Specifying reusable components using Contracts

Ian M. Holland[1]

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

Abstract. Contracts were introduced by Helm et al. as a high level construct for explicitly specifying interactions among groups of objects. This paper describes further developments and application of the Contract construct. We show how Contracts can be used to represent classic algorithms as large grained reusable object oriented abstractions, how these algorithms can be customized through Contract *refinement* and how they are reused through Contract *conformance*. The example algorithm used throughout is the classic graph depth first traversal algorithm. This algorithm is represented as a Contract which is then refined to specify algorithms which number connected regions of graphs and which check graphs for cycles. Changes to the Contract language are introduced and we discuss some new problems resulting from the simultaneous reuse of related contracts.

1 Introduction

Contracts were introduced by Helm et al. [9] as a construct for explicitly specifying interactions among groups of objects. The objects in such a group, called a behavioral composition, work together to accomplish a particular task or to maintain some invariant. Each object provides some of the required functionality and participates in a communication protocol with the other members of the group. Understanding the different roles individual objects play in behavioral compositions, and understanding how the objects collaborate, is crucial for object oriented software design and reuse [2, 27, 26, 23]. The Contract language addresses this crucial need by enabling the software developer to explicitly specify the different object roles in context.

The development of the Contract construct was motivated by our interest in describing the complex behavioral compositions found in the InterViews[17] C++ user interface framework. We have since extended our research to include business/MIS applications and algorithmic reuse. This paper describes results from the last of these areas. The two main contributions are:

- to present the further development and application of the Contract construct
- and to present a solution to the problem of representing and reusing algorithmic programming clichés.

We will show how Contracts can be used to represent variations of the classic depth first traversal algorithm as large grained reusable object oriented abstractions. Graph algorithms such as, search, node ordering, cycle checking and path finding can all be represented as variants of depth first traversal. Each variant requires

[1] Work supported in part by IBM T.J. Watson Research Center and in part by the National Science Foundation under grants CCR-9102578 (Software Engineering) and CDA-9015692 (Research Instrumentation).

customization of both the basic graph data structures and functionality. Through data and function abstraction, the core depthfirst algorithm will be isolated and represented in contract form. Each algorithmic variation is then expressed as a customization of this core using contract *refinement* and finally used in an application through Contract *conformance*.

Our combined functional and data abstraction approach is a generalization of the reuse technique described by Bishop[5]. She uses a data abstraction approach to isolate classic loop algorithms from data structure implementation details, special language control structures and deft programming techniques. She claims that these loop algorithms can then be packaged as reusable software components.

Contracts support the reuse and refinement of larger grained software components. They can be used to specify reusable abstractions composed of many interrelated subcomponents, some, or all, of which can be specialized to form new more detailed abstractions. The basic underlying model remains object oriented, consisting of objects, message passing, methods and inheritance.

The use of an object oriented model to represent and manipulate program fragments and programming knowledge, such as knowledge of data structures and graph traversal algorithms, differentiates this work from previous approaches. For example, the PARIS[14] specification language, the PROUST[11] intelligent tutoring system and the Programmers Apprentice project[25] each represent and manipulate programs at the meta-level. However, their representation languages reflect a procedural view of software. The Plan Calculus graphical notation of the Programmers Apprentice project is used to represent program clichés in the form of graphs similar to data flow diagrams. This is a consequence of the view of Rich and Shrobe[24]: "We view programs as being constructed of input-output segments connected by control and data flow." Clearly, new formalisms are required to describe object oriented representations of program clichés. We claim that Contracts fulfill that requirement.

The recent work by Rao[22] introducing *implementational reflection* provides further motivation for the development of a construct to represent, and reason about, object oriented software. He defines *implementational reflection* as reflection involving inspection and/or manipulation of the implementational structures of other systems used by a program. He argues that a software system's implementation architecture be made explicit and open, thus allowing customization. Currently, the architectural information is represented informally in the form of English descriptions of *meta-level* objects and their interactions, called *protocols*. The Contract language can support the formal description of these protocols which can then lead to the development of tools to manipulate the information.

We are developing a tool to support the direct reuse of Contracts for C++ application development. This has also resulted in some additions to the language. The main differences between the notation of [9] (which I shall henceforth refer to as $Contract_0$) and that used here are: an explicit distinction between the participating objects and their corresponding type obligations[2], an extension to the language used to express causal obligations, and the introduction of two keywords *public* and *default*.

[2]This notational change was suggested by Karl Lieberherr.

The last issue that will be briefly addressed is the problem of reusing conflicting components. This occurs at the macro level when attempting to reuse two incompatible class libraries[3]. It occurs at the micro level when two related contracts are used in a single application instantiation. We will address one aspect of this micro level problem which occurs when two variants of the same basic contract are reused in a single instantiation.

