all *Child* windows be enclosed in their *Parent*. Further specialization could require that all children be laid out vertically. Note that in contrast to the usual notions of specialization via sub-classing or sub-typing, specialization of behavioral compositions involves the obligations of multiple participants, which are specialized in concert.

2.1.2 Contract Inclusion

Often, the behavior of a subset of the participants in a complex behavioral composition may itself be described in terms of simpler compositions. The specification of this decomposition is expressed by including *sub-contracts* in a contract definition. These sub-contracts are denoted by the **include** statement which identifies a subset of a contract's participants and how they participate in the sub-contract. Participation in a sub-contract imposes additional obligations on participants over and above those defined in the contract. Rather than rewriting these obligations, they are implied by the included sub-contract. The **instantiation** and **invariant** clauses of a contract with sub-contracts are formed from the union of the corresponding statements from the sub-contracts.

Consider the behavioral composition, illustrated in figure 4, where an *Adjuster*, say the *UpMover* object, coordinates with a *Viewer* to display a portion of a *Picture*. A *Perspective* object is responsible for controlling which portion of the *Picture* is visible in the *Viewer* at any given time. Therefore, when the value of the *Perspective* changes, after being acted upon by the *UpMover*, the *Viewer* must be updated. The relationship between the *Perspective* and the *Viewer* is exactly that between a *Subject* and a *View* in the contract *SubjectView*. Clearly we wish to exploit this similarity by including the behavior defined by *SubjectView* as part of some more complex behavior.

The contract *AdjustView* in Figure 5 defines this behavioral composition. It is composed from sub-contracts *SubjectView* and a new contract *ParentChild*. The latter defines a behavioral dependency between a *Parent* graphical object, a window say, and the layout of its *Child* components,i.e. its sub-windows. It is defined in section 3.

Contract *AdjustView* captures the following behavioral dependencies amongst its participants. When the *Ad-*

```
contract ButtonGroup
    refines
        SubjectView(Views = Buttons, Subject = State)
    State supports [ ]
    Buttons : Setof(Button) where each Button supports [
        myvalue : Value
        chosen : Boolean

        Select() ↦ State→SetValue(myvalue)
        Refresh() ↦ Draw()
        Update() ↦ if State→GetValue() = myvalue then
                    Choose() else UnCoose()
        Choose() ↦ Δchosen {chosen = true}; Refresh()
        UnChoose() ↦ Δchosen {chosen = false}; Refresh()
        ]
    invariant
        Button.Select() ↦ (∀b : b ∈ Buttons : b.chosen ⇔ b.myvalue = State.value)∧
                           (∃!b : b ∈ Buttons : b.chosen)
    instantiation
        {∀b1, b2 : b1, b2 ∈ Buttons : b1 ≠ b2 ⇒ b1.myvalue ≠ b2.myvalue}∧
            {∃!b : b ∈ Buttons : b.myvalue = State.value}
        (|| b : b ∈ Buttons : State→AttachView(b) || b→SetSubject(State) )
endcontract
```

Figure 3: Contract ButtonGroup

juster receives an *Activate()* message, it changes its adjustment accordingly. This adjustment is passed to the *Viewer* via the *Adjust()* message. The *Viewer* computes a new value for the *Perspective* based on the total *shape* of the *Picture* and the adjustment, and then calls *Notify()*. The inclusion of the *SubjectView* contract implies the *Viewer* is a *View* of *Perspective*. It thus receives an *Update()* message, whereupon it calls *Draw()* on itself. The *SubjectView* sub-contract implies the invariant *Viewer* **reflects** *Perspective.value*.

At this high level of specification we are not concerned with the precise details of the adjustment update or the function used to compute the new value of the perspective (denoted by *fcn*). These details would be filled in during further design refinement or left until implementation.

