SOFTWARE ENGINEERING NOTES 19, 4 (OCT. 1994), 40-51.

Part III: Implementing Components in RESOLVE

Paolo Bucci
Joseph E. Hollingsworth
Joan Krone
Bruce W. Weide

A *realization module* is a RESOLVE unit containing code that implements the behavior of an associated conceptual module. The purpose of this paper is to explain realization modules, assuming that you have read the companion paper [Part II] explaining conceptual modules.

The organization is as follows. First we present some example realization modules with little elaboration. Then we discuss many aspects of realization modules in general, using the examples for illustration. Finally, we give an overview of the basis for modular verification of the correctness of realizations. The focus throughout is on the software component engineering rationale for some slightly unusual language features.

1. A Partial Taxonomy of Realizations (With Examples)

The simple examples in Figures 1–6 include most features of RESOLVE realization modules. Ada and C++ counterparts for pieces of these examples are provided in a companion paper [Part IV] and also are available on-line. You can consider the Ada/C++ versions to be the code we would generate if Ada/C++ were the target language of a RESOLVE compiler; or, in the absence of a RESOLVE compiler, you can consider these as the recommended Ada/C++ code to write when observing the RESOLVE discipline. By comparing Ada/C++ to RESOLVE code you should be able to understand the aspects of realization modules that we do not discuss explicitly in this paper.

While reading the examples, keep in mind that realization modules have several important properties that we do not discuss further:

- There can be many realization modules for each conceptual module.
- Each realization module comes in two parts: a *realization header* that contains information that a client needs to know in order to use the realization; and a *realization body* that contains everything else. A client sees only the header.
- The context of a realization header automatically includes that of the conceptual module it claims to implement, and the context of a realization body additionally includes that of the realization header.

- A realization module contains enough information to prove that it correctly implements the associated concept. There is no need to know anything else about the client that might use it. It therefore contains, for example, various formal assertions that are not required in most other languages.

The examples have been chosen in part to illustrate the two main categories that we suggest as the primary dimension for organizing RESOLVE realizations:

- A *fully-parameterized* realization — Figures 1–5 depict fully-parameterized realizations, i.e., ones in which all realization context that can be parametric actually is parametric. Figures 1–3 deal with variations on Stack_Template [Part II]. Figures 4 and 5 show two variants of an implementation of Partial_Map_-Template [Part II] using Stacks — an unusual but illustrative example that is easy to explain because we already have seen specifications for the concepts involved. The key thing to note about these realizations is that all types and operations used in the code are parameters to the concept or to the realization. This gives a client potentially great flexibility in choosing a performance profile for exported operations, but also encumbers him or her with great responsibility for selecting the "right" realizations for facility parameters [Sitaraman 92b].

- A *partial instantiation* of another realization — Figure 6 shows a second (closely related) implementation of Partial_Map_Template that is less general than the ones in Figures 4 and 5, but easier for a client to instantiate. The technique used here is to create a wrapper for a fully-parameterized realization, fixing some or all of the latter's realization parameters so a client does not have to do it.

We recommend that a mature component library should contain a fully-parameterized realization for each implementation strategy (i.e., for each major decision about data representations and algorithms). It also should contain some partial instantiations of that realization that meet the most common needs of clients, especially for use in early development of large systems before detailed performance concerns dominate the process. When performance tuning becomes more important, it is easy to create new partial instantiations that have different performance characteristics and to plug and unplug functionally identical realizations, with only local changes in a few declarations.

You can organize realizations along orthogonal dimensions, too, making it easy for a client to find the right implementation of a concept for the circumstances. For example, in order to facilitate modular testing and debugging we recommend constructing a layered "checking" realization (which reports precondition violations) and a layered "display" operation (which outputs a human-readable mathematical model value for an ADT) for each kernel concept. Details of these suggestions are beyond the scope of this paper but are covered elsewhere [Hollingsworth 92b].

We now briefly introduce the realization module examples. Figure 1 gives an obvious layered implementation for Stack_Top_Capability, which exports an operation to return a copy of the top Item of a Stack. The RESOLVE discipline suggests that this should not

be a primary operation [Part II], and the example shows why: It is easy to implement it by layering, gaining significantly in reusability while incurring at most a small constant factor performance penalty. Figure 2 gives another example of this kind. The highlight here is that a single piece of code can be instantiated in so many different ways, each of which copies a Stack from one representation to another. A related concept (not shown here) "converts" a Stack from any representation to any other by consuming, not preserving, the first argument of the exported operation. A single implementation of this concept does all the work of what would seem to require at least linearly, if not quadratically, many separate translators in a traditional approach [Sitaraman 92a].

The realizations in Figures 1 and 2 apparently have the same name: "Obvious". In fact, the names are "Obvious (for) Stack_Top_Capability" and "Obvious (for) Stack_-Replica_Capability" because each realization module is associated with a concept that it purports to implement correctly. (It is a separate matter to verify that this claim holds.)