The layout of the paper is as follows: The next section describes the aspects of the contract mechanism that were introduced in [9]. Section 3 describes the depth first algorithm and its contract representation. Section 4 defines a number of specializations of this contract. Section 5 describes how these variations are used in an application, Section 6 discusses some implementation concerns and section 7 concludes.

2 Contracts revisited

Contracts are proposed as a high level notation for describing the different paradigms encoded in frameworks such as the InterViews[17] C++ user interface class library. The user interface domain is rife with behavioral compositions which determine how graphical objects remain consistent with the data they represent, how scrollbars affect the contents of a window, how radio buttons work, and how the selection of a menu item causes the execution of an application function, etc. Some of these are specified in $Contract_0$. Understanding these behavioral compositions and identifying the classes which implement them is the first step to using the framework.

A Contract specifies behavioral compositions in terms of: the participating objects, the contractual obligations for each participant, invariants to be maintained by the participants and the methods which instantiate the Contract. The 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 sequences of actions, including sending messages to the other participants. Since Contracts are designed to represent message passing protocols between objects they are necessarily imperative in nature. This contrasts with the declarative nature of other object oriented formalisms, e.g. Features[13].

The behavior of a participant is specified by its obligations. However, since an object may participate in many Contracts, its total behavior may be quite complex. Each Contract factors this behavior into separate contexts, which can be independently understood and modified. This is similar to the approach of Hailpern and Ossher[8] who separate an object's total interface into subsets called *Views*. Restricting clients to using selective predefined views adds an additional layer of information hiding. Others factor an objects interface into subsets called *Roles* [20, 1], where a role represents a discrete stage in an object's life cycle.

The behavior of one or all participants of a Contract can be specialized and extended through the *Contract refinement* and *Contract inclusion* mechanisms. The inclusion and refinement mechanisms enable new contracts to be defined in terms of previously defined contracts. This ability to isolate, independently modify, and combine cohesive units of behavior is the important feature of Contract use which will be exploited in the next sections. In this sense, the Contract language can be considered a module interconnection language [21] for object oriented software.

```
contract UndirectedGraph
   participants
      graph : Graph;
      vertices : Set(Vertex);

   Graph supports [
      vertices : Set(Vertex);
      public insert(v : Vertex) : void;
      :
      ]
   Vertex supports [
      neighbors : Set(Vertex);
      :
      ]
   invariant
      graph.vertices = vertices ∧
      ⟨∀v ∈ vertices : v.neighbors ⊆ vertices⟩ ∧
      ⟨∀v ∈ vertices ⇒ ⟨∀v2 ∈ v.neighbors : v ∈ v2.neighbors⟩⟩
end contract
```

Figure 1:

Many other researchers in the object oriented field have recognized that the class abstraction is too fine grained for large scale object oriented development and provide constructs for grouping classes together, e.g. Class dictionaries [16], Mechanisms [6], Subsystems [27], Clusters [18], Features [13], and Frameworks [7, 10]. Johnson and Foote [10] state that such a group of abstract classes, can be used to express an abstract design. Reusing this abstract design involves not only reuse of individual classes but also reuse of the relationships and interconnections between these classes. Large grained reusable components which include aspects of control flow as well as functionality, are also called reusable software architectures [12] and program schemas [15, 14]. The reuse of design level information has been identified as critical to realizing the promise of reuse [4]. However, as Biggerstaff and Richter [4] state, the designs must be represented in a form that is machine processable. This is one of the primary motivations for the development of the Contract formalism.

The examples in this paper are based on extensions of the contract partially described in figure 1. The basic undirected graph data structure, which will be traversed by the algorithms defined in sections 3 and 4, is described by the UndirectedGraph contract between two kinds of participants: a graph object and a set of vertices. The contractual obligations on the graph participant specify an instance variable to store its vertices and a function insert (to insert each vertex). We require an adjacency list representation for the graph data structure as a basis for the traversal algorithms. Therefore, the obligations on each vertex participant include an instance variable to hold the set of its neighbors.

The *public* keyword in the declaration of the insert function specifies that the function may be invoked by functions external to the contract. The default visibility for functions is *private*, i.e. they may only be invoked by other functions defined for the contract.

For the purposes of this paper, the most important part of figure 1 is the contract invariant. It states: the vertices of the contract are precisely those stored in the instance variable of graph; the neighbor set of each vertex is a subset of the vertices and lastly; each vertex is an element of the neighbor set of each of its own successors (i.e. the edges are undirected). In other words, the invariant ensures that the participating objects do indeed represent an undirected graph.

The first benefit of specifying this information in Contract form is that the specification of the two components of the graph data structure (i.e. the Graph and Vertex) can be expressed in a single syntactic unit, the UndirectedGraph contract. This unit can be directly manipulated and used with the contract inclusion and refinement mechanisms to construct new contracts. In fact, section 3 will include UndirectedGraph for the definition of the Depthfirst contract. A second benefit is that the invariant allows one to express constraints on the *combined* behavior and structure of the contract participants. In UndirectedGraph, the invariant involves terms from *both* the Graph and Vertex participant specifications.

