Design document for the Dynamo code generators

February 15, 2005

Chapter 1

Introduction

In DYNAMO, we are exploring 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 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 (top) interface;
2. a run-time guarantee that the status representation remains up-to-date with respect to the actual state of the component; and
3. support for configuring an hierarchical assembly of mode components so that changes in the status of a component can trigger changes in components at higher levels in the assembly.

Currently, no programming language 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 document describes our prototype code generator, which supports assembly in C++.

Chapter 2

Mode-component assembly specifications

component: MC[C_1,...,C_m]

status variables:
$status_1$: *type* ;
...
$status_k$: *type* ;

services:
$T_1(\ldots)$: *type* ;
...
$T_n(\ldots)$: *type* ;

events:
$event_1(\ldots)$;
...
$event_l(\ldots)$;

invariants:
...
$F(status_1, \ldots, status_k,$
$status'_1, \ldots, status'_k,$
$C_1, \ldots, C_m)$;
...

responses:
...
on $event_i(\overline{p})$: $G($ $status_1, \ldots, status_k,$
$status'_1, \ldots, status'_k,$
$\overline{p}, C_1, \ldots, C_m$);
...

end component

Figure 2.1: Schematic of a mode-component declaration.

A *mode-component–assembly specification* (MCAS) declares a collection of mode components and describes how to assemble this collection into a composite system using layering and inter-component (status) constraints. **Figure 2.1** illustrates the schematic view of the syntax of an MCAS. An MCAS may be expressed either directly, as a text file in the form depicted in **Figure 2.1**, or indirectly 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 2.2**.

Our internal representation of a mode-component assembly is an instance of the class model in **Figure 2.2**. Notice that each mode component comprises three APIs, two of which define component services and the third of which defines the status variables provided by the component. Additionally, each mode component is associated with 0 or more *status API refinements*, which are generated from

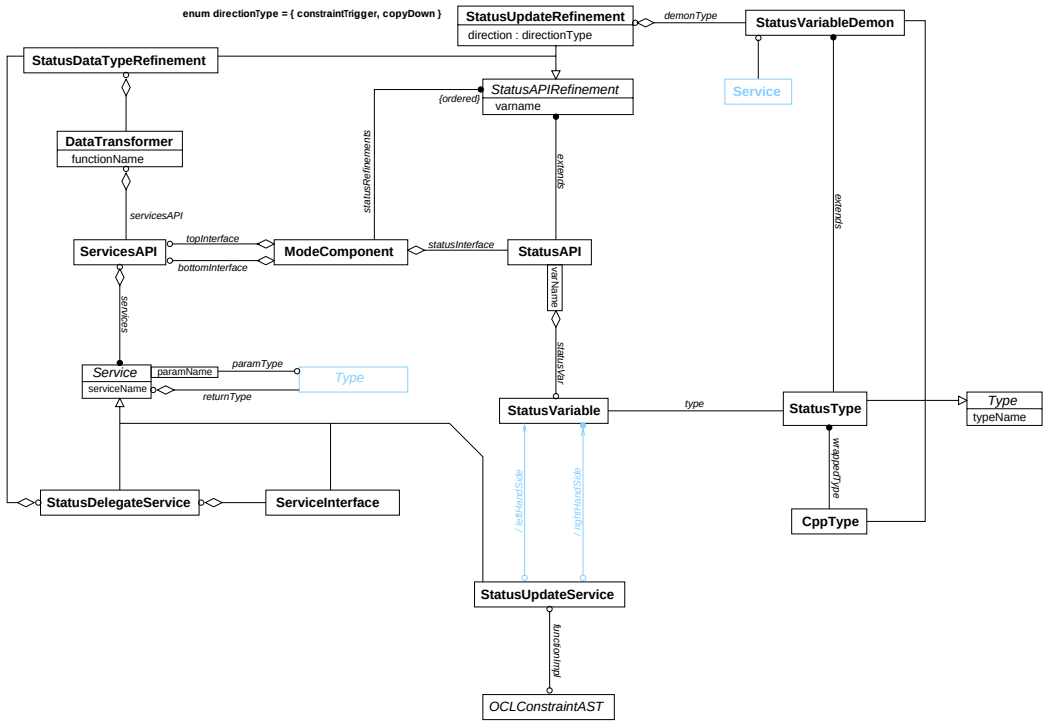


Figure 2.2: Abstract syntax of an MCAS

status invariants.