The mechanisms of inclusion and refinement can be simply combined when needed. Figure 6 defines the contract *AdjustViewWithFeedback*, which is a refinement of contract *AdjustView* extended with the new obligations that the *Adjuster* also reflect the current value of the *Perspective*. For example, a Scrollbar elevator reflects the position of the currently visible portion of a picture with respect to the total size of the picture (see Figure 4). This extended behavior can be specified by including a *SubjectView* contract between the *Perspective* and *Adjuster*. A result of this specialization is that *Adjuster* must now support *Update()* messages.

3 Contract Conformance : Satisfying contractual obligations

Contracts are defined independently of classes. However, class implementations must ultimately be mapped to participant specifications. This mapping is specified through *conformance declarations*.

A *conformance declaration* is a specification (ideally verifiable) of how a class, and thus its instances, supports the role of a participant in a contract. It describes the variables and methods the class provides to the role. Abstractly, a conformance declaration contains a set of bindings of the form $a : \sigma \leftarrow b : \tau$, which maps an identifier b of type τ defined in a class, to an identifier a of type σ in a participant. In our examples, class identifiers are always mapped to identically named participant identifiers (though this need not be the case in practice) and the explicit detailing of the mapping is omitted.

Intuitively a class conforms to a participant definition in a contract only if its methods and instance variables satisfy both the typing and causal obligations required by the participant definition. However, in defining con-

```
contract AdjustView
    Viewer supports [
        Adjust(a:Adjustment) ↦ Perspective→SetValue(fcn(Picture→getShape(),a))
        ]
    Adjuster supports [
        a : Adjustment
        Attach(v:Viewer) ↦ {Viewer = v}
        Activate() ↦ Δa; Viewer →Adjust(a)
        ]
    Perspective supports [ ]
    Picture supports [
        shape : Shape
        getShape() : Shape ↦ return shape
        ]
    includes
        SubjectView(Views = {Viewer}, Subject = Perspective)
        ParentChild(Children = {Picture}, Parent = Viewer)
    instantiation
        Adjuster →Attach(Viewer)
end contract
```

Figure 5: Contract AdjustView

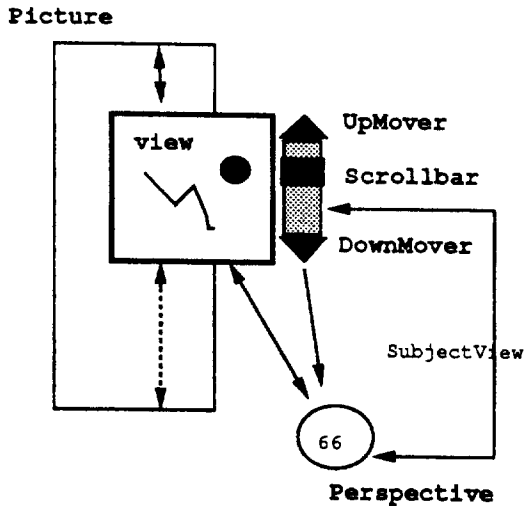

Figure 4: Adjusters and Views

tract conformance we must allow for the fulfillment of these obligations to be distributed among the implementations of an abstract class and its subclasses. This is accomplished by declaring explicitly in a conformance declaration which obligations are fulfilled by the implementation of the abstract class, what is implemented in the subclasses, and what relationships exist between the implementations, if any. This aspect of conformance declaration is crucial for class reuse.

```
contract AdjustViewWithFeedback
    refines AdjustView
    includes
        SubjectView(Views = {Adjuster},
                    Subject = Perspective)
end contract
```

Figure 6: Contract AdjustViewWithFeedback

Conformance declarations are an important part of the documentation of an application. A complex class may conform to participants in multiple contracts. This results in a large number of methods in its public interface. The conformance declarations factor this large interface into meaningful related subsets. For example, in the InterViews[18] library, class *Interactor* which has over a 100 methods, conforms to the participants of many contracts including *View* in contract *SubjectView* and *Child* in *ParentChild*.

