SOFTWARE ENGINEERING NOTES 19, 4 (OCT. 1994), 52-63.

Part IV: RESOLVE Components in Ada and C++

Joseph E. Hollingsworth
Sethu Sreerama
Bruce W. Weide
Sergey Zhupanov

By observing a careful discipline you can convert RESOLVE realizations to (or directly realize RESOLVE concepts in) Ada or C++, even though these languages do not mesh well with the RESOLVE realization language [Part III] in several places. In this paper we show some examples of how this works.

Ada and C++, while superficially quite different from each other in many respects, are about equally well-suited (perhaps we should say poorly-suited) substitutes for the RESOLVE realization language. Each language has a number of interesting features that we simply do not use, because we do not know how to modularly reason about code that uses these features and/or because we do not need to use them to imitate RESOLVE realizations. The RESOLVE/Ada and RESOLVE/C++ disciplines — strictly followed — are therefore conservative: If you follow our advice then you should be able to reason modularly about your programs; if not, then you still might be able to reason modularly if you know what you are doing and why, and if you understand the pitfalls that can disturb modularity and avoid them even while violating the discipline.

Given space constraints it is not possible to present the precise principles and conventions of either discipline here. However, the RESOLVE/Ada discipline is readily available [Hollingsworth 92b]; its counterpart for C++ is currently under development. As a surrogate, in this paper we show some Ada and C++ code that you might write for several of the examples in the companion papers [Parts II and III], to give the flavor of what is involved.

1. Ada Examples

An Ada generic package specification corresponds to a RESOLVE realization header plus the concept it implements (including both conceptual and realization parameters), and an Ada package body corresponds to a RESOLVE realization body. There is, then, one Ada generic package per realization, not one per concept as might be hoped. The reason is that the Ada component model is the traditional "one implementation per specification", which does not match the more flexible RESOLVE model [Part I].

Naming of Ada generic packages using both concept and realization names is a nuisance. The concept name is generally more important than the realization name, so by

convention we use the concept name followed by some short designator of the realization (not the entire realization name).

Figures 1-5 illustrate how various RESOLVE examples [Parts II and III] look when coded in Ada following the RESOLVE/Ada discipline. One things stands out immediately: the large volume of code, especially when compared to the RESOLVE and C++ counterparts. The two main reasons for this are that Ada currently does not have a method for passing a package instance as a generic parameter, and that it does not support automatic initialization and finalization of variables. Apparently Ada9X will fix both problems, but we have not yet investigated the impact of these changes on the details of the RESOLVE/Ada discipline.

Some specific areas where Ada code written under the RESOLVE/Ada discipline is a bit unusual are:

- *Swap procedure* — There is no ":=:" operator in Ada, so there is a Swap procedure for every type. (We use a tool called *rapp*, for RESOLVE/Ada preprocessor, that allows us to write ":=:" as though it were an operator. Rapp replaces each ":=:" by the appropriate Swap call. The Ada code in the examples does not use this or any other feature of rapp.)

- *Initialize and Finalize procedures* — There is no automatic variable initialization and finalization in Ada. So for every variable you declare, there are calls to Initialize and Finalize at the beginning and end of that variable's scope. Similarly, there are calls to Initialize_Package and Finalize_Package (the Ada counterparts of Initialize_Facility and Finalize_Facility) at the beginning and end of every package instance's scope. (Rapp automatically inserts all initialization and finalization calls for variables and package instances at the appropriate places.)

- *Scalar types* — Relatively seamless integration with the built-in scalar types is provided through library packages called Standard_Boolean_Facility, etc. These packages permit a uniform treatment of built-in scalar types by exporting the Initialize, Finalize, and Swap procedures that are needed when composing the built-in scalar types with other ADTs.

- *Record types* — With one exception (see the private part in Figures 3-5), Ada's built-in records are not used. RecordN_Template generic packages with Swap_Field operations [Part III] are used instead. (Rapp converts dot notation for field extraction into appropriate Swap_Field calls.)

- *Limited private types* — All types imported and exported by Ada generic packages are limited private, so the compiler can enforce RESOLVE's no-built-in-copying and no-built-in-equality-testing restriction.

- *Functions and procedures* — Ada has an unfortunate rule that requires all parameters to functions to have mode `in`, and this precludes many layered

implementations that are coded in the RESOLVE style. So all RESOLVE operations are implemented as Ada procedures, never as functions. For a RESOLVE functional operation an extra parameter is added to convey the return value. The naming convention is as follows: for RESOLVE procedural operation "P", the Ada procedure is "P"; for RESOLVE functional operation "F" returning type Boolean, the Ada procedure is "Test_If_F"; for RESOLVE functional operation "F" returning any other type, the Ada procedure is "Get_F".