3 Depth first traversal

A software developer's programming knowledge includes many programming tricks, paradigms, algorithms, implementation techniques and problem solving approaches. Typical components of this mental toolkit is knowledge of data structures and their associated algorithms. However, each time the developer needs to use this knowledge they must reimplement, and sometimes reinvent, the details. Representing this type of information, sometimes called programming clichés[25], and supporting its direct reuse has been a much sought after goal.

In this section, we will use Contracts to describe the representation of a generic depth first traversal algorithm over an undirected graph in adjacency list format. There are a number of benefits which result from representing generic algorithms such as this in a directly reusable form. Firstly, correctness and efficiency results proven for the generic representation will transfer to the specializations. Secondly, the representation makes the architecture of the algorithm explicit, names each of the constituent components and shows where customization is expected. This forms a meta-level programming language for considering and expressing the possible variations of the generic algorithm.

Figures 2a-2c show pseudo-code descriptions for three graph algorithms based on depth first traversal.

- Dft-Number: Number each vertex of the graph in the order it was visited by the traversal.

```
/* Graph in adjacency list form */
/* Number each vertex in the order traversed */
for each v ∈ Vertices v.mark := unvisited  /* initialize each graph vertex as unvisted */
DNumber := 0                               /* initialize the counter */
for each v ∈ Vertices Depthfirst(v)        /* traverse each vertex */

proc Depthfirst(v : Vertex)
    if v.mark = unvisited then             /* don't visit the same vertex twice */
        v.mark := visited
        v.dft-number := DNumber            /* assign the counter value to the vertex */
        DNumber := DNumber + 1             /* increment the counter */
        for each v2 ∈ v.Neighbors Depthfirst(v2)
                                           /* traverse each of the vertex neighbors */
    endif
endproc
```

Figure 2a: Dft-Number

- Dft-Connected: Number all the vertices of connected regions equally.
- Dft-CycleCheck: Check for cycles in the graph.

The informal procedural notation is typical of that used in standard data structures and algorithm texts. Each algorithm follows the same basic form, but the implementation of Depthfirst is slightly different. The other main variation is in the use of the variables required by each algorithm, DNumber, CNumber and Cycle.

We show in figure 3 the results of isolating the commonality of these algorithms in a generic form which provides the hooks for future customization. The creation of this generic form went through a number of refinements. We abstracted from a few of the algorithmic variants, then tested the result on the remainder adjusting the abstraction when necessary.

The process begins with the dissection of the algorithms of figure 2 to identify and name the discrete functional components. The common structure of these algorithms consists of:

- an initialization part,
- an outside loop over the elements of the vertex set invoking Depthfirst on each element.
- and then the Depthfirst procedure.

Each of the parts differ slightly in a number of the statements used across the three algorithms. The differing statements will be removed and replaced by calls to new functions. The final implementations of these added functions will then be different for each variation. We use data abstraction when assigning the responsibility for

```
/* Assign the same number to vertices in connected regions */
for each v ∈ Vertices  v.mark := unvisited /* initialize each graph vertex as unvisted */
CNumber := 0                               /* initialize connected component counter */
for each v ∈ Vertices do
    if v.mark = unvisited                  /* has the vertex been traversed already ? */
            CNumber := CNumber + 1
    Depthfirst(v)                          /* each time this call to Depthfirst returns,*/
enddo                                      /* an entire connected region has been */
                                           /* traversed */

proc Depthfirst(v : Vertex)
    if v.mark = unvisited then
        v.mark := visited
        v.c-number := CNumber
        for each v2 ∈ v.Neighbors  Depthfirst(v2)
    endif
endproc
```

Figure 2b: Dft-Connected

```
/* check if there are cycles in the graph */
for each v ∈ Vertices  v.mark := unvisited   /* initialize each graph vertex as unvisted */
for each v ∈ Vertices  v.path := offthepath  /* each vertex also has a path variable */
Cycle := false
for each v ∈ Vertices  Depthfirst(v)

proc Depthfirst(v : Vertex)
    if v.mark = unvisited and not Cycle then
                        /* stop the traversal after a cycle has been found */
        v.mark := visited
        v.path := onthepath
        for each v2 ∈ v.Neighbors do
                /* before visiting any neighbor, check whether it is on the path */
            if v2.path = onthepath  Cycle := true
            else Depthfirst(v2)
        enddo
        v.path := offthepath
    endif
endproc
```

Figure 2c: Dft-Cyclecheck

the functional components to the participating objects and function abstraction in identifying and representing the variation of these components.

Similarly, the basic structure of the Depthfirst procedure consists of some manipulation of each vertex visited, some manipulation of each edge traversed and finally some cleanup as the traversal exits each vertex.

