Metaprogramming Compilation of Invariant Maintenance Wrappers from OCL Constraints

Spencer Rugaber
Georgia Institute of Technology

Kurt Stirewalt
Michigan State University

Abstract

Invariants are system properties that describe relationships among components. When the state of a system changes so as to invalidate an invariant, the components in the system must react to reestablish it. Mode components are a new mechanism for providing invariant maintenance. Mode components transparently detect state changes and notify dependent components using implicit invocation. Mode components make use of mixin layers implemented by the metaprogramming features of C++. This report describes the design and implementation of mode components.

Keywords: implicit invocation, code generation, components, layered architecture, metaprogramming, OCL, invariant maintenance

1 Introduction

1.1 Motivation

Invariants are system properties that describe relationships among components. When an external event causes the state of a component to change, it may be necessary to alert other components so that system invariants can be reestablished. These activities are called *invariant maintenance*. Assembling large systems with many components is difficult because of the many interactions that the components can have. Interactions that cause state changes can initiate a cascade of further interactions as the system tries to reestablish its invariants. It is the goal of the work described in this report to support the expression of invariants declaratively at a high level of abstraction and then to generate wrapper code that detects state changes and initiates invariant reestablishment. Invariant specifications are expressed in the Object Constraint Language (OCL) [4][9], and wrapper code is generated using the metaprogramming (template) facilities of C++. This paper describes the approach taken and evaluates its strengths and weaknesses.

1.2 Background

For our purposes, we assume that a component corresponds to a C++ class. Components have state. Component state that is visible to other components is called *status.* A collection of components that comprise a system is called an *assembly.* An assembly interacts with its environment by reacting to events. Events trigger status changes that ultimately are communicated back to the environment via *percepts* (externally visible status).

Assemblies of components and their invariants are expressed using specially interpreted UML class diagrams annotated with OCL constraints. In the diagrams, components are denoted by UML classes, and invariants are denoted by associations annotated with OCL constraints. OCL can be thought of as first-order predicate logic combined with collection classes and syntax to express navigation within the class diagram. Both the Rational Rose and ArgoUML CASE tools sup-

port OCL and provide a way to export annotated diagrams, expressed in XML, for processing by other tools.

1.3 Non-functional Requirements

A solution to the invariant maintenance problem must satisfy several non-functional requirements. First, it must refrain, to the extent possible, from intruding into the components themselves. This requirement, called *transparency*, has several advantages. It separates reasoning about overall system properties from consideration of the individual components. Also, it reduces the need to modify the code of the components, thereby lessening the risk of introducing bugs.

The second requirement is *flexibility.* There are a variety of architectural mechanisms that support invariant maintenance. A flexible solution is one in which the architectural approach can be selected by the designer based on other desirable system properties. Moreover, flexibility supports reuse, enabling components to be packaged in various ways.

The third requirement is *low overhead*. In particular, it should be the goal of any invariant maintenance mechanism to cause no additional run-time costs over an *ad hoc* implementation. As a general rule, the more encapsulated and self-contained components are, the more complex is the collaboration mechanism required to support them. With complexity comes run-time overhead. A low-overhead solution supports collaboration without any additional run-time cost.

The fourth requirement is *intentionality.* That is, in order to reason about system behavior, it should be possible to relate the behavioral specification of an invariant to its implementation in a direct way. In particular, each invariant should be traceable to the code mechanism responsible for maintaining it. Intentionality also supports maintainability—changes to system requirements often mean changes to the system's invariants. Invariants implemented intentionally are easier to alter.

The final property is *abstraction*. Confidence in system behavior is reduced when code must be examined to determine the effects of intricate combinations of events. Such efforts are labor intensive and error prone. Abstraction supports maintainability. Changes to abstractly expressed system properties can be made without requiring in-depth knowledge of implementation internals.

1.4 Approach

The solution to the invariant maintenance problem described in this document comprises automatically generated component wrappers called *mode components*. A mode component transparently intercepts status changes and notifies dependent components. Notification is implicit insofar as the wrapped component is concerned. That is, the independent component is not aware that its status is being monitored or that other components are dependent on it. Both wrappers and implicit notification support transparency. Moreover, because the OCL invariants are compiled, abstraction and intentionality are supported. That is, there is a direct mapping from the invariants to the generated mode components.

Mode components are implemented as layered (nested) C++ class template instantiations. A class template is a parameterized class definition, where parameters are usually other classes. In particular, a parameter can be used as the base class for a template class thereby enabling the mode components to be stacked into layers. When combined with C++'s compile-time inlining mechanism, much run-time overhead can be removed. Moreover, layering in conjunction with implicit notification supports flexible reuse.

2 Example

A simple assembly of three components is used as a running example. The assembly acts as a text browser. That is, an end user can examine the contents of a document by scroll-

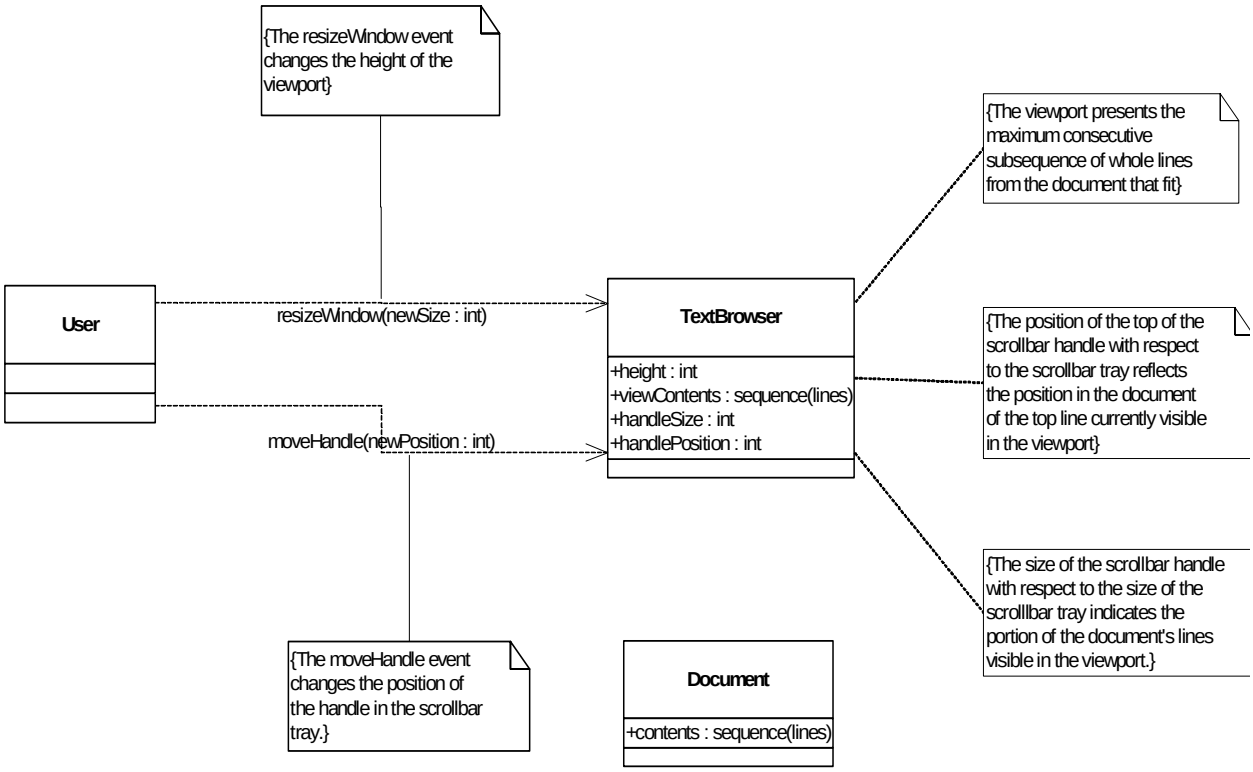

Figure 1. External Context for Text Browser

ing through it. The assembly comprises three components providing document access (FileManager), content display (Viewport), and positioning control (Scrollbar).

The browser is denoted by a single class (TextBrowser) in the UML class diagram shown in Figure 1. Other classes in the figure denote the user and the document whose contents are being displayed. The text browser reacts to two user events—one to set a new position to view (moveHandle) and the other to resize the viewport (resizeWindow). Likewise, there are four percepts that convey semantic content: the actual lines displayed (viewContents), the height of the Viewport (height), the position of the Scrollbar handle (handlePosition) and its size (handleSize).

Figure 1 also illustrate five properties that the assembly should provide in order to satisfy user expectation[1]. In the figure, they are denoted by natural language statements contained within annotation icons.

1. The resizeWindow event alters the size of the Viewport.
2. The moveHandle event alters the portion of the file displayed in the Viewport.
3. The Viewport displays the maximal consecutive subsequence of complete lines from the document that fit within it.
4. The position of the top of the Scrollbar handle relative to the Scrollbar tray reflects the position in the document of the line currently visible at the top of the Viewport. That is, moving the Scrollbar handle allows different portions of the document to be displayed.
5. The size of the Scrollbar handle with respect to the size of the Scrollbar tray is equal to the number of lines visible in the Viewport compared to the total size of the document.