- *Parameter modes* — All Ada parameters have mode **`in out`**, except for the counterparts of RESOLVE **`preserves`**-mode parameters of the built-in scalar types, which have mode **`in`**.

- *Type parameters to generic packages* — For every type that is imported/exported by an Ada generic package, the corresponding Initialize, Finalize, and Swap procedures are also imported/exported immediately after the type. (The use of box notation "<>" in declaring the formal parameters makes instantiating a generic package easy, since no actuals need to be passed explicitly for these procedures; see the private part in Figure 5.)

- *Facility parameters to generic packages* — For every RESOLVE facility parameter, the entire Ada interface is imported *except* Initialize_Package and Finalize_Package. Each type and procedure being imported is listed separately and in the same order as listed in the generic package that exports them. (The Ada type name is sometimes slightly different from the corresponding RESOLVE type in order to avoid conflicts with other imported or exported types; see Figure 3 for an example. Also note that box notation helps here even more than it does with RESOLVE type parameters; see the private part in Figure 5.)

- *Multiple implementations* — The exports of an Ada package specification are determined entirely from the RESOLVE concept interface, and never depend on realization-related information. The only differences between two Ada generic package specifications for the same concept are in the generic package names, in the context declarations (with/use clauses), and in the generic parameter lists where realization parameters occur.

Figure 1: Ada Generic Package Specification for a Realization of Stack_Template

```
with Standard_Integer_Facility; use Standard_Integer_Facility;

generic

   type Item is limited private;       -- type Item
   with procedure Initialize (
        x: in out Item
     ) is <>;
   with procedure Finalize (
        x: in out Item
     ) is <>;
   with procedure Swap (
        x1: in out Item;
        x2: in out Item
     ) is <>;

package Stack_Template_1 is

   procedure Initialize_Package;
   procedure Finalize_Package;

   type Stack is limited private;
   procedure Initialize (
        s: in out Stack
     );
   procedure Finalize (
        s: in out Stack
     );
   procedure Swap (
        s1: in out Stack;
        s2: in out Stack
     );

   procedure Push (
        s: in out Stack;
        x: in out Item
     );
   procedure Pop (
        s: in out Stack;
        x: in out Item
     );
   procedure Get_Length (
        s: in out Stack;
        result: in out Integer
     );

private

   -- Representation goes here

end Stack_Template_1;
```

Figure 2: Ada Generic Package for Realization "Obvious" for Stack_Top_Capability

```
generic

   type Item is limited private;        -- type Item
   with procedure Initialize (
        x: in out Item
     ) is <>;
   with procedure Finalize (
        x: in out Item
     ) is <>;
   with procedure Swap (
        x1: in out Item;
        x2: in out Item
     ) is <>;

   type Stack is limited private;       -- facility Stack_Facility
   with procedure Initialize (
        s: in out Stack
     ) is <>;
   with procedure Finalize (
        s: in out Stack
     ) is <>;
   with procedure Swap (
        s1: in out Stack;
        s2: in out Stack
     ) is <>;
   with procedure Push (
        s: in out Stack;
        x: in out Item
     ) is <>;
   with procedure Pop (
        s: in out Stack;
        x: in out Item
     ) is <>;
   with procedure Get_Length (
        s: in out Stack;
        result: in out Integer
     ) is <>;

   with procedure Get_Replica (         -- operation Replica
        x: in out Item;
        result: in out Item
     ) is <>;

package Stack_Top_Capability_1 is

   procedure Initialize_Package;
   procedure Finalize_Package;
```

```
   procedure Get_Top (
        s: in out Stack;
        x: in out Item
     );

end Stack_Top_Capability_1;

-------------------------------------------------------------------------

package body Stack_Top_Capability_1 is

   procedure Initialize_Package is
   begin
      null;
   end Initialize_Package;

   procedure Finalize_Package is
   begin
      null;
   end Finalize_Package;

   procedure Get_Top (
        s: in out Stack;
        x: in out Item
     ) is
        x_copy: Item;
   begin
        Initialize (x_copy);
     Pop (s, x);
     Get_Replica (x, x_copy);
     Push (s, x_copy);
        Finalize (x_copy);
   end Get_Top;

end Stack_Top_Capability_1;
```