Each service API 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, *status update services* and *status-delegate services*. A status-update service 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 status API of the mode component in which this service lives. The code for these services is generated from an OCL constraint. By contrast, a status-delegate service is generated in order to allow the value of a status variable in a superior component to flow *down* to an inferior component.[1]

A *status API* 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 *status type*, which wraps a more primitive C++ type. Notice that the association between a status variable and its types is one-to-one: No two status

[1]These tend to be used to break invariant cycles.

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 *status variable demons*, which invoke a service in some mode-component interface whenever the value of a given status variable changes. Status variable demons are encapsulated in *status update refinements*, which extend status APIs (using mixin-layer composition). Notice that a status update refinement 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*.

Status variable types that are collections (i.e., sets and sequences) may be refined by so-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 a sequence—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 $j \in 0 \ldots 15$, the new message will be forwarded to `view`; otherwise the message is dropped.

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

Chapter 3

Code generation

The DYNAMO code generators are concerned with generating executable code from an MCAS AST. Many of the features of an 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[1]; 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* [SB98], from which composites are easily assembled using C++ template instantiation.

[[Need a paragraph on what a mixin layer is, plus the diagram]].

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 do the following:

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.

[1] i.e., syntactic elements in an MCAS

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 (or classes) that ultimately implement a mode-component specification. The key to supporting tasks 2 and 3 is to intercept "normal" 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 dependent mode-component implementation).

To implement the interception and extension of assignments to program variables, we developed a *micro-architecture*[2] 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 over-riding 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, a combination of update actions may be easily "plugged together".

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[3]. 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 in DYNAMO 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.

[[Wrapper diagram goes here...]] 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.

[2] We use the term micro-architecture in order to distinguish it from the "architecture" of a mode-component assembly. Put simply, the mirco-architecture 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.

[3] This idea was pioneered in work on the Chiron environment [YTT88, T+95].

4

3.1 Status type refinement

A *status type refinement* refines a status ADT into a concrete type, which will be used 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* [BG97][5].

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`. That is, as each node in the AST is traversed, code is generated using `...`. This generator class is declared:

7 ⟨*class GenStatusTypeRefinement* 7⟩≡

```
class GenStatusTypeRefinement : public MCASVisitor,
                                protected GenMixinLayer {
 public:
  GenStatusTypeRefinement(ostream&, const string&);
  virtual ~GenStatusTypeRefinement() {}

  ⟨Declare visit methods ??⟩

 protected:
  string          modeComponentName;
};
```

[4] Need a brief intro to the literate programming conventions.

[5] Need to define bypassing, possibly in its own paragraph.

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:

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

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 Chunk 9a.

```
(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 3.1: The file `sb_status_type_refinements.H`.

9a ⟨*Generate status type refinements* 9a⟩≡

```
void GenStatusTypeRefinement::visitModeComponent( const ModeComponent* mc )
{
  const string thisMixinName(modeComponentName +
                             CG_STATUS_TYPE_REFINEMENTS_SUFFIX);

  ⟨Emit: open mixin-layer declaration for status-type refinements 9b⟩

  indent++;
  mc->getStatusInterface()->Accept(*this);
  indent--;

  ⟨Emit: bypass declaration for top services API 11b⟩

  ⟨Emit: close mixin layer declaration 10a⟩
}
```

To help illustrate each of these steps, we refer to **Figure 3.1**, which depicts this mixin layer for the scrollbar component that we used in the Baltimore file-browser demo.

3.1.1 Emitting mixin-layer declaration infra-structure

9b ⟨*Emit: open mixin-layer declaration for status-type refinements* 9b⟩≡

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

This step emits the code that "opens" the declaration of a status-type refinements mixin layer. In **Figure 3.1**, this code appears in lines 1-3.

10a ⟨*Emit: close mixin layer declaration* 10a⟩≡

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

3.1.2 Generating the status-type refinement for each status variable

10b ⟨*Generate impl declaration for status variable* 10b⟩≡