The designer of a text browser assembly chooses components from a library of candi-

1. Some details concerning the handling of empty files have been elided.

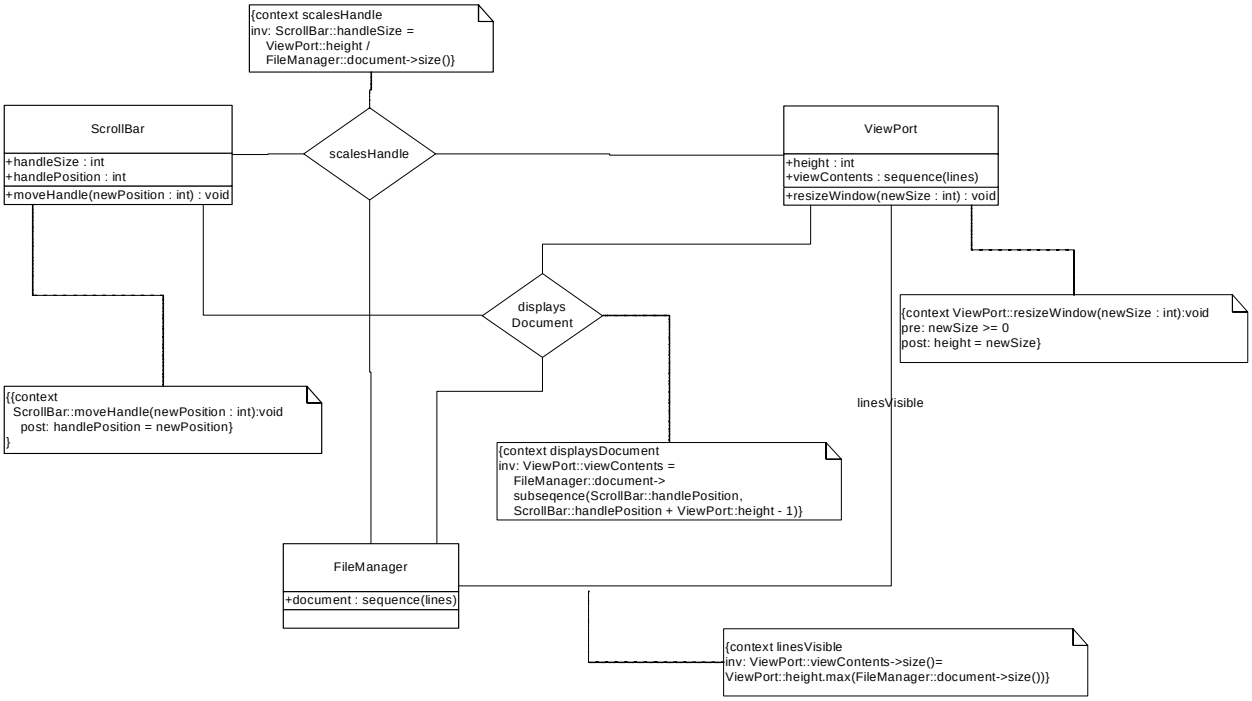

Figure 2. Text Browser Refinement

dates[2]. It is assumed that the components are correctly implemented and well documented. As the components are chosen, it is the obligation of the designer to construct the assembly from the available components in such a way that assembly properties are guaranteed. This task includes partitioning assembly responsibilities among components and translating informal invariant descriptions into OCL. Such a realization is illustrated in Figure 2.

In the figure, the three components are connected by three associations. Each association is annotated with an OCL constraint expressing the relationships among the adjacent components. Additionally, the two user events (described in Properties One and Two above) to which the assembly responds (resizeWindow and moveHandle) have been assigned to the components appropriate to initially receiving them (Viewport and Scrollbar, respectively).

In UML class diagrams, binary associations are denoted by lines connected the related classes. For example, linesVisible is a binary association between Viewport and FileManager that expresses the portion of Property Three that guarantees that the Viewport is as full as possible. The remainder of Property Three and the other two properties (Four and Five) are ternary associations, denoted by diamonds in the figure. For example, the remainder of Property Three relates the contents of the Viewport to the contents of the document. It is described in the OCL annotation of the displaysDocument ternary association. All three components are involved in describing and maintaining this invariant.

Figure 3 indicates how displaysDocument is translated into OCL. In OCL, each constraint is associated with a class, called it *context*[3]. In the case of the invariant being modeled, the displaysDocument association is the context. Context and inv are OCL keywords, the latter

2. Of course, many GUI libraries, such as Java Swing, contain components comprising all three aspects of the text browser example.

3. We have temporarily relaxed this rule to allow constraints to be attached to associations.

```
{context displaysDocument inv:
ViewPort::viewContents =
  FileManager::document->
    subseqence(
      ScrollBar::handlePosition,
      ScrollBar::handlePosition +
      ViewPort::height - 1)}
```

Figure 3. OCL Formalization of Property Four

separating the context information from the body of the constraint proper. This particular constraint takes the form of an equation that relates the values of status elements of components. OCL adopts the C++ class scope operator (::) to identify the component to which a status element belongs. For example, `ViewPort::viewContents` denotes the value of the viewContents attribute of the Viewport component. Its value must be kept equal to a subsequence of the lines provided by the FileManager. That subsequence, in turn, is determined by the position of the Scrollbar handle and the size of the Viewport.

3 Compilation Architecture

This section describes how OCL constraints are realized as generated C++ wrapper code. To do so, it makes use of two key devices—status variables and mode components.

3.1 Status Variables

A fundamental concept in our approach to solving the invariant maintenance problem is that of a status variable. The attributes of a component to which other components are sensitive are called its *status*. A *status variable* is a lightweight wrapper on an instance variable that detects changes to its value and invokes a method to handle announcements to dependent components.

To illustrate how status variables work, consider the trivial example of two components, A and B, with integer instance variables a and b, respectively, such that variable a must hold exactly twice the value of variable b, regardless of how b changes. That is, there is an invariant between A and B such that a = 2 * b. Expressed in OCL, this invariant is {context A inv: a = 2 * B::b}. It is assumed that the value of b can change in arbitrary ways. A C++ schematic for component B is shown in Figure 4, where tweak is an arbitrary, externally visible, method capable of altering the value of b.

```
class B {
  int b;
  public:
    ...
    void tweak(const int& x) {
      b = x;
    }
    ...
};
```

Figure 4. Schematic for an independent component

When tweak is called, b is altered, thereby requiring an update to A.

Status variables take advantage of several C++ features, the most important of which is the ability to overload the assignment operator. That is, when an assignment is made to a C++ variable, a programmer-provided method can be invoked to perform additional activities. The power of status variables is their use of assignment overload to transparently detect changes of status.

All status variables inherit from the class template StatusVariable defined in Figure 5. The template parameter T for class StatusVariable is the type of the attribute to be wrapped. In the case of attribute b, the type is int. Status variables have one attribute of their own, named data, protected from external access. This attribute holds the actual value being wrapped. Changes to b are trapped by the assignment overload method (operator=) on lines 7 and 8. This method is virtual (polymorphic) and will be extended in a derived class by a method that notifies component A that b has been altered. The only responsibility that

```
( 1) template <typename T>
( 2) class StatusVariable {
( 3)   public:
( 4)     StatusVariable() {}
( 5)     StatusVariable(const T& t)
( 6)       : data(t) {}
( 7)     virtual T& operator=
( 8)       (const T& t) {data = t;}
( 9)     virtual operator T() {
(10)       return data;
(11)     }
(12)   protected:
(13)     T data;
(14) }
```

Figure 5. StatusVariable class

the assignment overload operator has in the StatusVariable class is to assign the new value to data.

Clients of status variables, such as component A, do not know that attribute b has been wrapped. Hence, when they request the value of b, they must be provided an int, not a StatusVariable. C++ provides a supporting mechanism, called a user-defined conversion, as illustrated on lines 9-11 by operator T(). In our example, T is int, and the int() method is invoked whenever the value of b is requested, either explicitly within the code of B, or implicitly, via compiler-generated conversions. Hence, the int value of data will be returned whenever the value of the status variable wrapping b is requested.

The StatusVariable class also provides constructors (lines 4 and 5-6) in case class B provides an externally visible way to initialize b.

3.2 Using Status Variables

Given a constraint, its dependent and independent variables can be determined. Changes to the independent variables must be detected and the associated dependent variables adjusted to reflect the change. That is, each independent variable in each constraint must be wrapped as a StatusVariable so that changes in its value can be detected.

The wrapping is provided by a generated C++ template class. The name of the class is formed from the name of the status variable and the component containing it, hence ensuring uniqueneness. For variable b of component B, the generated template class has the name SV_B_b. SV_B_b has the form illustrated in Figure 6[4].

```
( 1) template <typename T>
( 2) class SV_B_b :
( 3)   public StatusVariable<T> {
( 4)   public:
( 5)     SV_B_b() {}
( 6)     SV_B_b(const T& x) :
( 7)       StatusVariable<T>(x) {}
( 8)     void setUpdater1(
( 9)       Updaters* sc1P) {
(10)       updater1P = sc1P;
(11)     }
(12)     T& operator=(const T& d) {
(13)       StatusVariable<T>::
(14)         operator=(d);
(15)       if (updater1P)
(16)         updater1P->update1();
(17)     }
(18)   protected:
(19)     Updaters* updater1P;
(20) };
```

Figure 6. Status change announcement mechanism

Note that SV_B_b derives from StatusVariable (line 3) and overrides the assignment operator (lines 12-17). The override first invokes the assignment operator in StatusVariable, thereby storing the assigned value. It then invokes an update method (update1), if it exists. The digit 1 in the method name indicates that this is the updater for the first constraint. The update method, which also must be generated, lives in component A (as wrapped), and contains the code to retrieve the new value of b and update a accordingly. When SV_B_b is generated, it must know the name of the update method to invoke (update1) and which component it lives in (A). It obtains this infor-

4. #include statements have been elided.

mation when the setUpdater1 method is called by the component aggregating the status variable (B).