Figure 1: One Realization for Stack_Top_Capability

```
realization header Obvious for Stack_Top_Capability

   context

      parametric context

         operation Replica (
                 preserves x: Item
              ): Item
            ensures   Replica = x

end Obvious


realization body Obvious for Stack_Top_Capability

   interface

      operation Top (
              preserves s: Stack
           ): Item
         context
            variables
               x, x_copy: Item
         begin
            Pop (s, x)
            x_copy := Replica (x)
            Push (s, x_copy)
            return x
         end Top

end Obvious
```

Figure 2: One Realization for Stack_Replica_Capability

```
realization header Obvious for Stack_Replica_Capability

   context

      parametric context

         operation Replica (
                preserves x: Item
             ): Item
           ensures   Replica = x

end Obvious


realization body Obvious for Stack_Replica_Capability

   context

      global context

         mathematics STRING_REVERSE_MACHINERY

      local context

         math facility STRING_REVERSE_FACILITY is
            STRING_REVERSE_MACHINERY (math[Item])

   interface

      operation Replica (
             preserves s: Stack_Facility_1.Stack
          ): Stack_Facility_2.Stack
         context
            variables
               s_copy, catalyst: Stack_Facility_2.Stack
               x, x_copy: Item
         begin
            loop maintaining
                  REVERSE (catalyst) * s = REVERSE (#catalyst) * #s
               while Length (s) > 0 do
                  Pop (s, x)
                  Push (catalyst, x)
            end loop
            loop maintaining
                  REVERSE (catalyst) * s = REVERSE (#catalyst) * #s  and
                  s_copy = s
               while Length (catalyst) > 0 do
                  Pop (catalyst, x)
                  x_copy := Replica (x)
                  Push (s, x)
                  Push (s_copy, x_copy)
            end loop
            return s_copy
         end Replica

end Obvious
```

Figure 3 shows the simplest way to realize Replicable_Stack_Template, which is specified as an enhancement of Stack_Template [Part II]. Here, a client has to supply as realization parameters instances of Stack_Template and Stack_Replica_Capability, and this realization simply re-exports their interfaces. At first glance, there is apparently little "value added" when such a concept — whose interface is specified to be simply a combination of other interfaces — is implemented by bundling together realizations of those interfaces. But the fact that Stack_Facility and Stack_Replica_Facility are realization parameters defers to the client the decision of which realizations of Stack_Template and Stack_Replica_Capability to use. This feature gives a client great performance flexibility.

Still, in the bundled realization there is no way for a client to avoid linear-time performance of the Replica operation for Stacks. It is beyond the scope of this paper to explain how, but we note that it is possible to implement Replicable_Stack_Template "from scratch" (i.e., without using existing realizations of Stack_Template or Stack_Replica_Capability) so that all exported operations run in constant time. Such a realization would look more like Figure 4, since the interface would not use re-exporting.

Figure 3: One Realization for Replicable_Stack_Template

```
realization header Bundled for Replicable_Stack_Template

   context

      parametric context

         facility Stack_Facility is Stack_Template (Item)

         facility Stack_Replica_Facility is Stack_Replica_Capability
               (Item, Stack_Facility, Stack_Facility)

end Bundled


realization body Bundled for Replicable_Stack_Template

   interface

      re-exports

         Stack_Facility
         Stack_Replica_Facility

end Bundled
```

Figure 4 shows how you might implement Partial_Map_Template [Part II] by representing a Partial_Map as a Stack. While this realization is suboptimal from the performance standpoint, it is straightforward but not trivial and it does help illustrate most of the remaining aspects of realization modules. It also emphasizes a very important point: Other more traditional realizations (e.g., hashing and binary search trees) clearly would require different realization parameters (e.g., a hash operation and a comparison operation for D_Items, respectively) [Sitaraman 92a]. Careful separation of specification and implementation demands separate parameterization of concepts and realizations because different implementation strategies have different needs for additional client-supplied information.

This example also involves non-trivial local context, including some mathematical machinery [Heym 94a] and two local operations. In order to understand this realization, in addition to what you know from the companion paper on specifying concepts [Part II] you need to know that OCCURS_COUNT(s,x) is the number of occurrences of entry x in string s; REVERSE (s) is the string consisting of the entries of s in reverse order; and ELEMENTS(s) is the set containing the entries occurring in string s.