3.1 Example: Graphical composition

To illustrate how conformance declarations can describe the distribution of a participant's implementation along an inheritance hierarchy, we use the contract *ParentChild* (Figure 7) and associated conformance declarations.

```
class Scene conforms to Parent in ParentChild
    Scene supports
        Change(c:Child) ↦ if propagate then
                DoChange(c) else Rearrange() ;
        Reshape();
        Reconfig();
        Place(c:Child, loc:Location);
    requires subclass to support
        Rearrange();
        DoChange(c:Child) ↦ Reconfig();
end conformance
```

Figure 8:

Contract *ParentChild*[2] defines a behavioral composition where a *Parent* graphical object, say a window, cooperates with a set *Child* graphical objects, say sub-windows, to manage their respective layouts, shapes and display. One aim of this contract is to ensure that the *Parent* shape always reflects the shape of its *Children* by enlarging its shape, or rearranging or reshaping its children. Note that for purposes of this exposition, we have only presented that portion of *ParentChild* relevant to maintaining the relative shapes of the *Parent* and *Children*.

In the InterViews class library, graphical composition objects are members of the abstract class *Scene*. The conformance declaration in Figure 8 specifies that *Scene* conforms to the *Parent* participant in the contract[3]. The methods that *Scene* implements are listed in the declaration. However, this declaration makes it clear that *Scene* is an abstract class, and does not conform completely to the role of *Parent*. Firstly, the implementation of the *Change()* method does not fully conform to the obligation defined in the contract. The call to *Reconfig()* is deferred, via *DoChange()*, to the implementation of *Scene*'s subclasses. When an implementation does not fully conform to the specification, the differences are described in terms of the original specification. This makes explicit the inter-class dependencies between subclass and abstract class in the context of the contract.

Secondly, subclasses of *Scene* customize the graphical layout of their children. Class *Deck* places its components one on top of the other, while class *Box* places its components one alongside the other. To allow for this flexibility, the implementation of *Rearrange()* function is deferred from *Scene* to its subclasses.

[2] This specification is a simplification of the complex behavioral relationship between classes Interactor and Scene in InterViews 2.6.

[3] For clarity, we have slightly renamed and simplified the details of the conformance of Scene as implemented in InterViews.

```
class Box conforms to Parent in ParentChild
    inherits from Scene;
    Box supports
        Rearrange() ↦
            (||c: c ∈ Children: PlaceElement(c));
        DoChange();
    requires subclass to support
        PlaceElement(c:Child) ↦ Δloc; Place(c, loc);
end conformance

class VBox conforms to  Parent in ParentChild
    inherits from Box;
    VBox supports
        PlaceElement(c:Child);
end conformance
```

Figure 9:

The conformance of the *Box* subclass is described in Figure 9. It has implementations for all the methods required by the *Parent* participant. Some of these are inherited from *Scene*, and others, including *Rearrange()* and *DoChange()*, are implemented locally. However, as with *Change()* above, the implementation of *Rearrange()* does not fully conform to the specification. This is because *Box* is itself an abstract class. The component layout and placement algorithm is deferred to the concrete subclasses, *HBox*(horizontal layout) and *VBox*(vertical layout). These classes are required to implement *PlaceElement()* to accomplish this task. Finally, the last declaration in Figure 9 declares that the class *VBox* completely conforms to the role of *Parent* in the contract. Thus the implementation of *Parent* is spread over these three classes (Figure 10).

Understanding the implementation dependencies between abstract classes and their subclasses is central to the successful use of application frameworks. Frameworks such as Choices [5] for operating systems, InterViews[18] for user interfaces, Unidraw[23] for graphics editors and MacApp [1] for Mactintosh applications, are libraries of classes which implement skeleton portions of applications which can be fleshed out to build a particular application. The skeletons are typically customized by subclassing particular abstract classes and providing the required implementation in the new subclass. However, it has been our experience that it is often difficult to identify which abstract class to subclass and which method to override, when customizing a particular behavior. Detailed knowledge of the library implementation is required.