Figure 3 shows the main part of the generic Depthfirst contract definition. This contract involves two kinds of objects: the graph object and the vertices of the graph. The contract includes the UndirectedGraph contract between graph and vertices which requires that these participants adhere to the obligations specified in that contract before particpating in the Depthfirst contract.

In addition, the new obligations on the graph participant stipulate that it must support an interface which includes the methods:

- depthfirst: This starts the traversal. The contract also defines causal constraints for this method, i.e. any implementation of depthfirst must accomplish at least the actions specified in the contract.
- init: to do any general initialization before the traversal begins.
- newRegionWork: to do any required processing on a vertex before sending it the message depthfirst. This is the first method invoked after a complete connected region of the graph has been traversed.
- finish: to return an answer (if one is required).

Customizing the generic algorithm with respect to the graph participant typically involves supplying more specialized versions of newRegionWork, init, finish, or all three. The depthfirst method could also be specialized further in the unlikely event that the definition provided was deficient. The customization may also add new methods and instance variables.

Similarly, each Vertex participant must support a number of instance variables and methods. The depthfirst method for Vertex also contains detailed causal constraints. Customizing the algorithm with respect to Vertex involves specializing some or all of:

- stop: when true, prevents the vertex from being traversed. Also used to stop the traversal altogether.
- vertexWork: executed on each vertex traversed.
- edgeWork: executed on each edge traversed.
- postWork: the last thing done on each traversed vertex.

The Graph and Vertex components implement the common functionality of the algorithms of figure 2 and provide the hooks needed to specialize this functionality. These components can now be independently specialized in refinements of this contract.

```
contract DepthFirst
    participants
        graph : Graph;
        vertices : Set(Vertex);

    includes UndirectedGraph(graph, vertices);

    Graph supports [
        public depthfirst(): Answer  [
            forall v in vertices{ v→ set-not-marked() };
            init();
            forall v in vertices{
                newRegionWork(v);
                v→ depthfirst(); }
            return finish();
            ]

        newRegionWork(v : vertex) : void;
        init() : void;
        finish() : Answer;
        ]

    Vertex supports [
        data : Data;
        mark : Boolean;
        set-not-marked() :void  [mark = false;]

        depthfirst()  [
            if (not (mark or stop())){
                mark = true;
                vertexWork();
                forall v in neighbors{
                    edgeWork(v);
                    v→ depthfirst(); }
                postWork();
                }
            ]

        default stop() : Boolean  [return false;]
        vertexWork() : void;
        edgeWork(v : vertex) : void;
        postWork() : void;
        ]

end contract
```

Figure 3:

```
instantiation
graph→ depthfirst()

invariant
graph→ depthfirst() ↦ ⟨∀ v : v ∈ Vertices : v.mark or v→ stop() ⟩
```

Figure 4:

To complete the definition of the contract, an *instantiation* (fig. 4) and an *invariant* clause can be added. The instantiation clause specifies which messages to send to initiate the contract. In this example, the contract starts when the graph participant receives the depthfirst message. Of course, an UndirectedGraph contract must already exist between the graph and its vertices. The invariant specifies that sending depthfirst will lead to the traversal of every vertex for which stop evaluates to false.

4 Customizing depth first traversal

Section 3 defined the basic depth first traversal algorithm in contract form. In this section we will reuse this contract to define some of the many graph algorithms based on depth first traversal. Contract *refinement* is the basic mechanism used to achieve the reuse.

Refinement allows the definitions of one, or all, of the participant types to be changed in the following ways:

- New instance variables or methods added.
- Causal obligations can be provided for currently empty methods, e.g. newRegionwork, init or vertexWork.
- Existing causal obligations can be specialized, e.g. the definition of Graph::depthfirst could be specialized if required.
- Default method definitions can be overridden, e.g. Vertex::stop.
- Unconstrained or general types (e.g. Answer) can be replaced with more specific types[19].
- New participants can be introduced.

Contract refinement conveniently allows the simultaneous customization of participant obligations to be represented in a single software unit.

Three refinements of Depthfirst are defined here, Dft-Connected (fig.5), Dft-Number (fig. 6) and Dft-CycleCheck (fig. 7). Each of these specializes a number of the methods of Graph and Vertex and provides a definition for a new participant, the Workspace object. Only new or specialized information needs to be specified in a contract refinement. All other obligations are inherited from the base contract. Any method declared in the generic contract but not defined in the specialization results in a default empty implementation.

```
contract Dft-Connected refines Depthfirst
   pariticipants
      workspace : Workspace

   Graph supports [
      finish() : Graph  [return this; ]
      newRegionWork(v : Vertex) : void  [
         if (not v→ marked()) workspace→ inc-value(); ]
         ]

   Vertex supports [
      componentNum : Number;
      vertexWork() : void  [
         componentNum = workspace→ get-value(); ]
      marked() : Boolean  [return mark; ]
      ]

   Workspace supports [
      value : Number;
      inc-value() : void  [value = value + 1; ]
      get-value() : Number  [return value; ]
      ]
end contract
```

Figure 5:

The variables used by the different variations of the depth first algorithm defined in figure 2, i.e. DNumber, CNumber and Cycle, will be provided by the workspace participant. Since every contract participant has global scope within the contract, the workspace participant is visible throughout the traversal and will provide the required storage and interface to change and retrieve the values of these variables. The three different contract specializations will define slightly different workspace participants according to their different requirements.