Figure 4: A Fully-Parameterized Realization of Partial_Map_Template

```
realization header Unordered_Stack for Partial_Map_Template

   context

      global context

         concept Record2_Template
         concept Stack_Template

      parametric context

         facility D_R_Pair_Facility is Record2_Template (D_Item, R_Item)

         facility Stack_Facility is Stack_Template (Record2)

         operation Are_Equal (
               d1: D_Item
               d2: D_Item): Boolean
            ensures   Are_Equal  iff  d1 = d2

end Unordered_Stack


realization body Unordered_Stack for Partial_Map_Template

   context

      global context
```

```
   mathematics STRING_OCCURS_COUNT_MACHINERY
   mathematics STRING_REVERSE_MACHINERY
   mathematics STRING_ELEMENTS_MACHINERY

local context

   renaming
      Record2 as D_R_Pair
      field1 as domain_value
      field2 as range_value

   math facility STRING_OCCURS_COUNT_FACILITY is
      STRING_OCCURS_COUNT_MACHINERY (math[D_R_Pair])

   math facility STRING_REVERSE_FACILITY is
      STRING_REVERSE_MACHINERY (math[D_R_Pair])

   math facility STRING_ELEMENTS_FACILITY is
      STRING_ELEMENTS_MACHINERY (math[D_R_Pair])

   math operation CONTAINS (
           s: string of math[D_R_Pair]
           d: math[D_Item])
        ): boolean
      definition   CONTAINS (s, d)  iff
                   there exists r: math[R_Item]
                      (OCCURS_COUNT (s, (d,r)) > 0)

   math operation CONTAINS_AS_FIRST_ENTRY (
           s: string of math[D_R_Pair]
           d: math[D_Item])
        ): boolean
      definition   CONTAINS_AS_FIRST_ENTRY (s, d)  iff
                   there exists r: math[R_Item],
                      t: string of math[D_R_Pair] (s = <(d,r)> * t))

   operation Pop_Until_Passed (
           alters    s1: Stack
           alters    s2: Stack
           preserves d: D_Item
        )
      requires  s2 = empty_string
      ensures   REVERSE (s2) * s1 = #s1  and
                (if     s2 = empty_string
                 then   not CONTAINS (s1, d)
                 else   CONTAINS_AS_FIRST_ENTRY (s2, d))
      context
         variables
            found: Boolean
            p: D_R_Pair
      begin
         loop maintaining
               not CONTAINS (s2, d)  and
               REVERSE (s2) * s1 = REVERSE (#s2) * #s1
            if Length (s1) = 0
               then
```

```
                  s1 :=: s2
                  exit
               end if
               Pop (s1, p)
               found := Are_Equal (p.domain_value, d)
               Push (s2, p)
               if found
                  then
                     exit
               end if
            end loop
         end Pop_Until_Passed

      operation Combine (
               alters    s1: Stack
               consumes  s2: Stack
            )
         ensures    s1 = REVERSE (#s2) * #s1
         context
            variables
               p: D_R_Pair
         begin
            loop maintaining
                  REVERSE (s2) * s1 = REVERSE (#s2) * #s1
               while Length (s2) > 0 do
                  Pop (s2, p)
                  Push (s1, p)
            end loop
         end Combine

interface

   type Partial_Map is represented by Stack
      convention      for all d: math[D_Item], r: math[R_Item]
                         (OCCURS_COUNT (m.rep, (d,r)) <= 1)
      correspondence  m = ELEMENTS (m.rep)

   operation Define (
            alters    m: Partial_Map
            consumes  d: D_Item
            consumes  r: R_Item
         )
      begin
         Push (m.rep, (d, r))
      end Define

   operation Undefine (
            alters    m: Partial_Map
            preserves d: D_Item
            produces  d_copy: D_Item
            produces  r: R_Item
         )
      context
         variables
            catalyst: Stack
      begin
```

```
            Pop_Until_Passed (m.rep, catalyst, d)
            Pop (catalyst, (d_copy, r))
            Combine (m.rep, catalyst)
         end Undefine

      operation Undefine_Any_One (
              alters    m: Partial_Map
              produces  d: D_Item
              produces  r: R_Item
            )
         begin
            Pop (m.rep, (d, r))
         end Undefine_Any_One

      operation Is_Defined (
              preserves m: Partial_Map
              preserves d: D_Item
            ): Boolean
         context
            variables
               catalyst: Stack
               defined: Boolean
         begin
            Pop_Until_Passed (m.rep, catalyst, d)
            if Length (catalyst) > 0
               then
                  Combine (m.rep, catalyst)
                  defined := true
            end if
            return defined
         end Is_Defined

      operation Size (
              preserves m: Partial_Map
            ): Integer
         begin
            return Length (m.rep)
         end Size

end Unordered_Stack
```

A client of the realization in Figure 4 must instantiate Record_2_Template and Stack_Template (in a certain way) before instantiating realization Unordered_Stack for Partial_Map_Template. The parametric context section of the realization explains what must be done, but it seems odd that the client should have to instantiate components used in the bowels of the representation. The apparent reason for this decision is to give the client performance flexibility.

But this flexibility is available even if the client just selects appropriate realizations of Record_2_Template and Stack_Template and leaves their use in instantiation to the

implementer of Unordered_Stack [Sitaraman 92b]. In RESOLVE, this can be described by changing the realization as shown in Figure 5.

Although we prefer the approach of Figure 5 for many reasons, our current component library uses the approach of Figure 4 because it can be directly translated into Ada or C++ code [Part IV]. These languages have no counterpart for an uninstantiated realization as a realization parameter.

Figure 5: Alternate Fully-Parameterized Realization of Partial_Map_Template

```
realization header Unordered_Stack_Alt for Partial_Map_Template

   context

      global context

         concept Record2_Template
         concept Stack_Template

      parametric context

         realization Record2_Realization for Record2_Template

         realization Stack_Realization for Stack_Template

         operation Are_Equal (
               d1: D_Item
               d2: D_Item): Boolean
            ensures   Are_Equal  iff  d1 = d2

end Unordered_Stack_Alt


realization body Unordered_Stack_Alt for Partial_Map_Template

   context

      ...

      local context

         facility D_R_Pair_Facility is Record2_Template (D_Item, R_Item)
            realized by Record2_Realization

         facility Stack_Facility is Stack_Template (Record2)
            realized by Stack_Realization

         ...
```

Figure 6 shows the fully parameterized Unordered_Stack realization of Figure 4 wrapped inside a new realization module that has as little parametric context as possible.