3.3 Component Modification

The actual wrapping of component status takes the form of a single alteration to the component's implementation. The changes replace the type of the variable (int in the case of b) with the type of the generated class (SV_B_b<int>). After this is done, alterations to b within B will be processed by the wrapper code in the SV_B_b and StatusVariable classes, which alert dependent components such as A to the change. A can then request the new value of b in order to set the value of a appropriately, thereby reestablishing the invariant. A UML class diagram showing the relationship of the classes responsible for wrapping status is shown in Figure 7.

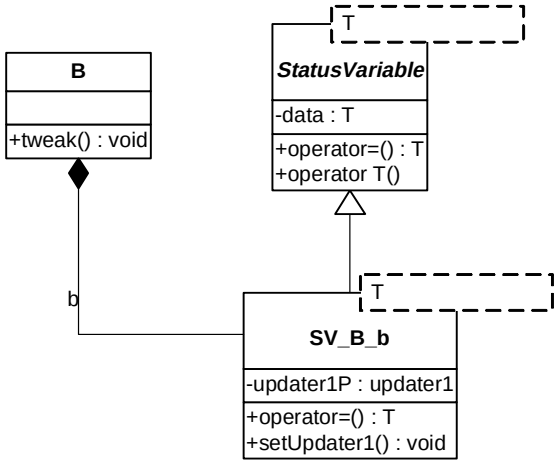

Figure 7. Status Variable Wrapping

3.4 Mode Components

It remains to describe how dependent components (such as A) are bound to independent components (such as B). In the example of A and B, A is responsible for updating the value of a when b is changed. It does this in a generated method like update1 in Figure 8. That is, a new method for A (update1) is generated, which is called when b changes. Its responsibility is to request the new value of b and, using it, to recompute the value of a. This raises several questions: Where does the code for update1 live; how does b know to call update1; and how does A know how to obtain the value of b?

```
void update1() {
  a = 2 * B.getValue_b();
}
```

Figure 8. Invariant reestablishment method

The update1 method logically lives in component A. However, as we wish to leave existing components untouched to the extent possible, we will generate a new wrapper that extends A with the update method. The wrapper for A is described in Section 3.5.

The other two questions can be resolved by further wrapping B in such a way that the required information is available. Once B is wrapped, it becomes a mode component.

A *mode component* is a wrapped component containing one or more status variables. The mode component wrapper for B is named MC_B, and it is generated based on the status variables and invariants specified for the assembly. MC_B is shown in Figure 9.

```
( 1) template <typename T>
( 2) class MCB : public T {
( 3)   public:
( 4)     MCB() {};
( 5)     MCB(const int& x) : T(x) {}
( 6)     int getValue_b(void) {
( 7)       return(b);
( 8)     }
( 9)     void bind_b_1(
(10)       Updaters* scP) {
(11)       b.setUpdater1(scP);
(12)     }
(13) };
```

Figure 9. Mode component wrapper for component B

MC_B is a class template. Moreover, it is a *mixin* class template. This means that its template parameter is a class, and that MC_B derives from that class. That is, MC_B is a sub-

class of the class bound to the template parameter T. Mixins have been used as a way to provide behavior to a class in addition to that derived from its normal base class [3]. In the case of MC_B, its parameter is B. If A then refers to MC_B instead of B, it will obtain the extended behavior.

MC_B adds two methods to those available in B. getValue_b provides access to the status variable b's value. It can be called by A when A is alerted to changes in b. bind_b_1 is the mechanism whereby A can inform B of any invariant reestablishment methods that must be called when B's status changes. Specifically, bind_b_1's argument is a pointer to the update method in A (update1) responsible for maintaining the invariant. MC_B's responsibility is to communicate this pointer to the wrapper for b (as seen on line 11).

3.5 Binding

The binding between components related by invariants is complex. Dependent components like A must be able to request status variable values, such as b. To do this, A must have access to B, the component that contains b. A straightforward way to do this is to have A contain a pointer to B. But pointers are costly, each access requiring the dereferencing of the pointer. The C++ template mechanism provides a better way—have A derive from B as a mixin. Then A can have direct access to b, just like it can to its own instance variables.

To summarize: A has four responsibilities that arise due to its interaction with B. 1) It must derive from B in order to access it efficiently. 2) It must let B know how to alert it when changes occur. 3) Once alerted, it must access the value of b. And 4) it must reestablish the invariant by recomputing the value of a.

To discharge these responsibilities while maintaining transparency, another wrapper is used. The generated wrapper code is illustrated in Figure 10.

```
( 1) template <typename T>
( 2) class Wrapper_A : public A,
( 3)   public Updaters, private T {
( 4)   public :
( 5)     Wrapper_A() {
( 6)       myB.bind_b_1(this);
( 7)     }
( 8)     void update1() {
( 9)       a = 2 * myB.getValue_b();
(10)     }
(11)   protected :
(12)     T myB;
(13) };
```

Figure 10. Wrapping dependent components

Wrapper_A is a mixin class template. Its template parameter is the component upon which it is dependent, B (as wrapped by MC_B). Wrapper_A mixes B in via private inheritance, thereby hiding B from subsequent clients of A. This inheritance discharges responsibility 1. In addition, Wrapper_A inherits publicly from two other classes, A and Updaters. Updaters is an interface class containing declarations for the types of updater methods.

The key feature of Wrapper_A is the update1 method on lines 8-10. This is the method called by the status variable b when it detects a change to its value. Notice that update1 accesses the value of b by using the getValue_b member function of component B. This method discharges responsibility 3. Line 9 also illustrates how the invariant is reestablished to discharge responsibility 4.

Responsibility 2 is handled by the wrapper's constructor shown on lines 5-7. When component A is instantiated, the method bind_b_1 is called in component B, passing the address of component A itself as an argument. The address is passed in turn to the setUpdater1 method of SV_B_b, where it is stored for use when b changes value.

3.6 Fine Print

In order to clearly and directly explain status variables and mode components, several details of the invariant maintenance process have been glossed over in the description above. Foremost among them is the seeming separation of A's invariant reestablishment wrapper (Wrapper_A) from B's announcement wrapper (MC_B). In reality, A itself may contain independent status variables participating in other invariants. For example, component Z might depend on variable a of component A. This would imply the need to generate an MC_A wrapper similar to MC_B. There is no reason why the code contained in Wrapper_A cannot be instead included in MC_A. That is, the extra facilities need to detect changes of status and inform superior components can be included in the same wrapper that receives notifications from inferior components in order to reestablish invariants.

Another detail not delved into is status variable initialization. If, for example, component B provided a way to initialize the value of b during construction, then the generated code has to include a memberwise initializer for it. There is no conceptual difficulty in doing so.

In addition to the code displayed above, the actual generation process includes generating several include files providing access to required names. These are also straightforward to provide.

Finally is the issue of extensibility. That is, how are multiple status variables and constraints handled, and what happens when the set of invariants being implemented presents a circularity in the dependency chain? These issues are described in the next subsection.

3.7 Extensibility

The approach described in Section 3.6 illustrated how an invariant dependent on a single status variable could be maintained. Real systems are more complex. This section describes how the approach can be generalized in several ways.

Multiple status variables. For each status variable x of type t aggregated by a component K, there is a corresponding generated class SV_K_x. Each x must be defined within K as a normal instance variable, but with type SV_K_x<t>. There is no limit to how many such variables K can have. Note that dependencies among status variables within a component are handled by traditional means within the component. That is, it is the responsibility of each component to maintain its own intracomponent invariants.

Multiple constraints. A given status variable, x, belonging to component K, may be involved in multiple constraints (C_i). Hence, multiple updates may have to be performed when the value of x changes. For each such constraint, an update method ($updateC_i$) and a bind method ($bind\_x\_C_i$) must be generated. Moreover, the code in the SV_K_x class must invoke each of the updater methods ($updateC_i$). Finally, the addresses of the updater methods must be remembered in the SV_K_x class with a function pointer (updaterCiP).

Circularities. In Section 3's example, component A is notified of changes to component B and then requests new values from it. The mode component mechanism for providing this ability takes advantage of C++ ability to nest templates. That is, component A (as wrapped) has as a template parameter component B (as wrapped). This mechanism is inherently asymmetric. That is, it cannot be used to have component A notify component B because then we would have a circularity in the template instantiation ordering.

Two things should be noted about circular dependencies such as this. First, there is no reason why components A and B cannot use traditional intercomponent messaging when A needs to notify B of a change. That is, B can provide an update method that A can call directly. The second observation is that a circularity is often a symptom of a design problem. One manifestation of the problem is an endless

loop—B notifying A which notifies B which ... Hence, any circularity in the dependency graph is likely a sign of a problem and should be avoided.

Multiple components per layer. Sometimes circular dependencies are inherent. For example, in the TextBrowser example, the Viewport must notify the Scrollbar when it is resized so that the Scrollbar can change the size of its handle. Likewise, the Scrollbar must notify the Viewport when the handle has been moved so that different lines can be displayed. These are circular dependencies that do not lead to an endless loop. As an alternative to the asymmetric mechanism of mode component, both components can be configured as nested classes contained within a single mode component class. C++ supports nested classes, including nested class templates. By placing both ViewPort and Scrollbar within the same class, say UIClass, the symmetry is maintained. And UIClass can become a component within other classes in the stack of nested class templates.

4 TextBrowser Example Continued

4.1 Layering

In order to generate wrapper code for the components described in Section 2, the components must be organized into layers. The primary determinant of the layering, is the dependencies inherent in the OCL constraints. Hence, via Property 3, the ViewPort is dependent on both the ScrollBar and the FileManager. For example, changes to the ScrollBar handle should be reflected in changes to what is displayed in the ViewPort. Consequently, the ViewPort should be assigned to a layer above the other components. Because the FileManager is not dependent on either of the other components, it resides in the bottom layer, with the ScrollBar in between.