Figure 5 shows new definitions of finish and newRegionWork for Graph. The new definition of finish specifies that calling depthfirst to run the connected component algorithm will return the graph object. Note that Answer is replaced with Graph. The newRegionWork method simply increments the value of an integer stored in the workspace. This value is later retrieved by vertexWork.

The definition of Vertex is extended with a new instance variable componentNum,

```
contract Dft-Number refines Depthfirst
    pariticipants
        workspace : Workspace

    Graph supports [
        finish() : Graph  [return this;]
        ]

    Vertex supports [
        dfsnum : number;
        vertexWork() : void  [
            dfsnum = workspace→ get-value();
            workspace→ inc-value(); ]
            ]

    Workspace supports [
        value : Number;
        inc-value() : void  [value = value + 1; ]
        get-value() : Number  [return value; ]
        ]
end contract
```

Figure 6:

```
contract Dft-CycleCheck refines Depthfirst
   pariticipants
      workspace : Workspace

   Graph supports [
      init() : void  [workspace→ reset(); ]
      finish() : Boolean  [return workspace→ get-cycles(); ]
      ]

   Vertex supports [
      on-the-path : Boolean;
      vertexWork() : void  [on-the-path = true; ]

      edgeWork(w : vertex) : void  [
         if (w → on-the-path()) workspace → set-cycles(); ]

      postWork() : void  [on-the-path = false; ]
      stop() : Boolean  [return workspace→ get-cycles();]
      ]

   Workspace supports [
      cycles : Boolean;
      set-cycles() : void  [cycles = true; ]
      get-cycles() : Boolean  [return cycles; ]
      reset() : void  [cycles = false;]
      ]
end contract
```

Figure 7:

a new method marked and a specialization of vertexWork. After the algorithm is executed, all the vertices of a connected region will have the same value in their componentNum instance variable. This provides a cheap way to test if two vertices of the graph are part of the same region and, in the case of undirected graphs, whether one vertex is reachable from another. The workspace keeps track of the numbering and provides the means to increment and retrieve the value.

Figure 6 shows the Dft-Number refinement. This numbers the vertices of the graph in the order they are traversed. It requires fewer new details than Dft-Connected. In this case, the change to the number takes place in vertexWork. The definition of Workspace is identical, as is the specialization of finish.

The Dft-CycleCheck refinement is more complex, requiring the specialization of vertexWork, edgeWork, postWork and stop. However, all these definitions are compact. The complex logic of the algorithm is coded in the depthfirst methods of the base contract which are reused. Note that this variant requires a Boolean return value to indicate whether or not the graph has a cycle. The specialization of finish is defined with a Boolean return type and the answer is retrieved from workspace.

```
class Network has parts
   type : String;
   :
end class Network

class Connection has parts
   data : ComputerInfo;
   :
end class Connection

class MyWorkspace
end class MyWorkspace

class ComputerInfo has parts
   :
end class ComputerInfo

conformance
   Dft-CycleCheck(Network,Connection,MyWorkspace);

conformance
   Dft-Connected(Network,Connection,MyWorkspace);
```

Figure 8:

5 Using Contracts

The above sections illustrate how algorithmic components can be specified and refined in contract form. This section discusses how these can be reused in the development of an object oriented program. The goal is to generate appropriate class and method definitions from the contract specifications. This generative approach to contract implementation requires more detailed contract specifications than those described in *Contract*$_0$.

The main focus of *Contract*$_0$ is the use of the contract formalism for high level specification of object behavior and intercommunication. In this context, detailed control logic is usually omitted. The burden of detail belongs to the implementor rather than the designer. Implementation is then mapped to the contract specification with *conformance declarations.* A conformance declaration states how a class implementation conforms to a contract participant specification. Verifying the correctness of conformance declarations is an important but difficult problem.

In the current context of automatically generating contract implementations, the verification problem is removed but it shifts the burden of detail to the specification. In the context of algorithm reuse this is very appropriate. The important aspects of an algorithm that we want to encode in a reusable component are precisely the detailed control logics, e.g. the depthfirst method definitions in the Depthfirst contract.

To reuse the structures and code specified in a contract, we select the application classes whose instances will participate in that contract. For example, an application to control a local area network, may have a class Network and a class Connection. Suppose the network is to be implemented as an undirected graph with the network connections as the vertices and the implementation requires cycle checking and component numbering, the Network object can participate as the Graph participant and the Connection objects can participate as the Vertex participants in the Dft-CycleCheck and Dft-Connected contracts. This can be specified using a new form of conformance declaration (fig. 8). In these conformance statements the application classes are listed in same order as the corresponding participant definitions in the contract.