Figure 6: Partial Instantiation of Unordered_Stack

```
realization header Unordered_Stack_Fixed for Partial_Map_Template

   context

      parametric context

         operation Are_Equal (
              d1: D_Item
              d2: D_Item): Boolean
           ensures   Are_Equal  iff  d1 = d2

end Unordered_Stack_Fixed


realization body Unordered_Stack_Fixed for Partial_Map_Template

   context

      global context

         concept Record2_Template
         concept Stack_Template
         realization Standard for Record2_Template
         realization List for Stack_Template
         realization Unordered_Stack for Partial_Map_Template

      local context

         facility D_R_Pair_Facility is Record2_Template (D_Item, R_Item)
           realized by Standard

         facility Stack_Facility is Stack_Template (Record2)
           realized by List

         facility Partial_Map_Facility is Partial_Map_Template
              (D_Item, R_Item)
           realized by Unordered_Stack
              (D_R_Pair_Facility, Stack_Facility, Are_Equal)

   interface

      re-exports

         Partial_Map_Facility

end Unordered_Stack_Fixed
```

For obvious reasons, this approach to realization is called partial instantiation. By comparing the realization header of Figure 6 to those of Figures 4 and 5, you can see that the fully parameterized versions ask a client to supply either instances or realizations of Record2_Template and Stack_Template. With partial instantiation the implementer selects these realizations and declares instances in the local context section of the realization body. This trades client flexibility in tuning performance for simplified client instantiation.

Clearly, you can create similar partially instantiated versions of Unordered_Stack in which only one of the two facility parameters is fixed. In general, if there are k facility parameters to a fully parameterized realization module, there are $2^k - 1$ partial instantiations of this kind, and for each of these there are as many choices for the realizations of the fixed facilities as there are different realizations of those concepts. So even in a component library with only two or three major implementation variants for each concept, the combined power of layering and fully parameterized realizations leads to a combinatorially large number of performance profiles even for a single realization of a single concept [Sitaraman 92b]. This is one reason that we do not immediately concede the need for libraries of thousands of vaguely related components. Our components are organized by concept, by realization style (fully parameterized or partial instantiation), and along the other orthogonal realization dimensions mentioned earlier. This relatively small library of carefully designed components stands in place of a huge library of components that might be built without heavy use of layering or fully parameterized realizations.

2. Realization Headers

Now we present most of the syntax for realization modules, with the help of some of the non-terminal symbols and the conventions followed in the companion paper [Part II] and a supporting report on mathematical foundations [Heym 94a]. Accompanying discussion centers on the need for and use of the most important features in disciplined software component engineering.

We begin with realization headers. As noted previously, every realization module is physically divided into two parts.[1] The realization header may contain only a context section that adds to the context of the concept it implements. The context section itself does not appear if there is no additional parametric context:

---

[1] The fine structure of realization modules surely will change in future versions of RESOLVE. For example, there might be more than one realization body for each realization header, and there might be a new kind of module between concepts and realizations to contain non-functional (e.g., performance) specifications [Sitaraman 94]. Details of these revisions need to be worked out carefully in order to retain the modularity-of-reasoning property.

```
<realization_header> ::=
   realization header <realization_module_id>
      for  <conceptual_module_id>
         [<realization_header_context_section>]
   end <realization_module_id>
```

The realization header includes information that a client of a realization *must know* in order to instantiate it, and nothing else. Specifically, it lists all additional formal parameters to the realization and additional context (e.g., mathematical machinery) needed to explain them to a client. All remaining information beyond what is in the associated concept belongs in the realization body; see Section 3.

A realization header can add three different kinds of context: global, parametric, and local; but it must have parametric context:

```
<realization_header_context_section> ::=
   context
      [<concept_global_context_section>]
      <realization_header_parametric_context_section>
      [<realization_header_local_context_section>]
```

Global context serves the same role as in conceptual modules, and in realization headers it has the same syntax and semantics as for concepts. The need for and use of global context is illustrated in Figures 4 and 5, where the parametric context includes instances or just realizations of two concepts (Record2_Template and Stack_Template) that are not in the conceptual context of Partial_Map_Template — because they have nothing to do with explaining its functionality.

Record2_Template takes as parameters two types, called Item1 and Item2. Their mathematical models are the math types of two fields called field1 and field2 in the tuple that models exported type Record2. The use of instances of Record2_Template in the realization body involves special syntax as a matter of convenience (as explained further in Section 3.2.3), but this special nature is not seen in the realization header.

Parametric context is a bit different from that of conceptual modules because a realization can ask a client to provide slightly different information than is needed for a concept. There are five kinds of parameters to realization modules: (program) constants, (program) facilities, and math operations as in conceptual modules, and now additionally (program) operations and realizations:

```
<realization_header_parametric_context_section> ::=
   parametric context
      <realization_header_parameter_sequence>
      [<concept_parametric_context_restriction>]

<realization_header_parameter> ::=
   <concept_constant_parameter> |
   <concept_facility_parameter> |
   <concept_math_operation_parameter> |
   <realization_header_operation_parameter> |
   <realization_header_realization_parameter>

<realization_header_operation_parameter> ::=
   <operation_declaration>

<realization_header_realization_parameter> ::=
      realization <realization_id> for <concept_id>
```

Constant, facility, and math operation parameters are the same as for conceptual modules, and the **`restriction`** clause is also identical. Types need not be allowed as realization parameters because the only possible uses for additional types in a realization body demand that particular operations should be available for them. Additional types used in realizations, therefore, come from facilities that are parameters to the realization or that are locally declared in the realization body. These facilities export the needed operations along with their types.

The new kind of formal parameter to realizations is the operation parameter, the use of which is demonstrated in Figures 1, 2, 4, and 5. In each example, the realization needs to invoke an operation which does not appear in the concept (where it is not needed to explain functionality) and which cannot be declared locally within the realization. Therefore, it must be supplied by a client as a parameter.