Note that this layering is not the only one possible. Due to the circularity mentioned in Section 3.7, the ScrollBar is also dependent on the ViewPort, so it could appear in a superior layer. A configuration in which the ScrollBar and the ViewPort are assigned as multiple components in the same layer is also possible.

Once an assignment to layers is made, the UML associations can be replaced by dependency relations (dashed, directed lines) and the constraints assigned to the component that is responsible for reestablishing it when status changes. The resulting diagram is shown in Figure 11 on the next page.

4.2 Description of Generated Code

Using metaprogramming techniques (templates and mixins), we can create reusable mode components for each box in Figure 11. The mode components can be combined in a variety of ways, depending on implementation considerations. For example, imagine several extensions to the TextBrowser example of Section 2. One extension is to use a remote file stream as a source of lines to be displayed. Another extension improves performance by avoiding requests to FileManager for lines that are already being displayed.

Figure 12 depicts an instantiation of these

```
typedef fm_updates<
  fmStream<
    fm_status_filesize_constraints<
      fm_data_transformer_subsequence<
        fm_status_variable_impls > > > >::
          FM    FileManager;
```

Figure 12. Instantiation of the FileManager mode component

extension to FileManager. The modified version is synthesized from smaller fragments, each of which is a mixin layer. The layer fmStream implements the basic file-management functionality by reading and caching lines of text from a remote server over a network. Because this layer is reusable over many different assemblies, we store it in our library. Every other layer in Figure 12 is specific to this assembly and thus must be generated. Working

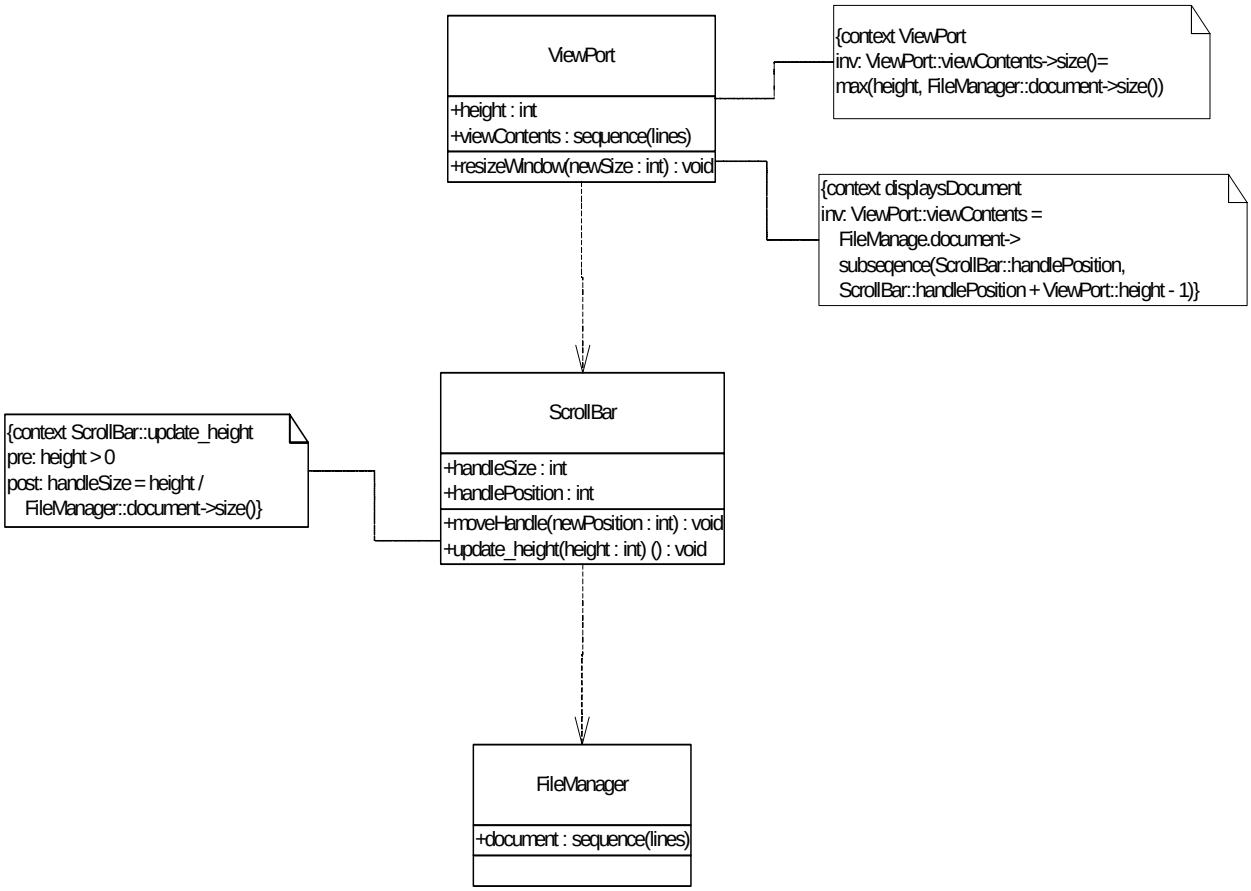

Figure 11. Layering of TextBrowser Components

inside out, the layer fm_status_variable_impls defines the type of the status variable document to be a sequence of strings that notifies listeners when elements are inserted into or deleted from the sequence.

The layer fm_data_transformer_subsequence then refines this type so that notifications are only issued if the events pertain to or effect sequence elements that lie within a given range of indices. We refer to this class of refinement as a *data transformer*, specifically a *subsequence data transformer*, and our library contains code that supports this refinement. In addition to refining the type of document to limit notifications, this layer also binds these notifications to invoke a specific update service in the ViewPort component. Because the ViewPort component has yet to be assembled, the binding is based on an interface class, which the ViewPort component needs to implement.

fm_status_filesize_constraints defines the type of the status variable filesize, which is referenced only indirectly in Figure 11[5]. This variable derives its value from insertion/deletion notifications from document, and it announces changes in the size of the file to all registered listeners. Because both the ViewPort and the ScrollBar depend on the value of filesize, this layer binds such notifications to invoke specific update services in higher-level components. Similar to the binding of document notifications, this binding is based on interace classes, which ViewPort and ScrollBar must ultimately implement.

5. All occurrences of the method invocation FileManager::document->size() are replaced by the status variable filesize.

Finally, the layer fm_updates refines the FM type, which is defined in fmStream to include additional configuration operations that enable higher level components (i.e., instances of ViewPort and ScrollBar) to register for notification of changes to document and filesize. In addition, the refinement defines services that enable higher-level components to adjust the upper and lower bounds of the subsequence. Because the subsequence is effectively encapsulated by FileManager, this explicit introduction of setter operations is necessary.

```
typedef sb_updates<
  sbSlider<
    sb_status_constraint_position_updates_vp<
      sb_status_variable_impls<
        sb<FileManager> > > > >::SB ScrollBar;}
```

Figure 13. Instantiation of the ScrollBar mode component

Figure 13 depicts the instantiation of the component named ScrollBar, which, like FileManager, is synthesized from smaller mixin layers. The layer sbSlider implements the basic scrollbar functionality and (like fmStream} is in our library. The layer sb is also in our library; it merely declares the basic types of status variables position and handleRatio. Every other layer must be generated.

Working inside out, the layer sb_status_variable_impls refines the types of status variable position and handleRatio to notify listeners of changes. The layer sb_status_constraint_position_updates_vp then further refines these types by binding the notifications to invoke a specific update service in the ViewPort component. Again, because the ViewPort component has yet to be assembled, the binding is based on an interface class, which ViewPort must implement. Finally, the layer sb_updates refines the SB type, which is defined in sbSlider, to include the update services used by the file manager. In addition, this layer provides configuration operations that enable higher level components (i.e., instances of ViewPort) to register for notification of changes to position.

```
typedef vtexttt{p_updates<
  vtexttt{pInvariant1<
    vtexttt{p_status_capacity_copydown<
      vtexttt{p_status_upper_bound_copydown<
        vtexttt{p_status_lower_bound_copydown<
          vtexttt{p_status_variable_impls<
            vp<ScrollBar> > > > > > >::VP ViewPort;
```

Figure 14. Instantiation of the ViewPort mode component

Figure 14 depicts the instantiation of the ViewPort component that is synthesized from smaller mixin layers. This synthesis is similar to the others; indeed the layers vp, vp_status_variable_impls, and vp_updates serve exactly the same function as their counterparts in the ScrollBar synthesis. Layers whose names end in the suffix copydown refine each of the status variables (i.e., capacity, lower_bound, and upper_bound) so that they invoke operations on subordinate components (specifically ScrollBar) whenever these variables witness a change. Having synthesized these mode components as depicted in Figure 12, Figure 13 and Figure 14, one can then instantiate an assembly by allocating an object of class ViewPort.

5 Evaluation

5.1 Transparency

What alterations to the source code of existing components are required in order to make them into mode components? Really only one change is necessary on the part of a programmer—the types of status variables must be adjusted. That is, member variables of components upon which other components are dependent must be defined as such. Two scenarios can be imagined. In the first, the developer of a component library is seriously concerned with reuse. Components are designed, and potentially interesting status is declared as such in the component code. The

second scenario is the adaptation of an existing component into a mode component. In this case, the adaptor must not only decide what facilities of the component are required of other components, but must also locate the definitions of these variables in the code, so that there types may be altered. In both sceneries, the coding effort required of the developer compises adding #include statements and changing the types of variable declarations. Any scheme for intercomponent invariant maintenance must provide access to the constituent state. Hence, we judge the mode component approach to be adequately transparent.