We have identified many examples of behavioral compo-

```
contract ParentChild
    Parent supports [
        shape : Shape
        propagate : Boolean

        Reconfig() ↦ {shape = ⟨+c : c ∈ Children : c.shape⟩}
        Change(c:Child) ↦ if propagate then Reconfig() else Rearrange()
        Reshape(s:Shape) ↦ Δshape {shape = s}; Rearrange(); Reconfig()
        Rearrange() ↦ (|| c : c ∈ Children : Δloc; Place(c,loc) )
        Place(c:Child, loc:Location) ↦ c→SetLocation(loc)
        ...]
    Children : Set(Child) where each Child supports [
        shape : Shape;
        location : Location;

        Reshape(s:Shape) ↦ Δshape {shape = s} Parent→Change(self)
        SetLocation(loc:Location) ↦ {location = loc}
        ...]
    invariant
        Parent.Reshape() ∨ Child.Reshape() ↦ {Parent.shape = ⟨+c : c ∈ Children : c.shape⟩}
end contract
```

Figure 7: Contract ParentChild

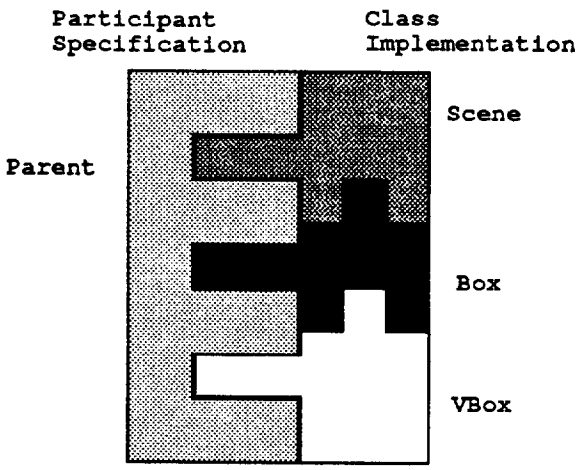

Figure 10: Conformance to Parent

sitions in InterViews and Unidraw and expressed them as contracts. It is our view that the abstract behavior represented by an application framework should be formally described in terms of contracts. This would provide the application builder with a vocabulary with which to describe their application. The classes implemented by the framework should be described in terms of conformance declarations which relate the library implementation to the associated behavior. As illustrated by the above example, the conformance declarations will also show the hooks for customization. We believe that the contract approach will greatly assist in the understanding and use of these "White Box" frameworks.

Note that causal obligations specify the minimum actions required by the contract to be performed in some method implementation. Typically, a method implementation may implement more than required by the contract. Indeed, the same method implementation may conform to more than one action body if the class to which the method is attached participates in more than one contract.

4 Contract Instantiation

We have shown how behavioral compositions are specified by contracts, and how conformance declarations determine the legal types of objects allowed to participate in a particular composition. What remains to be discussed is the creation of behavioral compositions within an application.

Behavioral compositions are created by *instantiating* contracts. This requires identifying objects as participants in the desired contract, and then establishing the contract via the methods specified in the contract's **instantiation** statement. These methods typically ensure that participant objects have references to other participants

and that initial conditions are set up.

We envisage programming language constructs that make explicit the creation of behavioral compositions via the instantiation of contracts, rather than calling the methods of the instantiation statement directly. These constructs would have the same name as the contract and the participating objects would be supplied as parameters. The advantage of such constructs is that they explicitly show the behavioral composition, the objects involved in it, and where it is established. Our initial experience providing such constructs (in the form of procedure calls) within applications and classes in the InterViews toolkit suggests that they make for better and easier understanding of the application architecture. They facilitate the replacement of objects by functionally equivalent objects when a change is needed. This contrasts with our previous experience where the creation of behavioral compositions was either hidden inside class constructors or implicit in certain sequences of method calls.