Local context for realization headers also is a restricted version of that for concepts, in that it may not contain facility declarations:

```
<realization_header_local_context_section> ::=
   local context
      <realization_header_local_context_item_sequence>

<realization_header_local_context_item> ::=
   <math_subtype_declaration> |
   <math_operation_header_declaration> |
   <math_operation_definition_declaration>
```

Local context in realization headers is not illustrated in the examples. However, it can be used to declare additional mathematical machinery involved in pre- and postconditions for operation parameters, and for stating restrictions on parametric context.

3. Realization Bodies

The realization body contains the remainder of the realization module:

```
<realization_body> ::=
   realization body <realization_module_id>
      for  <conceptual_module_id>
         [<realization_body_context_section>]
         realization_body_interface_section
   end <realization_module_id>
```

In addition to an optional context section, a correct realization body has an interface section that declares a representation for each type exported by the concept it claims to implement, a code body for each operation exported by that concept, and no more.

3.1. Context

The context section of a realization body may introduce additional global and local context — but not parametric context, which must be in the realization header:

```
<realization_body_context_section> ::=
   context
      [<realization_body_global_context_section>]
      [<realization_body_local_context_section>]
```

3.1.1. Global Context

Global context is just as it is for concepts and realization headers, with one addition:

```
<realization_body_global_context_section> ::=
   global context
      [<realization_body_global_context_item_sequence>]

<realization_body_global_context_item> ::=
   <concept_global_context_item> |
   realization <realization_id> for <concept_id>
```

The context of a realization body, unlike that of a concept, may include other realizations so complete facility declarations can be made locally, as explained in the next section.

3.1.2. Local Context

Local context also is similar to that for concepts, but again a bit more complex. Besides additional mathematical machinery, the local context section of a realization body may rename types and operations; and it may declare instances of concepts with specific realizations, local (unexported) operations, and/or module-level state information:

```
<realization_body_local_context_section> ::=
   local context
      [<realization_body_local_context_item_sequence>]
      [<realization_body_local_context_state_variables_section>]

<realization_body_local_context_item> ::=
   <math_subtype_declaration> |
   <math_operation_definition_declaration> |
   <concept_renaming_section> |
   <realization_body_facility_declaration> |
   <realization_body_local_operation_declaration>
```

The mathematical local context items are discussed elsewhere [Heym 94a], as is renaming [Part II]. The only difference here is that the renaming of types and operations in the local context section applies only within the realization body, and does not affect the interface.

One major difference between local context in conceptual modules and in realization bodies involves the declaration of facilities. In the conceptual world, where only abstract behavior matters, a facility is an abstract instance that is defined by fixing the parameters of a concept. In the realization world, you must *complete* such a facility declaration by selecting some realization for that concept and fixing the realization's parameters:

```
<realization_body_facility_declaration> ::=
   facility <facility_id> is <conceptual_module_id>
         [(<concept_facility_argument_list>)]
      realized by <realization_module_id>
         [(<realization_facility_argument_list>)]
```

In fully parameterized realizations, you do not declare facilities in the local context section because all facilities needed are declared by the client. For partial instantiations of fully parameterized realizations (and for some other realizations), you do need to make facility declarations as part of local context.

The second new kind of local context in realization bodies is the local operation, i.e., an operation that is not exported as part of the interface but that simplifies coding of the exported operations:

```
<realization_body_local_operation_declaration> ::=
   <operation_declaration>
      [<operation_body_context>]
      begin
         <statement_sequence>
      end <operation_id>

<operation_body_context> ::=
   context
      [<realization_body_facility_declaration_sequence>]
      [variables
         <variable_declaration_sequence>]

<variable_declaration> ::=
   <variable_id> : <type_id>
```

The context section of an operation may contain declarations of facilities and/or variables, whose scope is local to the operation and whose lifetime is the duration of the operation's execution. The examples only illustrate local variables, not local facilities.

Figure 4 shows a couple local operations. These and all other local operations should satisfy two conditions:

- A local operation has a formal behavioral specification.
- A local operation may not include code that involves variables of any type exported by the module (but it may, of course, involve variables of their representation types).

The reason for the first condition should be clear. The reason for the second is more subtle. Each exported type has what we call a representation "convention"; others call this a "representation invariant" or "class invariant". The convention for a type is required to hold only at the beginning and end of every primary operation for that type — not necessarily *during* execution of such an operation. The correctness of implementations of the primary operations is based on so-called "data type induction" in which the convention is shown to hold following initialization, and is shown to hold at the end of each primary operation given that it holds at the beginning. Now suppose a local operation were to try to do something to a variable of an exported type as though it were abstract. It would have to do so by invoking a primary operation since these are the only operations available for the exported type at this point. A primary operation is known to work correctly only if the convention holds when it is called. But the local operation can be called from within the body of a primary operation, and at the time of a call the convention might not hold for variables of an exported type that are parameters to the local operation. (There also are module-level conventions, discussed below, which further complicate matters.)

It is possible to build a modular proof system that handles internal calls to exported operations, thereby (apparently) giving an implementer greater "freedom". Our experience suggests, however, that an outright ban on such poor practices — even if they can be dealt with in principle — eliminates many hard-to-find bugs. This kind of syntactic check (which, as it turns out, is always easy to live with) is especially helpful in the absence of a mechanical verifier that embodies a sophisticated proof system.