5.2 Flexibility

Our approach to component assembly enables designers to compose systems from existing components by organizing them into layers. Layering is inherently asymmetric. That is, component A may be layered above component B, or B may be above A, but not both at the same time. This restriction, in principle, limits flexibility. However, codependent components can be placed in the same layer using the technique described in Section 3.7. Situation in which this choice is not satisfactory are often indicative of design problems. Hence, we feel that our provides a flexible approach to component assembly.

5.3 Overhead

Flexibility normally leads to overhead. Typically, flexibility is achieved by using indirection through pointers. Making use of pointers implies dereferencing, which in turn means an extra operation on every access. Our approach reduces overhead by making use of two features of C++: template classes and inlining.

Components are normally constructed independently and encapsulated in their own classes. This reduces coupling and enhances maintainability. But, because components need to interact, they often hold pointers to each other. An alternative approach is to have one component be a subclass of another. Then the subordinate component can directly access the features of the superordinate component without the overhead of a pointer. But such an approach is intrusive and unnatural.

From the point of view of design abstraction, a derived type defines instances that are also instances of the base type. Clearly, components do not share such a relationship. Mixin inheritance is an alternative to subtyping. A mixin adds a feature to a class without requiring the mixin be a subtype.

Mixins are implemented in C++ using template classes in which the template parameter is the base class from which the template class derives. In the case of mode components, the subordinate component is the template argument. Hence, superior components have access to inferior features without the cost of pointer indirection.

The other C++ feature that can reduce overhead is inlining. Normally, the compilation of a method call introduces significant overhead at the calling site. The C++ compiler can detect situations where a copy of the code for the called method can be inserted directly at the call site without the associated overhead. This technique is particularly applicable when the method code is short, such as obtains with instance-variable access routines (*getters* and *setters*). In this way, components can retain their encapsulation without engendering normal intercomponent communication overhead. Templates and inlining enable our approach to provide low overhead invariant maintenance.

5.4 Intentionality

The goal of the work described in this report on component assembly is to increase assurance. It accomplishes this by providing an invariant maintenance mechanism. Invariants are directly manifest in the code. In particular, each independent variable in each constraint results in the generation of a status variable wrapper to provide change notifica-

tion and an update method to reestablish the invariant when one of its constituents changes. Because this code is generated, it is possible for the designer to have confidence that the specification is being met. Hence, the approach is intentional.

5.5 Limitations

As mentioned throughout this report, the described approach does have some limitations. They are summarized here.

Loss of symmetry with layers. Components nested as template mixins are inherently asymmetric. This loss of flexibility is compensated for by the reduced overhead required.

Constructiveness. Not every invariant can be expressed as a mode component constraint. Constraints in which a single variable appears on the left hand side are called *constructive* or *applicative*. This is a theoretical limitation of the approach that has not proven a problem in practice.

Circularities. More serious is the issue of cycles, as described in Section 3.7. As mentioned there, codependencies can be grouped into the same layer, providing a symmetric solution.

6 References

[1] D. Batory and B. Geraci. “Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions Software Engineering*, 23(2):67–82, February 1997.

[2] D. Batory and S. O’Malley. “The Design and Implementation of Hierarchical Software Systems with Reusable Components.” *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[3] Gilad Bracha and William Cook. “Mixin-based Inheritance.” *Proceedings ECOOP/OOPSLA '90,* October 21-25, 1990, 303-311.

[4] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Abstraction and Reuse of Object-Oriented Software.* Addison-Wesley, 1995.

[5] Object Management Group. “Unified Modeling Language, Version 1.4.” OMG Document Number 01-09-67, Chapter 6, http://www.omg.org/cgi-bin/apps/doc?formal/01-09-67.pdf.

[6] Y. Smaragdakis and D. Batory. “Implementing Layered Designs with Mixin Layers.” *Proceedings of the 12th European Conference on Object-oriented Programming,* 1998.

[7] R. N. Taylor et al. “Chiron-1: A Software Architecture for User Interface Development, Maintenance, and Run-Time Support.” *ACM Transactions on Computer-Human Interaction*, 2(2):105–144, June 1995.

[8] M. Young, R. N. Taylor, and D. B. Troup. “Software Environment Architectures and User-Interface Facilities.” *IEEE Transactions on Software Engineering*, 14(6):697–708, June 1988.

[9] Jos Warmer and Anneke Kleppe. *The Object Constraint Language.* Addison Wesley, 1999.

[10] IBM OCL parser

[11] Dresden OCL parser

Appendix: Wrapper Generation

In this report, we describe how to implement component assemblies with guarantees. To accomplish assembly with guarantees, we developed a design abstraction called a *mode component* that extends the GenVoca [1] model of layered assembly with explicit support for the assembly of interactive systems. The key extensions are:

1. An explicit representation of a (mode) component’s abstract state (a.k.a. *status*) in its interface;
2. A guarantee that the status representation remains up-to-date with respect to the actual state of the component; and
3. Support for configuring a hierarchical assembly of mode components so that changes in the status of a component can trigger changes in other components at higher levels in the assembly.

Currently, no programming langauge provides a construct with the declarative power of mode components. Thus we use a declarative specification language in conjunction with code generators to implement mode components and their assembly. This appendix describes our prototype code generator, which supports assembly in C++.

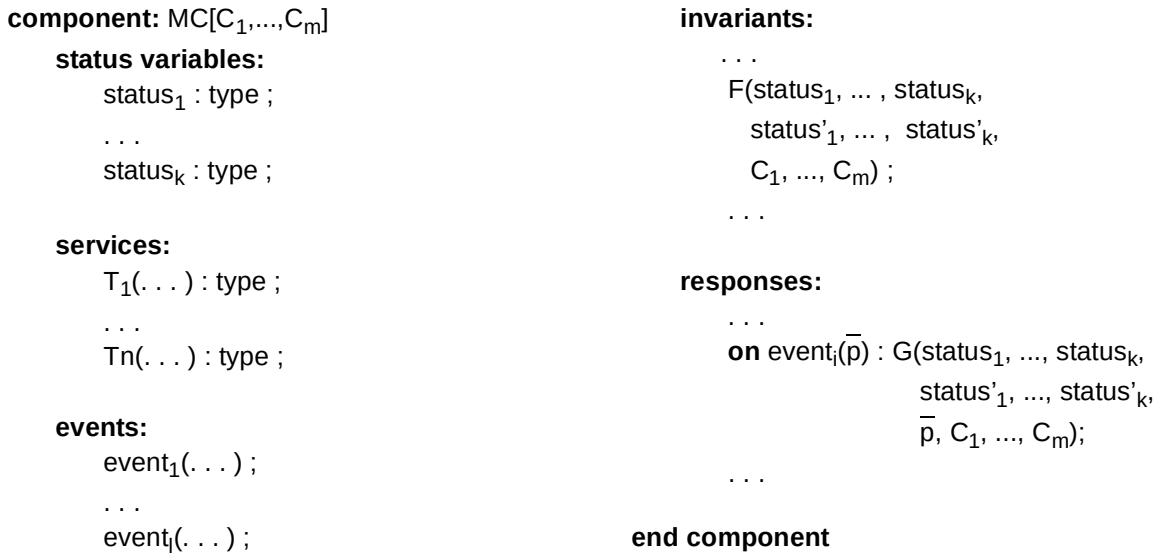

Figure 15. Schematic of a mode-component declaration.

A.1 Multicomponent Assembly Specifications

A *mode-component-assembly specfication (MCAS)* declares a collection of mode components and describes how to assemble this collection into a composite system using layering and intercomponent (status) constraints. Figure 15 provides a schematic view of the syntax of an MCAS. An MCAS is expressed as a stylized UML class model augmented with OCL constraints. In the sequel, we will not be concerned with the concrete syntax of an MCAS, but rather will concern ourselves with the abstract syntax, which is depicted as a UML model in Figure 16.

Our internal representation of a mode-component assembly is an instance of the class modeComponent in Figure 16. Notice that each mode component comprises three APIs, two of which (ServicesAPI) define component services and the third (StatusAPI) of which defines the status variables provided by the component. Additionally, each mode component is associated with zero or more statusAPIRefinements, which are generated from status invariants.

Each servicesAPI comprises one or more Services. We distinguish top services from bottom services. A mode component's *bottom services* are generated when invariants are compiled, and these services are visible only to generated code in inferior mode components. By contrast, a mode component's *top services* may be either generated or implemented by the original component developer, the latter being visible to code written by the developer of superior mode components and all of them being visible to generated code in superior components. A stand-alone mode component that has yet to be used in an assembly contains a top-services API that lists only the services implemented by the component developer and an empty bottom-services API. When a mode component is used in an assembly, the top- and bottom-services API may be extended with (generated) services that are used to maintain invariants among status variables.