As an example of the descriptive power of explicit contract instantiation, consider the partial class definition of a graphics editor (Figure 11) which has four instance variables and a function *create()* which builds the graphical view of the editor. We make extensive use of the contracts discussed earlier and assume all classes satisfy the obligations of the relevant participants. This example makes use of a new contract called *DirectManipulation*, adapted from [23]. Rather than give its full description, we give a brief summary, identifying participants in *italics* font.

The *DirectManipulation* contract establishes this behavioral composition between four participants, a *Tool*, a *Command*, a *DrawingArea*, and a *Manipulator*. To edit the picture, the editor provides a number of tools to manipulate graphical components such as lines and rectangles. Each *Tool* provides a *Command*, say to move or create graphical components. To allow for direct manipulation of the *Tool* on the *DrawingArea*, to select the position and size of a new rectangle for example, a *Manipulator*, such as a 'rubber rectangle', tracks the mouse events and updates an appropriate representation on the *DrawingArea*. When the desired size and shape have been selected, the *Tool* uses the information in the *Manipulator*'s state to execute the *Command* on the *DrawingArea*.

In Figure 11, binding of objects of particular participants is implicit in the ordering of objects in the **instantiate** statement. This ordering is given as a comment at the top of the figure.

Some of the relationships created by this class are given by the dashed lines in Figure 12 where **DM**, **SV**, *AV*, and **AVF** stand for *DirectManipualtion*, *SubjectView*, *AdjustView* and *AdjustViewWithFeedback* respectively. Of course, in practice, there would exist many more behavioral compositions defined by this class: not only those which provide the full functionality of the editor, but also many based on *ParentChild* and its specializations to lay out all the graphical objects.

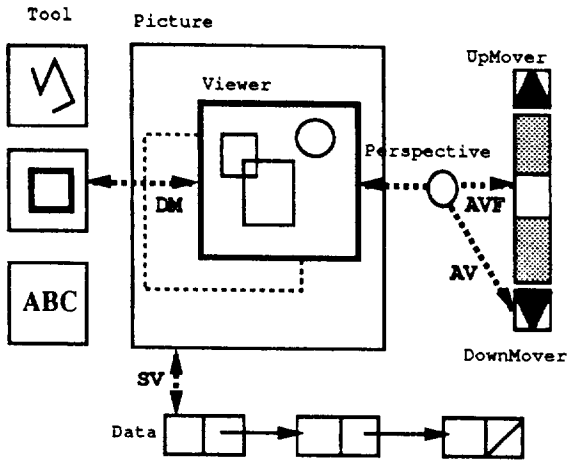

Figure 12:

The contracts define the building blocks of the application domain. The explicit instantiation of contracts clearly show which blocks are being used to build a particular application. The important object relationships are visible directly in the application code. We believe that this approach leads to easier application development and maintenance.

5 Conclusion

We have presented contracts for the abstraction, composition and specialization of behavioral compositions. Through conformance declarations we explicitly state how a class and possibly its subclasses support the role of a participant, and through contract instantiation make explicit the creation of behavioral compositions. We have described how these features each contribute to program design, understanding, maintenance and reuse.

We are currently building on the ideas presented in this paper in the areas of formal theory, language support, design methodology and tools. We are developing a formal semantics of the contract construct and a formal correctness criterion to check conformance declarations at compile time. The latter is based on extended definitions of type and type conformance which allow for causal obligations. We envisage direct programming language support for the definition and instantiation of contracts and the monitoring of contract invariants.

```
// SubjectView(Subject,View)
// AdjustView(Viewer, Picture, Adjuster, Perspective)
// AdjustViewWithFeedBack(Viewer, Picture, Adjuster, Perspective)
// DirectManipulation(Tool, Manipulator, DrawingArea, Command)

class EditorFramework has parts
    picture : PictureView;
    frame : ViewPort;
    pictureData : PictureData;
    perspective : Perspective;

    void create() {
    instantiate AdjustView(frame, picture, new UpMover, perspective);
    instantiate AdjustView(frame, picture, new DownMover, perspective);
    instantiate AdjustViewWithFeedBack(frame, picture, new VScrollbar, perspective);
    instantiate SubjectView(pictureData, {picture});
    instantiate DirectManipulation(new RectTool, new RubberRectangle, picture, new NewRectCommand);
    instantiate DirectManipulation(new LineTool, new RubberLine, picture, new NewLineCommand);
    :
    }
end class  EditorFramework
```