The last kind of local context declaration involves module-level state variables. These are similar to conceptual state variables, except that they are programming and not mathematical objects:

```
<realization_body_local_context_state_variables_section> ::=
   state variables
        <realization_body_local_context_state_variable_sequence>
      [convention <assertion>]
      [correspondence <assertion>]

<realization_body_local_context_state_variable> ::=
   <variable_id> : <type_id>
```

Each instance of a realization module with state variables has its own set of them. These variables are accessible both to local and to exported operations but are, of course, not visible outside the module.

The **convention** and **correspondence** clauses are used to characterize the relationship between the realization module's state variables and the associated conceptual module's state variables. However, even if the concept has no state variables, the realization may still have some. A typical case involves shared storage among variables of an exported type [Ernst 94]. Conventions and correspondences for types, which are similar, are covered in Section 3.2.2.

The possibility of module-level state variables raises the issue of facility initialization, and (because facilities can be declared within operations, i.e., in dynamic scopes) facility finalization. Initialization of state variables to initial values of their types is automatic. If any additional processing is needed to get module-level state variables to a desired initial configuration, this code is supplied in an interface operation called "Initialize_Facility" which has no parameters. Similarly, in the relatively rare case that finalization requires extra processing before finalization of the state variables, it is done in an operation called "Finalize_Facility". A typical realization body contains neither of these operations.

3.2. Interface Section

The interface section of a realization body defines representations for exported types and code bodies for exported operations:

```
<realization_body_interface_section> ::=
   interface
      [re-exports
         <concept_re-exported_item_sequence>]
      [<realization_body_interface_item_sequence>]

<realization_body_interface_item> ::=
   <type_representation_declaration> |
   <realization_body_exported_operation_declaration>
```

3.2.1. Re-exported Interfaces

RESOLVE lets you *re-export* from a realization R the entire interface implementation of any facility that appears in R's global, parametric, or local context. Figure 5 illustrates the use of this feature in creating a partial instantiation.

As for concepts, you may rename any type or operation when re-exporting it, but the examples do not illustrate this. Renaming allows construction of very simple alternate "views" of concepts by making the by-name connection between types and operations in a concept and realization slightly more flexible. However, there is no shorthand method for changing any other part of an interface when re-exporting it. For example, suppose you have an instance of a concept C that exports part of the interface needed from realization R — except that the names of the operations are not identical. This problem can be solved by renaming. But if the order of operands for one of C's operations needs to be reversed, then renaming cannot be used. The only way to achieve the desired result is to write trivial wrapper operations in R, each of which calls through to the appropriate operation from C. The potential problems here can be minimized by adopting library-wide conventions, as we have done, but we are considering language extensions that would solve this minor problem as a side-effect of dealing with other limitations related to "common interface models" for components [Edwards 93, Edwards 94].

3.2.2. Types

The representation of each exported programming type is another program type that is in the context of the realization body:

```
<type_representation_declaration> ::=
   type <type_id> is represented by <type_id>
      [convention <assertion>]
      correspondence <assertion>
```

In Figure 4, for example, the type Partial_Map is represented as a Stack of Records whose fields are of type D_Item and R_Item. The representation type is built up, level by level, using facility declarations in the realization header and/or body context section.

It seems it would be convenient to have built-in type constructors (especially records), but this would introduce several complications, too. In particular, the uniformity seen in fully parameterized realizations such as in Figure 4 would be lost. And little would be gained, because the real economy of having built-in records comes in the use of compact record field extraction notation, not in record declaration and naming. So RESOLVE includes special syntax for extracting record fields (see Section 3.2.3), just as there is special syntax for expressions involving the special scalar types Boolean, Character, and Integer [Part II]. But the language contains no built-in array or pointer type constructors at all. These capabilities are provided by concepts that have no special status whatsoever among all reusable abstract components [Hollingsworth 92a, Hollingsworth 92b].

In order to reason about the correctness of ADT representation, you must define a relation between the (mathematical model of the) representation type and the mathematical model used in the exported type's specification. This relation is, in general, partial; i.e., certain configurations of the representation might never arise and, if they did, would have no abstract counterparts. The **`convention`** clause describes the domain of this relation, i.e., it characterizes the allowable representation states. The **`correspondence`** clause characterizes the relation between representation and conceptual values, given that the convention holds.

In Figure 4, for example, the convention says that no (D_Item, R_Item) pair occurs more than once in the Stack that represents a Partial_Map value. The correspondence says that the pairs in the set modeling a Partial_Map value are exactly those appearing in the Stack that represents it. A Stack is treated as a string in these assertions because that is its mathematical model.

When a variable of type T is declared in a client program, T's representation (say, of type T′) is automatically initialized. If an initial value of type T′ does not represent an initial value of type T according to the convention and correspondence, then the realization exporting type T must describe the additional processing that is required upon variable initialization. To do this, you write an operation called "Initialize" which has one parameter of type T and whose functional specification is the **`initialization`** clause from the concept's declaration of T. Similarly, if you need to do anything before the representation of a variable of type T is finalized, this must be done in an operation called "Finalize" which also has one parameter of type T.

3.2.3. Operations

The executable statements in an operation body are similar to the standard constructs of other imperative languages, with the exceptions briefly elaborated below:

```
<statement> ::=
   <swap> |
   <function_assignment> |
   <procedure_operation_call> |
   <selection> |
   <loop>

<swap> ::=
   <variable_id> :=: <variable_id>

<function_assignment> ::=
   <variable_id> := <function_operation_call>

<loop> ::=
   loop maintaining <assertion>
      <loop_statement>
   end loop
```

Both calls to procedural operations and selection statements are treated as in conventional imperative languages, and are not discussed further. The swap statement is rarely seen in other languages but should not be surprising in RESOLVE because the "swapping paradigm" is the basis for our designs [Harms 91].