We generate two kinds of services, StatusUpdateServices and StatusDelegateServices. A StatusUpdateService is a generated bottom service that reestablishes an invariant involving a dependent status variable based on a notification of a change to one of the independent status variables. Note that the dependent status variable must be in the statusAPI of the

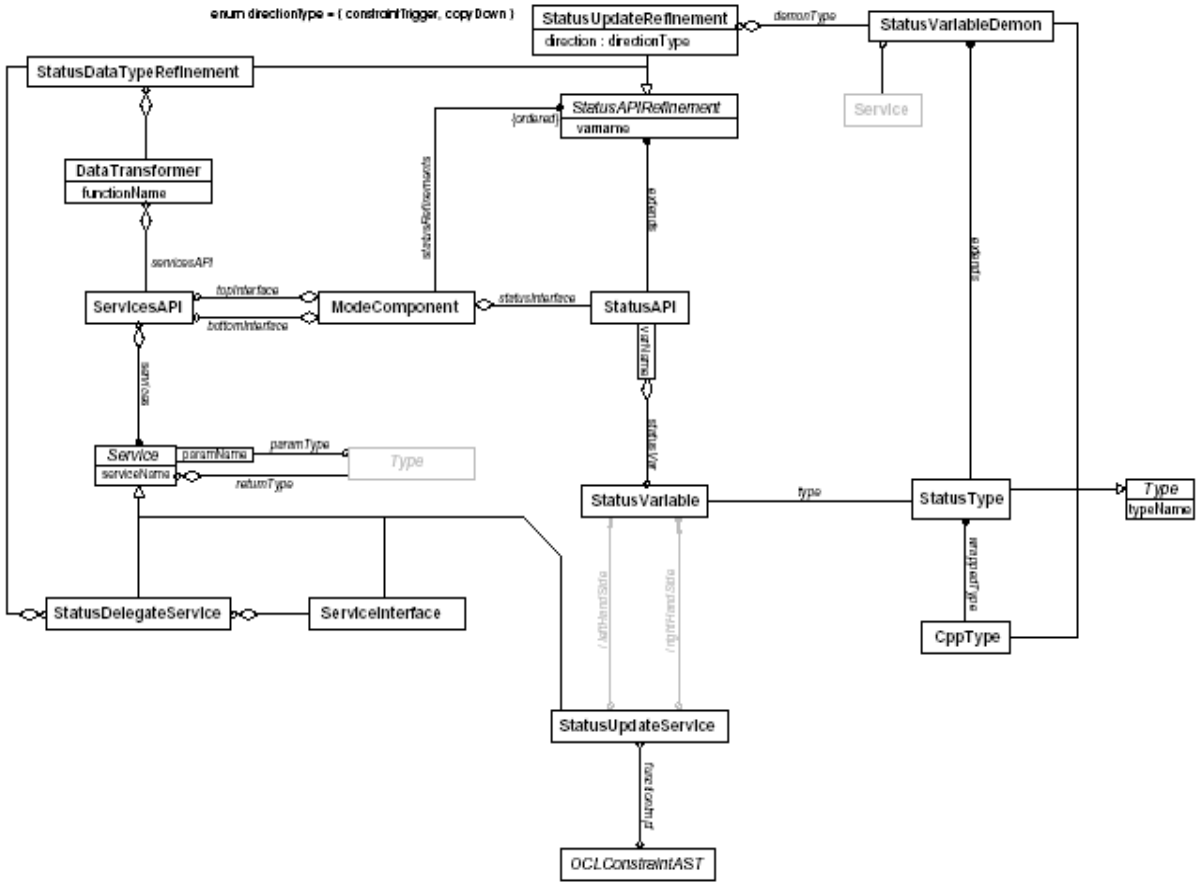

Figure 16. Abstract Syntax of an MCAS

mode component in which this service lives. The code for these services is generated from an OCL constraint. By contrast, a StatusDelegateService is generated in order to allow the value of a status variable in a superior component to flow down to an inferior component[6].

A statusAPI lists the status variables that are declared as part of the top interface of a mode component. Each status variable has a name and a statusType, which wraps a more primitive C++ type (CppType). Notice that the association between a status variable and its types is one-to-one: No two status variables are of the same status type, but two status variables might share the same wrapped type. Status types can be extended with reactive behavior via StatusVariableDemons, which invoke a service in some mode-component interface whenever the value of a given status variable changes. StatusVariableDemons are encapsulated in StatusUpdateRefinements, which extend statusAPIs (using mixin-layer composition). Notice that a StatusUpdateRefinement can propagate information either up the layering hierarchy or down. We call refinements of the former sort *constraint-trigger refinements*} and refinements of the latter sort *copy-down refinements*.

A.1.1 Data Transformers

Status variable types that are collections (i.e., sets and sequences) may be refined by so-

6. Status-delegate services are used to break invariant cycles.

called *data transformers,* which add behavior for computing and generating messages to maintain the consistency of collection objects whose contents are some function of the collection being refined. More precisely, a data transformer intercepts and transforms messages bound for the refined collection into new messages that are then forwarded to some target collection(s) provided the transformation succeeds. In addition to computing and forwarding these new messages, the original message is always forwarded to the originally intended recipient. We call these refinements data transformers because the new messages maintain a functional constraint between the original and the target collections.

An example is the *subsequence* transformer (not depicted in the figure), which can be applied to any sequence type. The subsequence transformer intercepts messages—such as insert, remove, append, or update—and then:

- forwards the message to the original sequence,
- computes a new message that is appropriate to forward to some target sequence, based on the constraint that the target must be a subsequence of the original, and then
- if an appropriate message can be constructed, forwards this new message to the target.

Suppose, for example, that a status variable named contents, whose type is a sequence of strings, is refined using a subsequence data transformer whose target is another status variable named view, whose type is also a sequence of strings. Suppose also, that the data transformer has been initialized with the indices used to extract the subsequence, and that these indices are 5 and 20. Then every message of the form insert(i, S) where i is a natural number and S is a string will engender a new message insert(j, S), where j = i - 5. Provided that j is between 0 and 15, the new message will be forwarded to view; otherwise the message is dropped.

A.1.2 Status Data-Type Refinements

A *status data-type refinement* collects one or more data transformers. Both status data-type refinements and status update refinements are statusAPIRefinements, which means that the statusAPI of a given mode component may be refined by one or more statusVariableDemons and one or more data transformers.

A.2 Code Generation

The OCL code generator is concerned with generating executable code from an MCAS AST. Many of the features of MCAS are declarative and must therefore be refined in order to render executable code. We exploit three distinct code-generation phases:

1. From each individual mode component declaration, generate a *component wrapper* that operationalizes the declarative features of the mode component's specification (e.g., status variables, event-response logic, and intra-component invariants);
2. From each mode-component assembly directive, which specifies how to connect two or more interacting mode components, generate a *connector* that will be used to compose the corresponding component wrappers; and
3. Configure the component wrappers and connectors to yield a *composite* that implements the MCAS.

Note that mode-component declarations and mode-component–assembly directives are specification constructs[7]; whereas component wrappers, connectors, and composites are code (e.g., C++) constructs. A key observation that we exploit in code generation is that component wrappers, connectors, and hand-coded implementations of the "functional part" of a mode component all can be represented explicitly as *mixin layers* [6], from which composites are easily assembled using C++ template instantiation. A mixin is a template class one of whose template parameters is the class's

7. i.e. syntactic elements in an MCAS.

superclass. A mixin layer is a set of mixin all residing as inner classes of the same class and all being instantiated together.

A mode component is a design abstraction, which is declared in an MCAS and ultimately implemented in C++. In addition to any application-specific functionality, the C++ implementation of a mode component must accomplish the following three tasks:

1. Represent status information explicitly in its modular interface;
2. Provide the (intra-component) infra-structure required to support inter-component composition; and
3. Provide run-time support for the up-to-date guarantee of status information.

Moreover, the implementation must accomplish these tasks as efficiently as possible. Task 1 is easily implemented by declaring status variables to be public data members of the C++ class that ultimately implements a mode-component specification. The key to supporting tasks 2 and 3 is to intercept assignments to variables in a program and be able to wrap them with additional functionality that maintains an intra-component invariant or that propagates a change in status to a dependant mode-component implementation.

To implement the interception and extension of assignments to program variables, we developed a micro-architecture[8] that separates the definitions of data types from the declarations of variables of these types. This separation allows for each functional extension to be expressed as a type refinement, which is encapsulated in a mixin layer. Separating a variable's type definition from its declaration allows a designer to extend a type by overriding the assignment operator with update actions without having to change the code that declares the variable or makes the assignment. Moreover, by encapsulating update actions into mixin layers, update actions may be easily composed.

A status variable is a program variable with the additional property that changes in its value can trigger updates of variables in mode components at higher levels in an assembly. To the implementor of a mode component, a status variable appears to be indistinguishable from any other local variable (i.e., it can be referenced and assigned to using standard C++ operators). We implement a status variable's update behavior by wrapping the definition of the type of the variable with a *listening agent* that exports the same abstract interface as the variable's type [7][8]. Thus, when an assignment statement within a mode component updates a status variable, it is actually invoking an operation in the wrapper, which delegates the operation to the wrapped object and then performs some additional processing, including notifying higher level components of changes.

We implement this behavior using a mixin class called a *status wrapper*, which is a listening agent that:

- is parameterized by the type that it wraps; and
- provides a default implementation of the assignment operation that can be over-ridden using inheritance.

When instantiated with a valid C++ type, a status wrapper yields a *status variable type* that is indistinguishable (in programming contexts) from the type that it wraps. Thus, each status variable is an instance (i.e., an object) of some status variable type. However, the wrapper provides additional hooks that allow (for example) assignments to a status variable to cause the invocation of other methods, such as constraint evaluation methods.

A.2.1 Status Type Refinement

A *status type refinement* refines a status ADT into a concrete type, which will be used

8. We use the term *microarchitecture* in order to distinguish it from the architecture of a mode-component assembly. Put simply, the microarchitecture is concerned with the design of a single mode component; whereas the architecture is concerned with the design of an assembly of multiple mode components.

to declare a private data member (instance variable) in the mode-component–implementation class. Each status ADT in our library has a default concrete type. In addition, a status constraint can be interpreted to specify how to refine a status ADT into a concrete type through a sequence of data transformers. A status-type refinement is then a composition of one or more data transformers, the result of which implements the interface of the status ADT.