```
void GenStatusTypeRefinement::visitStatusVariable( const StatusVariable* sv )
{
    ⟨Emit: bypass declaration of status ADT 11a⟩
    ⟨Generate: refine status ADT using data transformers 10c⟩
}
```

10c ⟨*Generate: refine status ADT using data transformers* 10c⟩≡

```
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;
```

This statement defines the symbol `STATUS_IMPL_`$\underline{\text{status}_i}$ using a sequence of data transformers.

3.1.3 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. In **Figure 3.1**, lines 5 and 7 are examples of bypass declarations.

11a ⟨*Emit: bypass declaration of status ADT* 11a⟩≡

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

11b ⟨*Emit: bypass declaration for top services API* 11b⟩≡

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

3.2 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 3.2** illustrates an example of such a refinement from the Baltimore demo.

The generator is implemented as a visitor class[6] 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:

12 ⟨*class GenConstraintTriggerRefinements* 12⟩≡

```
class GenConstraintTriggerRefinements : public MCASVisitor,
                                        protected GenMixinLayer {
 public:
  GenConstraintTriggerRefinements( ostream&, const string& );
  virtual ~GenConstraintTriggerRefinements() {}

  ⟨Declare visit methods ??⟩

 protected:
  string     modeComponentName;
};
```

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 and passed as a parameter to the `Accept` method of the object at the root of the AST. Consider, for example, the elided code:

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

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 following method invocation:

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

This method, the "entry point" into the logic for generating constraint trigger refinements, is described below.

[6] See the visitor design pattern in [GHJV95]

```
(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 3.2: Example: constraint trigger refinement

14 ⟨*Generate constraint trigger refinement* 14⟩≡

```
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();
}
```

- 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;
```

  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;
```

15 ⟨*Generate status variable demon* 15⟩≡

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

  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 16a⟩

  ⟨Emit: constraint-trigger constructor and destructor 16b⟩

  ⟨Emit: assignment operations 17a⟩

  ⟨Emit: initialize constraint method 17b⟩
```

⟨*Emit: private declarations* 18⟩

```
    closeClassDeclaration();
  }
```

16a ⟨*Emit: type synonyms for status var type and status-var data type* 16a⟩≡

```
    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;
```

16b ⟨*Emit: constraint-trigger constructor and destructor* 16b⟩≡

```
    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;
```

This code adds constructor as follows:

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

and destructor code:

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

17a ⟨*Emit: assignment operations* 17a⟩≡

```
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;
```

17b ⟨*Emit: initialize constraint method* 17b⟩≡

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

18 ⟨*Emit: private declarations* 18⟩≡

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

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

3.2.1 Constraint update files

- Add to this class the following:

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

3.2.2 mc_impl.H

This file contains the outermost template that will serve as the implementation of the class. It can only be partially generated, the programmer must fill in details.

- 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
  ```

- 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, at least (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 definitions into scope). The constructor for *mc*`_impl` should call *MC*`_FAMILY::`*MC*, passing it the status variables.

sb_impl.H

```
 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

31      };
32  };
```

```
33  #include "sb.cxx"

34  #endif // !_sb_implDOTh_
```

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

22 ⟨*Declare generated-file naming conventions and macros* 22⟩≡

```
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;
}
```

These naming conventions are used as follows. Assume the designer has created a mode-component specification that conforms to the schematic in **Figure 2.1**. 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 produce the file:

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

or each inter-component status constraint that connects status variable *sv* in component *MC* to an update in component *MC*2, the code generator produces the file:

`ConstraintUpdateRefinementFile("MC", "sv", "MC2")` ==
MC_status_constraint_sv_updates_MC2.H

Bibliography

[BG97] D. Batory and B. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Trans. Softw. Eng.*, 23(2):67–82, February 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley Publishing Company, Reading, Massachusetts, 1995.

[SB98] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. of the 12th European Conference on Object-oriented Programming*, 1998.

[T+95] R. N. Taylor et al. Chiron-1: a software architecture for user interface development, maintenance, and run-time support. *ACM Trans. on Computer-Human Interaction*, 2(2):105–144, June 1995.

[YTT88] M. Young, R. N. Taylor, and D. B. Troup. Software environment architectures and user-interface facilities. *IEEE Trans. Softw. Eng.*, 14(6):697–708, June 1988.

Index

data transformer, 7

status ADT, 7
status type refinement, **7**