Figure 11:

Behavioral compositions provide a rich, high-level vocabulary for design which shifts emphasis away from classes and implementation details, to interactions between objects. We call this paradigm shift *Interaction-Oriented* design. In Interaction-Oriented design, interactions between objects are first class entities in the design space. Our experience in using contracts for the design of a number published problems, the Car Controller[24], Data-Logging Buoys[4], suggest that this shift of emphasis greatly facilitates the early identification and abstraction of patterns of interaction between objects and their reuse. Design then becomes a two-step process. Firstly, behavioral compositions are defined via contracts. Then contracts are factored into class definitions and hierarchies via conformance declarations. With Interaction-Oriented design, the specification of a class becomes spread over a number of contracts and conformance declarations, and is not localized to one class definition. This Interaction-Oriented approach to the design of object-oriented systems will be the subject of a forthcoming paper[10].

Our experience also suggests that a lack of a suitable vocabulary for expressing behavioral compositions can lead to non-optimal programming techniques where, for example, many spurious classes are created. A representative example we have encountered is when a programmer, who did not fully understand the behavioral compositions in InterViews, subclassed a *VerticalScrollbar* class to provide an additional *UpMover* and *DownMover*. This subclass is totally unnecessary. Equally unnecessary was their next attempt which involved the new class *VerticalScrollbarWithMovers* which contained the *VerticalScrollbar*, *UpMover* and *DownMover* as instance variables. As we have seen in the editor framework, exactly the the same functionality can be achieved through instantiating behavioral compositions based on *AdjustView* and *AdjustViewWithFeedback*. Building applications from behavioral compositions via the instantiation of contracts is a form of reuse which contrasts with those based on "buying" or "inheriting" behavior discussed in [22].

Application frameworks[12], such as Unidraw, define domain specific application skeletons in terms of interactions, that is behavioral compositions, between abstract classes. It is evident that achieving wide applicability for a framework is difficult as it requires careful class and interface design. It is also evident from recent published examples, for example InterViews and Unidraw, that the behavioral dependencies involving multiple classes, such as those defined by DirectManipulation, are an important part of application frameworks. Behavioral compositions expressed via contracts, provide a means to describe and specify these relationships and factor out some of this complexity. The examples of contracts introduced in this paper essentially provide a vocabulary for the design of window based application frameworks.

This Interaction-Oriented approach to design is currently being investigated in Sculptor, a reuse assistant for window based interfaces[9]. In Sculptor, portions of application designs are expressed abstractly in terms of predefined behavioral compositions. Once an abstract design is completed, each participant is required to participate in a number of different contracts. This constrains the set of candidate classes which can implement that participant. This information, along with that in conformance declarations, is used to help retrieve candidate classes from a class repository which can be reused as "Black Boxes", or if there are none, abstract classes which can be reused as "White Boxes".

Constructs to express behavioral composition are being incorporated into the Demeter [14] [17] language which currently provides high level constructs for structural composition. The Demeter CASE system provides a programming language independent development environment for object-oriented software. This system is being extended to support continued system development and maintenance [15] [16]. The contract mechanism provides essential information about inter-class dependencies and module coupling which is needed for these extensions.

6 Acknowledgements

The authors wish to thank Harold Ossher, Brent Hailpern and Karl Lieberherr.

References

[1] Apple Computer Inc. *Macintosh Programmers Workshop Pascal 3.0 Reference.* Cupertino, California, 1989.

[2] Kent Beck and Ward Cunningham. A laboratory for teaching object oriented thinking. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 1–6, ACM, New Orleans, LA, 1989.