For convenience, we group each status-type refinement into a mixin layer, and the service APIs are propagated without change using a technique called *bypassing* [1].

The generator is implemented as a visitor class called GenStatusTypeRefinement, which generates code as a side effect of traversing an AST whose root is an object of class ModeComponent. This generator class is declared as shown in Figure 17:

```
class GenStatusTypeRefinement :
   public MCASVisitor,
   protected GenMixinLayer {
   public:
      GenStatusTypeRefinement(ostream&,
         const string&);
      virtual ~GenStatusTypeRefinement() {}
   // Declaration of visit methods goes here
   protected:
      string modeComponentName;
};
```

Figure 17. Generator Class

The constructor takes two parameters: an opened output stream in which to emit the generated code, and a string that contains the name of the mode component that the generated code refines. The visit methods are declared exactly as prescribed by the visitor pattern; thus, for brevity, they have been elided in this presentation.

To generate the code for this refinement from a given ModeComponent AST, an instance of this visitor must be created and passed as a parameter to the Accept method of the object at the root of the AST. Consider, for example, the elided code in Figure 18:

```
(1) ModeComponent* mc = /* Create mode
                         component AST */;
(2) GenStatusTypeRefinement gen(...);
(3) mc->Accept(gen);
```

Figure 18. Visitor

Here, code is being generated as a result of the call to Accept on line 3. Commensurate with the operation of the visitor pattern, this statement causes the invocation of the visitModeComponent method with mc as a parameter. This method, the "entry point" into the logic for generating status type refinements, is described in Figure 19

```
void GenStatusTypeRefinement::
   visitModeComponent(
      const ModeComponent* mc)
{
   const string thisMixinName(
      modeComponentName +
   CG_STATUS_TYPE_REFINEMENTS_SUFFIX);
   // Emit: open mixin-layer declaration for
   // status-type refinements Figure 21
   indent++;
   mc->getStatusInterface()->Accept(*this);
   indent--;
   // Emit: bypass declaration for
   // top services API (Figure 26)
   // Emit: close mixin layer declaration (Figure 22)
}
```

Figure 19. Generate Status Type Refinements

To help illustrate each of these steps, we refer to Figure 20**,** which depicts this mixin layer for the scrollbar component that we used in the TextBrowser example.

A.2.1.1 Emitting Mixin Layer Declaration Infrastructure

This step in the generation process, shown in Figure 21, emits the code that "opens" the declaration of a status-type refinements mixin

```
openMultipleIncludeProtection(thisMixinName);
openMixinLayerClassDeclaration(thisMixinName);
```

Figure 21. Open mixin-layer declaration for status-type refinements

```
(1)  #include <MC/StatusImpls/ReactiveScalar.h>
     #include <MC/EventMgrs/EmptyScalarMgr.h>

(2)  template <class SB_FAMILY>
(3)  class sb_status_type_refinements : public SB_FAMILY {
(4)     public:

(5)        typedef typename SB_FAMILY::STATUS_POSITION STATUS_POSITION
(6)        typedef ReactiveScalar< EmptyScalarMgr< int > > STATUS_IMPL_POSITION;

(7)        typedef typename SB_FAMILY::SB SB;
(8)  };.
```

Figure 20. The file sb_status_type_refinements.H.

layer. In Figure 20, this code appears in lines 1-3. Similarly, "closing" the declaration is shown in Figure 22.

```
closeClassDeclaration();
closeMultipleIncludeProtection();
```

Figure 22. Close mixin layer declaration

A.2.1.2 Generating the Status-Type Refinement for each Status Variable

Figure 23 defines the symbol STATUS_IMPL_$status_i$ using a sequence of data transformers.

```
void GenStatusTypeRefinement::
  visitStatusVariable(const StatusVariable* sv)
{
   // Emit: bypass declaration of status ADT
   // See Figure 25.
   // Generate: refine status ADT using data
   // transformers. See Figure 24.
}
```

Figure 23. Generate impl declaration for status variable

A.2.1.3 Emitting Bypass Declarations

A bypass declaration is implemented by declaring type synonyms (`typedefs`) that are statically bound to types of the same name in the substrate mixin layer (see Figure 26). In Figure 20, lines 5 and 7 are examples of bypass declarations.

```
output
<< indent
<< TYPEDEF
<< CG_CONDITION_IMPL_MIXIN
<< OPEN_TEMPLATE_PARAMS
<< TYPENAME
<< mixinLayerParameterName
<< DOUBLE_COLON
<< sv->getType()->getTypeName()
<< CLOSE_TEMPLATE_PARAMS
<< sv->getType()->getTypeName()
<< SEMI_COLON
<< endl;
```

Figure 24. Generate: refine status ADT using data transformers

```
output
<< indent
<< TYPEDEF
<< TYPENAME
<< mixinLayerParameterName
<< DOUBLE_COLON
<< sv->getType()->getTypeName()
<< TAB
<< sv->getType()->getTypeName()
<< SEMI_COLON
<< endl;
```

Figure 25. Emit: bypass declaration of status ADT

```
indent++;
output
<< indent
<< TYPEDEF
<< TYPENAME
<< mixinLayerParameterName
<< DOUBLE_COLON
<< modeComponentName
<< TAB << modeComponentName
<< SEMI_COLON
<< endl
<< endl;
indent--;
```

Figure 26. Emit: bypass declaration for top services API

A.2.2 Constraint Trigger Refinement

A *constraint trigger refinement* is a mixin layer that refines the assignment operator of a status variable with functionality that causes an update in another component. One of these refinements must be generated for each external constraint—i.e. constraint in another mode component other—that depends on a status variable var_i in this mode component. Figure 27 illustrates such a refinement from the TextBrowser example.

The generator is implemented as a visitor[9] class called `GenConstraintTriggerRefinements`, which generates code as a side effect of traversing an AST whose root is an object of class `StatusAPIRefinement`. This generator class is declared as shown in Figure 28: The constructor takes two parameters: an opened output stream in which to emit the generated code, and a string that contains the name of the mode component that the generated code refines. The visit methods are declared exactly as prescribed by the visitor pattern; thus, for brevity, they have been elided in this presentation.

To generate the code for this refinement from a given `StatusAPIRefinement` AST, an instance of this visitor must be created

9. See the visitor design pattern in [4].

```
class GenConstraintTriggerRefinements :
   public MCASVisitor,
   protected GenMixinLayer {

 public:
    GenConstraintTriggerRefinements(ostream&,
       const string&);
    virtual ~GenConstraintTriggerRefinements() {}

    // Declare visit methods

  protected:
     string modeComponentName;
};
```

Figure 28. Class GenConstraintTrigger-Refinements

and passed as a parameter to the `Accept` method of the object at the root of the AST. Consider, for example, the elided code shown in Figure 29: Here, code is being generated as

```
(1) StatusAPIRefinement* ctr = /* Create CTR AST*
(2) GenConstraintTriggerRefinements gen(...);
(3) ctr->Accept(gen);
```

Figure 29. Visitor code generation

a result of the call to `Accept` on line 3. Commensurate with the operation of the visitor pattern, this statement causes the following method invocation:

```
gen.visitStatusAPIRefinement(ctr);
```

This method, the entry point into the logic for generating constraint trigger refinements, is described in Figure 30.

- Add an internal class $STATUS\_var_i$ that inherits from $MC\_FAMILY::STATUS\_var_i$ with a public and a private section.
- To this new internal class add the following to the public section:

```
typedef typename MC_FAMILY::STATUS_var_i
    parent_STATUS_var_i;
typedef typename
    parent_STATUS_var_i::DATATYPE DATATYPE;
```

```
(1) #ifndef SB_STATUS_CONSTRAINT_POSITION_UPDATES_VP_H
(2) #define SB STATUS_CONSTRAINT_POSITION_UPDATES_VP_H

(3) #include "vp.H"

(4) template <class SB_FAMILY>
(5) class sb status constraint position updates vp : public SB_FAMILY {
(6) public:

(7)   typedef typename SB_FAMILY::SB SB;

(8)   class STATUS POSITION : public SB FAMILY::STATUS POSITION _
(9)     public:

(10)      typedef typename SB FAMILY::STATUS_POSITION parent STATUS_POSITION;
(11)      typedef typename parent_STATUS_POSITION::DATATYPE DATATYPE;

(12)      STATUS_POSITION(const DATATYPE& d) :
(13)        SB_FAMILY::STATUS_POSITION(d),
(14)        vpP(0) {}

(15)      virtual STATUS_POSITION() {}

(16)      void initializeConstraint(vp::VP* ptr)
(17)        { vpP = ptr; }

(18)      void operator=(const DATATYPE& d)
(19)      {
(20)        parent STATUS_POSITION::operator=(d);
(21)        if (vpP) vpP->update();
(22)      }

(23)    private:
(24)      vp::VP* vpP;
(25)   };

(26) };

(27) #endif
```

Figure 27. Example: Constraint Trigger Refinement

```
void GenConstraintTriggerRefinements::visitStatusAPIRefinement(
      const StatusAPIRefinement* ctr)
{
   const string thisMixinName(CG_ConstraintUpdateRefinementFileName(
         modeComponentName,
         ctr->getDemonType()->getExtends()->getTypeName(),
         ctr->getReferences()->getName()));

      openMultipleIncludeProtection(thisMixinName);
      openMixinLayerClassDeclaration(thisMixinName);

      ctr->getDemonType()->Accept(*this);       // Generate code for the demon

      closeClassDeclaration();
       closeMultipleIncludeProtection();
}
```

Figure 30. Generate constraint trigger refinement

This idiom is used to derive the original parameter T in status<T>.