Declaring that Connection conforms to Vertex in the Dft-Connected contract implies that Connection objects must have instance variables data, mark and componentNum, and must have methods stop, vertexWork, edgeWork, postWork, and marked. If these are not provided by the local definition of Connection, they can be derived from the contract definition and added. In order to do this automatically, the conformance information for the other contract participants is needed. This additional information is not required when conformance is used only for verification.

The conformance statements in figure 8 will also result in MyWorkspace objects supporting value, cycles, inc-value, get-value, set-cycles, get- cycles and reset. These are the combined Workspace requirements of both contracts.

6 Implementation issues

The development of the C++ prototype has turned up some interesting problems revolving around the simultaneous reuse of closely related contracts. The problem is analogous to the multiple inheritance problem of inheriting multiple definitions of the same method. In the example of figure 8, both of the contracts used, Dft-CycleCheck and Dft-Connected, inherit the implementation of Graph::depthfirst from the base Depthfirst contract. Even though neither of the contracts specialize depthfirst, subsequent functions called by depthfirst are specialized, e.g. vertexWork. At run time, the depthfirst method must dynamically chose the appropriate implementation of vertexWork from Dft-CycleCheck or Dft-Connected.

To distinguish which contract instantiation is required, we supply a contract lens[3] object(fig. 9) which supplies the information used by depthfirst to make the choice. The generated implementation for depthfirst is coded to use the contract lens, ensuring that the appropriate specialized functions are invoked. A sketch of

[3]These are very similar to the meta-level contract objects used by Rao[22] in his implementation of the Silica system. Dan Walkowski suggested the lens metaphor.

the generated implementation is shown below, the complete technical details of the contract lens implementation and the conflict resolution mechanism will be described in a future paper.

To facilitate maximum flexibility the implementation mechanism supports[4] the following features:

1. Type substitution[19]: e.g. Data of Vertex is replaced by ComputerInfo of Connection.
2. Method overriding: local definitions of methods in the application class will override similarly named methods defined for the contract participants. This allows the implementor to non- intrusively customize the component should the defaults be deficient in some way.
3. Combination: A class can conform to participants of many contracts, even if there are conflicting methods with the same name defined in the different contracts. The Connection class conforms to Vertex in Dft-CycleCheck and Vertex in Dft-Connected and these have two different definitions for the vertexWork method.

```
main(){
:
Network * net;
MyWorkspace * workspace;

:
DfsNum_Contract * dfsnumC =
        DfsNum_Contract::Instantiate(net, net→ get_adj(), workspace);

net→ depthfirst(dfsnumC);
:
CycleCheck_Contract * cycleC =
        CycleCheck_Contract::Instantiate(net, net→ get_adj(), workspace);

net→ depthfirst(cycleC);
:
}
```

Figure 9:

[4]The initial prototype does not fully support the first two features.

Combination allows each contract to be designed independently of other contracts and of their eventual use. The basic rule governing conflicts resulting from combination is: instance variables (or methods) with the same name are merged if their definition is identical, otherwise they are named apart using the contract name. Code using the instance variables (or methods) is modified accordingly. This is the default mechanism. Mechanisms to rename apart, or conversely, to rename equal, derived instance variables and methods are being investigated.

To illustrate the problem being addressed here, it is useful to consider how this software might be implemented using standard approaches. The usual approach to implementing a framework such as this would probably produce the class hierarchy shown in figure 10. The abstract class Vertex implements the depthfirst method and defines the protocol for vertexWork, edgeWork etc. Two subclasses, CycleVertex and ConnectedVertex provide the implementations for these functions and add the required instance variables. However, this supports the reuse of one of the variants, cycle checking or connected component numbering, but not both. With this hierarchy, a graph could have instances of CycleVertex for vertices *or* instances of ConnectedVertex, but not both. To further specialize the vertices for an application, the implementor must choose either CycleVertex or ConnectedVertex to specialize.