[3] Grady Booch. *Object-Oriented Design.* Benjamin/Cummings Publishing Comapany, Redwood City, California, 1991.

[4] Grady Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, SE-12(2), Feb. 1986.

[5] Roy H. Campbell, Vincent F. Russo, and Gary Johnston. Choices: the design of a multi-processor operating system. In *Usenix C++ Workshop*, pages 109–123, Santa Fe, New Mexico, November 1987.

[6] L. Peter Deutsch. Design reuse and frameworks in the smalltalk-80 system. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability. Volume II. Applications and Experience*, ACM, 1989.

[7] M. Evangelist, N. Francez, and S. Katz. Multiparty interactions for interprocess communication and synchronization. *IEEE Transactions on Software Engineering*, 15(11):1417–1426, November 1989.

[8] N. Francez, B. Hailpern, and Gadi Taubenfeld. Script: a communication abstraction mechanism, and its verification. *Science of Computer Programming*, 6(1):35–88, 1986.

[9] D. Gangopadhyay and A.R. Helm. *A Model-Driven Approach for the Reuse of Classes from Domain Specific Object-Oriented Class Repositories.* Technical Report RC14510, IBM Research Division, March 1989.

[10] Richard Helm and Ian Holland. *An Interaction Oriented approach to design using contracts.* Technical Report In preparation, IBM Research Division, 1990.

[11] H. Jarvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systa. Object-oriented specification of reactive systems. In *International Conference on Software Engineering*, pages 63-71, Nice, France, 1990.

[12] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[13] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 26–49, August/Spetember 1988.

[14] Karl Lieberherr. Object-oriented programming with class dictionaries. *Journal on Lisp and Symbolic Computation*, 1(2):185–212, 1988.

[15] Karl J. Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, 38–48, September 1989.

[16] Karl J. Lieberherr and Ian Holland. Tools for preventive software maintenance. In *Conference on Software Maintenance*, pages 2–13, IEEE Press, Miami Beach, Florida, October 16-19, 1989.

[17] Karl J. Lieberherr and Arthur J. Riel. Demeter: a CASE study of software growth through parameterized classes. *Journal of Object-Oriented Programming*, 1(3):8–22, August, September 1988. A shorter version of this paper was presented at the *10th International Conference on Software Engineering, Singapore, April 1988, IEEE Press*, pages 254-264.

[18] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with interviews. *IEEE Computer Magazine*, 8–22, February 1989.

[19] Bertrand Meyer. *Object-Oriented Software Construction. Series in Computer Science*, Prentice Hall International, 1988.

[20] M. B. Rosson and J. M. Carroll. *Deliberated Evolution: Stalking the View Matcher in Design Space.* Technical Report Submitted for Publication, IBM Research Division, 1990.

[21] Ashwin Shah, Jung Hamel, and Renee Borsari. Dsm: an object-relationship modeling language. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 191–202, ACM, New Orleans, LA, 1989.

[22] David Tanezer, Murthy Ganti, and Sunil Podar. Object-oriented software reuse: the yoyo problem. *Journal*

of Object-Oriented Programming, 2(3):30–36, September/October 1989.

[23] John M. Vlissides and Mark A. Linton. Unidraw: a framework for building domain-specific graphical editors. In *ACM/SIGGRAPH/SIGCHI User Interface Software Technologies Conference*, ACM, Williamsburg, VA, February 1989.

[24] Anthony I. Wasserman. An object-oriented structured design method for code generation. *ACM/SIGSOFT, Software Engineering Notes*, 14(1), January 1989.

[25] Rebecca Wirfs-Brock and Ralph E. Johnson. A survey of current research in object-oriented design. *Communications of the ACM*, 1990 To Appear.

[26] Rebecca Wirfs-Brock and Brian Wilkerson. Object-oriented design: a responsibility-driven approach. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 71–76, ACM, New Orleans, LA, 1989.