- Add the following function:

```
void initializeConstraint(other::OTHER *ptr) {
     other p = ptr;
}
```

- Overload assignment as follows:

```
void operator=(const DATATYPE& d) {
     parent STATUS_var_i::operator=(d);
     if (other p) other p->update();
}
```

- The `private` section should contain only one declaration:

```
other::OTHER *other p;
```

The code in Figure 33. adds constructor as follows:

```
STATUS_var_i(const DATATYPE& d) :
     parent_STATUS_POSITION(d),
other_p(0) {}
```

class. It can only be partially generated, the programmer must fill in details.

```
output << indent << TYPEDEF
       << TYPENAME << statusVarParentClass
       << TAB
       << superClassTypeName
       << SEMI_COLON << endl;

output << indent << TYPEDEF
       << TYPENAME << superClassTypeName
       << DOUBLE_COLON << DATA_TYPE
       << TAB << DATA_TYPE << SEMI_COLON
       << endl << endl;
```

Figure 32. Emit: type synonyms for status var type and status-var data type

and destructor code:

```
virtual STATUS_var_i() {}
```

A.2.2.1 Constraint Update Files

- Add to this class the following:

```
typedef typename MC_FAMILY::MC MC;
```

A.2.2.2 mc_impl.H

This file contains the outermost template that will serve as the implementation of the

```
void GenConstraintTriggerRefinements::visitStatusVariableDemon(
   const StatusVariableDemon* demon)
{
   const string statusVarType(demon->getExtends()->getTypeName());
   const string superClassTypeName(SUPERCLASS + statusVarType);
   const string svcCntnr(demon->getServiceToInvoke()->getServiceContainer()->
      getContContainer()->getName());

   const string serviceContainerPointerName(svcCntnr + "P");
   const string serviceContainerName(svcCntnr +
                                     mccg::toString(DOUBLE_COLON) +
                                     upperCase(svcCntnr));

   const string statusVarParentClass(mixinLayerParameterName +
                                     mccg::toString(DOUBLE_COLON) +
                                     statusVarType);

   openClassDeclaration(statusVarType, statusVarParentClass);

   // Emit: type synonyms for status var type and status-var data type (Figure 32.)

   // Emit: constraint-trigger constructor and destructor (Figure 33.)

   // Emit: assignment operations (Figure 34.)

   // Emit: initialize constraint method (Figure 35.)

   // Emit: private declarations (Figure 36.)

   closeClassDeclaration();
}
```

Figure 31. Generate status variable demon

```
output << indent << VOID << INITIALIZE_CONSTRAINT << OPEN_PAREN
       << serviceContainerName << STAR << PTR << CLOSE_PAREN << endl
       << indent << OPEN_BRACE
       << serviceContainerPointerName << GETS << PTR << SEMI_COLON
       << CLOSE_BRACE << endl;
```

Figure 35. Emit: initialize constraint method

- First create the class template mc_impl, which will contain only a public section:

```
template<class MC_FAMILY>
class mc_impl : public MC_FAMILY
```

- Next, for each status variable var_i, add the following typedef:

```
typedef typename MC_FAMILY::STATUS_var_i
   STATUS_var_i
```

```
output << indent << statusVarType
       << OPEN_PAREN << CONST << DATA_TYPE << AMPERSAND << "d" << CLOSE_PAREN
       << COLON << endl;

indent++;

output << indent << statusVarParentClass
       << OPEN_PAREN << "d" << CLOSE_PAREN << COMMA
       << endl
       << indent << serviceContainerPointerName << OPEN_PAREN
       << ZERO << CLOSE_PAREN << endl;

output << indent << OPEN_BRACE << CLOSE_BRACE
       << endl << endl;

indent--;

output << indent << VIRTUAL << TILDE << statusVarType
       << OPEN_PAREN << CLOSE_PAREN << OPEN_BRACE << CLOSE_BRACE
       << endl << endl;
```

Figure 33. Emit: constraint-trigger constructor and destructor

```
indent--;
output << indent << PRIVATE << COLON << endl;
indent++;

output << indent << serviceContainerName <<
    STAR << serviceContainerPointerName <<
    SEMI_COLON << endl << endl;
```

Figure 36. Emit: private declarations

- Next add an internal class MC as follows:

  ```
  class MC : public MC_FAMILY::MC
  ```

If the mode component is to inherit from another class (say, a window class in some GUI toolkit), then the programmer should add the inheritance in at this point. This class will contain a public and a private section, and possibly a protected section of the programmer chooses to add one.

This class should contain a pointer (private) to any (lower-level) mode component whose status variables it uses (a other::OTHER*), and these pointers, in turn, must be initialized somehow (in the constructor, say). It should contain a declaration for every virtual function declared in the first class created (mc::MC), and it must contain declarations of all of the status variables, in the form of STATUS_var$_i$ var$_i$. A file mc.cxx should be created with stubs for each of the methods in this class, and an empty constructor, and this file should be #included in mc impl.H (to bring all the template defintions into scope). The constructor for mc impl should call MC_FAMILY::MC, passing it the status variables.

A.2.3 Naming Conventions for Code-Generation Artifacts

Our prototype code generators processes MC and MC-assembly specifications to synthesize mixin layers, from which a composite can be constructed using C++ template instantiation. Given an MCAS with N mode components and M status constraints, we actually generate between 2N+M and 3N+M mixin layers. Each of these generated mixin layers is stored in a C++ file, each of which is an intermediate artifact of the assembly

```
output << indent << VOID << OPERATOR_ASSIGN
       << OPEN_PAREN << CONST << DATA_TYPE << AMPERSAND << "d" << CLOSE_PAREN
       << endl
       << indent << OPEN_BRACE
       << endl;

indent++;

output << indent << superClassTypeName << DOUBLE_COLON
       << OPERATOR_ASSIGN << OPEN_PAREN << "d" << CLOSE_PAREN << SEMI_COLON
       << endl;

output << indent
       << IF << OPEN_PAREN << serviceContainerPointerName << CLOSE_PAREN
       << OPEN_BRACE << endl;

indent++;
output << indent << serviceContainerPointerName << RIGHT_ARROW;

demon->getServiceToInvoke()->Accept(*this);

output << SEMI_COLON << endl;

indent--;

output << indent << CLOSE_BRACE << endl;

indent--;

output << indent << CLOSE_BRACE
       << endl << endl;
```

Figure 34. Emit: assignment operations

process. This section is concerned with the conventions used to name these intermediate files.

These naming conventions are used as follows. Assume the designer has created a mode-component specification that conforms to the schematic in Figure 16. From such a description, the code generator will produce at least two files:

```
InterfaceHeaderFile("MC") == MC.H
ImplHeaderFile("MC") == MC_impl.H
```

If the specification declares status variables, then the code generator will also produce the file:

```
StatusTypeRefinementFile("MC") ==
  MC_status_type_refinement.H
```

For each inter-component status constraint that connects status variable sv in component MC to an update in component MC2, the code generator produces the file:

```
ConstraintUpdateRefinementFile("MC", "sv",
  "MC2") ==
  MC_status_constraint_sv_updates MC2.H
```

The overall approach is illustrated in Figure 38.

```
 1 #ifndef _sb_implDOTh_
 2 #define _sb_implDOTh_

 3 #include <FL/Fl.H>
 4 #include <FL/Fl_Window.H>
 5 #include <FL/Fl_Int_Input.H>

 6 #include "fm.H"

 7 template <class SB_FAMILY>
 8 class sb_impl : public SB_FAMILY
 9 {
10 public:

11     typedef typename SB_FAMILY::STATUS_POSITION STATUS_POSITION;
12     class SB : public SB_FAMILY::SB,
13                public Fl_Window
14     {
15         public:

16             // Services
17             void setNumVisibleItems(int v)
18                 { numVisibleItems = v; }
19             void setPosition(int v)
20                 { _position = v; }

21             // Status Variables
22             STATUS_POSITION _position;

23             void userEvent(int);
24             void update();

25             SB(fm::FM*, int, int, int, int);

26         private:

27             Fl_Int_Input   input;
28             int            numVisibleItems;
29              fm::FM        *fmP;

30             void draw();
31     }
32 };

33 #include "sb.cxx"

34 #endif // !_sb_implDOTh_
```

Figure 37. sb_impl.h

```
const string DOT_H(".H");
const string CG_IMPL_SUFFIX("_impl");
const string CG_STATUS_TYPE_REFINEMENTS_SUFFIX("_status_type_refinement");
const string CG_UPDATE_REFINEMENT_INFIX1("_status_constraint_");
const string CG_UPDATE_REFINEMENT_INFIX2("_updates_");
const string CG_CONDITION_IMPL_MIXIN("condition_impl");
const string CG_STATUS_MIXIN("status");

inline string CG_InterfaceHeaderFileName(const string& mc)
{  return mc + DOT_H;  }

inline string CG_ImplHeaderFileName(const string& mc)
{  return mc + CG_IMPL_SUFFIX + DOT_H;  }

inline string CG_StatusTypeRefinementFileName(const string& mc)
{  return mc + CG_STATUS_TYPE_REFINEMENTS_SUFFIX + DOT_H;  }

inline string CG_ConstraintUpdateRefinementFileName(
    const string& srcMc,
    const string& statusVariable,
    const string& targetMc )
{  return srcMc + CG_UPDATE_REFINEMENT_INFIX1+ statusVariable +
          CG_UPDATE_REFINEMENT_INFIX2 + targetMc + DOT_H;  }
```

Figure 38. Declare generated-file naming conventions and macros