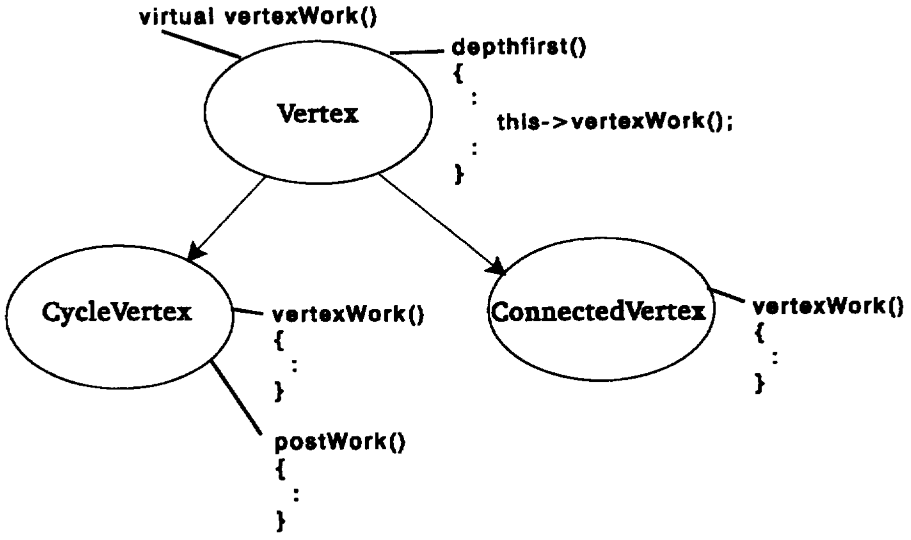

Figure 10:

Two problems arise if multiple inheritance is used to subclass from both CycleVertex and ConnectedVertex. First, there is a name conflict between CycleVertex::vertexWork and ConnectedVertex::vertexWork. Second, there is the related problem of invoking the appropriate implementation of vertexWork from Vertex::depthfirst. There are similar problems with implementations based on parameterized classes. Neither type parameterization nor inheritance fully support the features listed above.

Contracts allow individual classes to independently conform to many participant definitions, even if two, or more, of these definitions are refinements of a single definition. The combination mechanism resolves the naming problem and ensures the correct dispatching of messages.

The generated implementation supporting the combination mechanism has four main components:

1. The abstract contract class definition.
   An abstract base class Dfs_Contract is defined with virtual functions for each participant method which is specialized in more than one of the contract refinements e.g.

   ```
   init(Graph *, Dfs_Contract *)
   vertexWork(Vertex *, Dfs_Contract *)
   edgeWork(Vertex *, Dfs_Contract *, Vertex *)
   ```

   Each function is passed the participant responsible for providing the required implementation as well as the current contract lens instance.

2. The contract lens class definitions.
   A concrete class is defined for each contract refinement used. These classes specialize the corresponding abstract contract class. In the example, classes DfsNum_Contract and CycleCheck_Contract are defined. Each provides a one line implementation for the methods they specialize, e.g.

   ```
   DfsNum_Contract::vertexWork(Vertex * v, Dfs_Contract * contract){
       v→DfsNum_vertexWork(contract); }

   CycleCheck_Contract::vertexWork(Vertex * v, Dfs_Contract * contract){
       v→CycleCheck_vertexWork(contract); }
   ```

3. Extending the definitions of the conforming classes.
   Each application class is extended by the attributes and methods defined for the participants. This is accomplished by creating a new super class for each of the application classes. For example, a super class Vertex is defined for the application class Connection. The attributes and methods for each participant the application class conforms to are defined in the new super class. Method name conflicts are resolved by prefixing the method name with the contract name, e.g.

   ```
   Vertex:::DfsNum_vertexWork(Dfs_Contract * contract) and
   Vertex:::CycleCheck_vertexWork(Dfs_Contract * contract).
   ```

4. Generated method implementations.
   The generated implementations are different from the contract implementations in two ways. First, contract language elements such as the forall construct are replaced by conventional C++ code. Second, any method invocation involving a method with conflicting implementations is sent to the contract lens instead. The contract lens then dispatches the appropriate implementation variant. For example, the following fragment from the Vertex:::depthfirst method from the contract in figure 3:

```
if (not (mark or stop())){
    mark = true;
    vertexWork();
    :
```

is replaced by

```
if ( ! (( mark == 1 )
        contract→stop( this, contract ) == 1)){
    mark=1;
    contract→vertexWork( this, contract );
```

7 Conclusion

This paper described the results of applying the Contract language in the domain of algorithmic reuse. We have shown how classic algorithms, specifically depth first traversal of undirected graphs, can be formulated, specialized and reused. We have shown how specialized algorithms can be defined in terms of more general ones and how users can customize these algorithms for their particular application. Performing data and functional abstraction on previous instantiations of the algorithm produces a decomposition of the algorithm into discrete, named structural and functional components. We have formulated the algorithms in terms of objects, which package the algorithmic components into cohesive units, and organized these objects as participants of a contract.

The contract construct builds on the object oriented features of inheritance and class parameterization to provide the required flexibility. When refining an algorithm in contract form, any of the structural or functional components can be extended or overridden with new definitions. Following the object oriented paradigm of 'programming by difference', only that which changes needs to be specified. All the other components are inherited from the base contract. This avoids having an explicit variable part declared for the reusable abstraction and simplifies reuse by providing appropriate defaults. Type substitution allows any unconstrained type used in the base contract to be replaced in a refinement of the contract or by an application implementation. This allows the user to use newly defined application specific types when reusing a contract.