The first significant difference here involves assignment statements, which are called "function assignment statements" [Harms 91] because of their restricted form. In RESOLVE, the right-hand side of an assignment must be an invocation of a functional procedure (or an expression involving the special scalar types, which is just shorthand for such an invocation). In most other languages, copying is assumed to be available for every type, and if it is not then the compiler simply copies the representation type. This approach is a disaster for modular reasoning [Harms 91]. So copying is not assumed to be available for all types in RESOLVE; swapping is. The special scalar types such as Integer are really no different. When you write "x := y" (you may for convenience) you really mean "x := Replica (y)". The compiler simply knows how to copy the standard representations of the special scalar types, and they can be copied inexpensively, so no performance or other problem arises from making this small concession to convenience at the apparent expense of uniformity.

The second noticeable difference is in loops, where every specific kind of loop (while loop, general exit loop, etc.) is enclosed within a **`loop`** block that includes a syntactic slot, the **`maintaining`** clause, for the loop invariant.

There is one other convenience feature that is not evident from the syntax at this level of description, but which is illustrated in the examples. For each possible number of fields N of a record (e.g., two), the RESOLVE library in principle includes a concept (e.g., Record2_Template) that exports a type (e.g., Record2) whose model is a tuple with N components. These concepts are declared as needed in global context and instantiated like other concepts.

For each component of a Record type (e.g., field1, whose type is **math**[Item1]) there is an associated operation that swaps it with a variable of the appropriate type, e.g.:

```
operation Swap_Field1 (
        alters     r: Record2
        alters     x: Item1
    )
  ensures    r.field1 = #x  and  x = #r.field1  and
             r.field2 = #r.field2
```

The pervasiveness of records in typical data representations only amplifies the annoyance of accessing each field by swapping it into a simple variable, modifying that variable, and then swapping back. There is no danger — from the standpoint of modular reasoning or otherwise — in using ordinary "dot" notation for record field extraction in executable statements, just as it is used in other programming languages and in RESOLVE assertions for tuple component extraction. So you may use r.f (for field f of record r) anywhere a variable might appear. It is straightforward to translate this mechanically into calls to Swap_Field operations as needed.

It might seem odd that we should comment on this at all. But unfortunately, when programming in the RESOLVE/Ada discipline without any support tools [Part IV], you have to use Swap_Field calls to access record fields! In the RESOLVE/C++ discipline, on the other hand, you can use dot notation if you are careful to follow the other aspects of the discipline.

4. Verification of Realizations

Writing a correct realization module is just part of the work involved in creating a certified concrete component. Before it can be confidently (re)used, a realization must be shown to conform to its specification. The potential for modular formal verification of correctness of realizations is a central objective of the RESOLVE framework, discipline, and language [Part I, Hollingsworth 92b, Weide 94a].

We emphasize that we do *not* intend — in the short term at least — that every RESOLVE realization must be formally verified before being placed in a component library, because mechanical discovery of formal proofs even for rather simple components is beyond current theorem proving technology. So a sound and rigorous informal argument, perhaps presented in a code review and augmented by systematic testing for added comfort, is a good temporary substitute for complete formality. However, we *do* intend that every realization in principle should be formally *verifiable* because the existence of a sound and rigorous informal argument is predicated on the existence of a formal proof of comparable length and complexity. In other words, if we can't mechanically discover a formal proof of correctness for a correct component, then the problem is not due to deficiencies in our components or in our programming style.

4.1. Modular Reasoning

Achieving modular proof-of-correctness (which is tantamount to modular reasoning about program behavior) means that:

(1) A programmer using an abstract component can reason about its functional behavior in a client program independently of the concrete components implementing that abstract component.

(2) A programmer developing a concrete component to implement an abstract component can reason about its functional correctness independently of the client programs using that concrete component.

For this approach to succeed, an abstract component must be considered to define a *contract* between implementer and client. It must state precisely all the responsibilities and guarantees of both parties. For example, the client programmer must make sure that every call to an operation occurs when that operation's precondition holds, and the implementer must make sure that every exported operation eventually returns in a state satisfying the postcondition of that operation. In RESOLVE as in other verification systems, such proof obligations are raised in the client program and the realization body.

The first problem to be addressed is that many programming practices involving operation calls — repeated arguments to calls, use of global variables as arguments to calls, aliased pointers and array references, and a variety of other more-or-less subtle problems — can thwart modularity [Hollingsworth 92b, Weide 93, Weide 94a]. Much of this territory has been mapped out in work involving Hoare-style systems, which have typically considered statement-level verification and modular verification across calls [Weide 93]. In RESOLVE, the offending practices are syntactically outlawed or simply cannot arise. (In Ada and C++, the discipline — not the compiler — "outlaws" them.)

Most of the RESOLVE work has focused on relatively uncharted areas of modular verification of parameterized components and component composition. Some of the most interesting barriers to modularity arise from these more advanced constructs. For example, in the RESOLVE framework a client instantiates a component before using it. This involves fixing conceptual parameters and fixing realization parameters. The claim that a concrete component implements an abstract component makes sense only if the client fixes these parameters in accordance with the restrictions stated in the parametric context sections of the concept and the realization header — which is why these restrictions are stated formally in RESOLVE. For example, in Figure 1 (one realization for a Top operation for Stacks) consider a client that passes for the formal "Replica" operation an operation which has the proper parameter profile, but which doesn't copy an Item. Then surely a proof that the code in the realization body correctly implements the Top operation is worthless — that's not at all what the code actually does in such a case. Instantiation, then, raises a proof obligation in the client program.