As with multiple inheritance, reuse of multiple contracts can cause name conflicts. However, since all of the conflicting methods are required at one time or another, we can neither choose one over the rest, nor rename them. To resolve the conflict, and to ensure the correct versions are executed, we augment the message with contextual information, i.e. an appropriate contract lens object. The code generated from the contract definitions uses the contract lens to execute the required version. We have built a first prototype which takes contract definitions as input together with conformance statements and application class definitions and generates the appropriate C++ code, utilizing contract lens objects. Future work will yield other tools which support the use of contracts for application development.

8 Acknowledgements

The author would like to thank A. Richard Helm, Harold Ossher, Brent Hailpern, Markku Sakkinen and Professor Karl Lieberherr for their detailed feedback on earlier drafts of this paper. Thanks also to Christos Stamelos for turning the ideas into code.

References

[1] Constantin Arapis. Specifying Object Life-Cycles. In Dennis Tsichritzis, editor, *Object Management*, pages 133–195. Centre Universitaire D'Informatique, Genève, 1990.

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

[3] Lucy Berlin. When objects collide: Experiences with reusing multiple class heirarchies. In *Object-Oriented Programming Systems, Languages and Applications Conference/European Conference on Object-Oriented Programming, in Special Issue of SIGPLAN Notices*, pages 181–211, Ottatwa, Canada, 1990. ACM.

[4] Ted Biggerstaff and Charles Richter. Reusability framework, assesment and directions. *IEEE Software*, pages 41–49, July 1987.

[5] Judy Bishop. The Effect of Data Abstraction on Loop Programming Techniques. *IEEE Transactions on Software Engineering*, 16(4):389–402, April 1990.

[6] Grady Booch. *Object-Oriented Design With Applications*. Benjamin/Cummings Publishing Company, Inc., 1991.

[7] Peter Deutsch. Reusability in the Smalltalk-80 Programming System. In *ITT Proc. Workshop on Reusability in Programming*, pages 72–82, Newport, RI, 1983.

[8] B. Hailpern and H. Ossher. Extending Objects to Support Multiple Interfaces and Access Control. *IEEE Transactions on Software Engineering*, 16(11), November 1990.

[9] A. Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 169–180, Ottowa, 1990. ACM Press. joint conference ECOOP/OOPSLA.

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

[11] W. L. Johnson and E. Soloway. PROUST: Knowledge-based program understanding. *IEEE Transactions on Software Engineering*, 11:267–275, 1985. reprinted in: C. Rich and R.C. Waters, eds., Readings in Artificial Intelligence and Software Engineering (Morgan Kaufmann, Los Altos, CA. 1986).

[12] T. Capers Jones. Reusability in Programming: A Survey of the State of the Art. *IEEE Transactions on Software Engineering*, 10(5):488–494, September 1984.

[13] Gail E. Kaiser and David Garlan. MELDing Data Flow and Object-Oriented Programming. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, volume 22, pages 254–267, Orlando, Florida, 1987. ACM SIGPLAN Notices.

[14] Shmuel Katz, Charles A. Richter, and Khe-Sing The. PARIS: A System for Reusing Partially Interpreted Schemas. In *International Conference on Software Engineering*, pages 377–385, 1987.

[15] Charles Krueger. Models of reuse in software engineering. Technical Report CMS-CS-89-188, Department of Computer Science, Carnegie Mellon University, December 1989.

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

[17] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing User Interfaces using InterViews. *IEEE Computer Magazine*, pages 8–22, February 1989.

[18] Bertrand Meyer. Tools for the new culture: Lessons from the design of the Eiffel libraries. *Communications of the ACM*, 33(9):68–88, September 1990.

[19] Jens Palsberg and Michael I. Schwartzbach. Type substitution for object-oriented programming. In *Object-Oriented Programming Systems, Languages and Applications Conference/European Conference on Object-Oriented Programming, in Special Issue of SIGPLAN Notices*, pages 151–159, Ottatwa, Canada, 1989. ACM.

[20] Barbara Pernici. Class Design and Meta-Design. In Dennis Tsichritzis, editor, *Object Management*, pages 133–195. Centre Universitaire D'Informatique, Genève, 1990.

[21] Ruben Prieto-Diaz and James M. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4):307–334, November 1986.

[22] Ramana Rao. Implementation Reflection in Silica. In *European Conference on Object-Oriented Programming*. Springer Verlag, 1991.

[23] T. Reenskaug and E. Nordhagen. The Description of Complex Object-Oriented Systems:Version 1. Technical report, Senter for Industriforskning, Oslo, Norway., 1989.

[24] C. Rich and H. E. Shrobe. Initial report on a Lisp programmer's apprentice. In D. R. Barstow, H. E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 443–463. McGraw-Hill, 1984.

[25] Charles Rich and Richard C. Waters. *The Programmers Apprentice*. Frontier Series. ACM Press, 1990.

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

[27] Rebecca J. Wirfs-Brock and Ralph E. Johnson. A survey of current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, September 1990.