The key result of modular verification is simple and important: Once a concrete component is verified to be correct, it is guaranteed to be correct forever. If you put it in a component library and use it over and over again, the per-use cost of this proof (however expensive it is initially) becomes negligible. Contrast this with a proof system that requires reverification for each *use* of a component and the primary benefit of modular verification is clear.

We have developed a proof system that includes verification rules for all RESOLVE constructs. Details of various pieces of this system, which are beyond the scope of this paper, can be found in various places [Krone 88, Ernst 91, Ernst 94, Heym 94b].

4.2. Semantics

The denotational semantics for RESOLVE are based on a rich structured space that includes models of the same ideas as those in the proof system: mappings from identifiers to (generic) abstract and concrete components and their instances in order to record "declaration meanings" arising during compilation and instantiation; the usual variable-to-value mappings in order to denote the usual effects of execution; and a very important bit called the *assert status*. Proofs of soundness and (relative) completeness are conceptually straightforward but still distressingly intricate. This section provides an overview of the metatheory.

The foremost implication of doing modular verification in a RESOLVE-like framework is the requirement for a different definition of validity (correctness). It is not possible to conclude that a concrete component is valid even if an instance of it produces correct outputs for all executions. To see why, consider a concrete component R that uses an abstract component C. Suppose that there is an implementation of C where each operation satisfies its postconditions even when the corresponding preconditions are not met. If R uses this implementation of C and violates some of the preconditions, it may still be valid under a usual definition if it meets its specification. However, if an alternative implementation is used for C — one that behaves in an unspecified way when its operations' preconditions are violated — then R will not be valid. (Similarly, an invalid implementation of C may make R invalid.) In other words, the usual definition of validity depends on which implementation of C is used. Modular verification requires that validity be based only on the specifications of reused components.

The semantic space, therefore, includes an assert status which allows us to capture an appropriately modified notion of validity for declarations and operation execution. Initially the assert status is *neutral* (NL). It may become *categorically false* (CF) when, for example, a parameter to a facility instantiation does not satisfy the stated context restrictions, or when the precondition of a called operation is violated, or when the code for a local or exported operation does not satisfy its postcondition. It may become *vacuously true* (VT) when a called operation violates its postcondition. Once the assert status becomes CF or VT, it remains that way for the rest of a declaration and/or operation execution. A declaration or concrete operation is valid whenever the assert

status, starting from NL, does not end up CF; and a concrete component is valid whenever all its declarations and operations are valid [Krone 88, Ernst 94].

Operation implementation semantics are also different in RESOLVE, partly because operations may be specified to compute relations rather than functions (as with Undefine_Any_One in Partial_Map_Template [Part II]). In the absence of non-determinism arising from the language, it might seem that any particular implementation of an operation with a relational specification should actually compute a function that agrees with the given relation. But an operation's implementation might be layered on other operations with relational specifications [Ernst 94], and representations of certain data abstractions may involve relational correspondences. Furthermore, in specifying most optimization problems, choice is inherent in the problem statement — there are many possible correct answers for the same input [Weide 94c]. Another situation involves the non-determinism that potentially can be introduced by data abstraction itself [Ernst 94]. The semantics and modular proof rules must deal with these cases.

4.3. Soundness, Expressiveness, and Relative Completeness

Proving that a realization is correct is a syntax-directed process that effectively converts some program-like text — the realization, along with the specification it purports to implement and the specifications of concepts it uses — into a mathematical assertion. The idea is that if the assertion holds, then the program is valid; if not, the program still might execute correctly but you should not trust it to do so. (The latter case might arise, for example, if a loop invariant is too weak.)

Given these notions, establishing soundness and relative completeness of the proof system essentially involves showing that each rule used to carry out an incremental step in the program-to-assertion transformation preserves validity in both directions.

While soundness is an essential requirement of any proof system, relative completeness is also important. With respect to the execution-time aspect of performance, for example, it implies the ability to demonstrate tightest possible bounds. The RESOLVE proof system is only "relatively" complete because the specification language includes theories for which there is no complete deduction system. "Relative" means that the proof system itself does not introduce any incompleteness.

5. Conclusion

The RESOLVE realization language, at the level of executable statements, is not much different from other modern imperative languages. However, at the level of constructs needed for effective software component engineering, RESOLVE is unusual in many respects. The most significant are support for multiple implementations of a single concept, separate parameterization of realizations, separation of realization header from realization body, and syntactic slots for assertions needed in program proofs.

Parts of the RESOLVE proof system that handle shared realizations [Ernst 94] and preliminary work on extensions to handle performance verification in a modular fashion [Krone 93, Sitaraman 94] show the difficulties involved in dealing with modern programming language features in a modular way. This is ironic in a sense because these features were introduced largely to permit modular design and programming. In general they are almost too powerful, leaving the software engineer with too many choices and little guidance. It is little wonder, then, that most programs using these features do not satisfy all the technical conditions under which modular reasoning is possible. One of the key contributions of the formal methods portion of the RESOLVE work is the translation of technical results from formal methods into programming language syntax, principles, and guidelines that can be applied by practicing software engineers to improve their design and coding habits.