Figure 3: Ada Generic Package for Realization "Bundled" for Replicable_Stack_Template

```
with Standard_Integer_Facility; use Standard_Integer_Facility;

generic

   type Item is limited private;       -- type Item
   with procedure Initialize (
        x: in out Item
     ) is <>;
   with procedure Finalize (
        x: in out Item
     ) is <>;
   with procedure Swap (
        x1: in out Item;
        x2: in out Item
     ) is <>;

   type Stack1 is limited private;     -- facility Stack_Facility
   with procedure Initialize (
        s: in out Stack1
     ) is <>;
   with procedure Finalize (
        s: in out Stack1
     ) is <>;
   with procedure Swap (
        s1: in out Stack1;
        s2: in out Stack1
     ) is <>;
   with procedure Push (
        s: in out Stack1;
        x: in out Item
     ) is <>;
   -- Similarly for Pop, Get_Length

   with procedure Get_Replica (     -- facility
        x: in out Stack1;           --   Stack_Replica_Facility
        result: in out Stack1
     ) is <>;

package Replicable_Stack_Template_1 is

   procedure Initialize_Package;
   procedure Finalize_Package;

   type Stack is limited private;
   procedure Initialize (
        s: in out Stack
     );
   procedure Finalize (
        s: in out Stack
     );
```

```
   procedure Swap (
        s1: in out Stack;
        s2: in out Stack
     );

   procedure Push (
        s: in out Stack;
        x: in out Item
     );
   -- Similarly for Pop, Get_Length, Get_Replica

private

   type Stack is
     record
        rep: Stack1;
     end record;

end Replicable_Stack_Template_1;

-------------------------------------------------------------------------

package body Replicable_Stack_Template_1 is

   procedure Initialize_Package is
   begin
     null;
   end Initialize_Package;

   procedure Finalize_Package is
   begin
     null;
   end Finalize_Package;

   procedure Initialize (
        s: in out Stack
     ) is
   begin
     Initialize (s.rep);
   end Initialize;

   procedure Finalize (
        s: in out Stack
     ) is
   begin
     Finalize (s.rep);
   end Finalize;

   procedure Swap (
        s1: in out Stack;
        s2: in out Stack
     ) is
   begin
     Swap (s1.rep, s2.rep);
   end Swap;
```

```
   procedure Push (
        s: in out Stack;
        x: in out Item
     ) is
   begin
      Push (s.rep, x);
   end Push;

   -- Similarly for Pop, Get_Length, Get_Replica

end Replicable_Stack_Template_1;
```

Figure 4: Ada Generic Package for Realization "Unordered_Stack" for Partial_Map_Template

```
with Standard_Boolean_Facility; use Standard_Boolean_Facility;
with Standard_Integer_Facility; use Standard_Integer_Facility;

generic

   type D_Item is limited private;      -- type D_Item
   with procedure Initialize (
        x: in out D_Item
     ) is <>;
   with procedure Finalize (
        x: in out D_Item
     ) is <>;
   with procedure Swap (
        x1: in out D_Item;
        x2: in out D_Item
     ) is <>;

   type R_Item is limited private;      -- type R_Item
   with procedure Initialize (
        x: in out R_Item
     ) is <>;
   with procedure Finalize (
        x: in out R_Item
     ) is <>;
   with procedure Swap (
        x1: in out R_Item;
        x2: in out R_Item
     ) is <>;

   type Record2 is limited private;     -- facility D_R_Pair_Facility
   with procedure Initialize (
        r: in out Record2
     ) is <>;
   with procedure Finalize (
        r: in out Record2
     ) is <>;
```

```
   with procedure Swap (
        r1: in out Record2;
        r2: in out Record2
     ) is <>;
   with procedure Swap_Field1 (
        r: in out Record2;
        x: in out D_Item
     ) is <>;
   with procedure Swap_Field2 (
        r: in out Record2;
        x: in out R_Item
     ) is <>;

   type Stack is limited private;      -- facility Stack_Facility
   with procedure Initialize (
        s: in out Stack
     ) is <>;
   with procedure Finalize (
        s: in out Stack
     ) is <>;
   with procedure Swap (
        s1: in out Stack;
        s2: in out Stack
     ) is <>;
   with procedure Push (
        s: in out Stack;
        x: in out Record2
     ) is <>;
   -- Similarly for Pop, Get_Length

   with procedure Test_If_Are_Equal (  -- operation Are_Equal
        d1: in out D_Item;
        d2: in out D_Item;
        result: in out Boolean
     ) is <>;

package Partial_Map_Template_1 is

   procedure Initialize_Package;
   procedure Finalize_Package;

   type Partial_Map is limited private;
   procedure Initialize (
        m: in out Partial_Map
     );
   procedure Finalize (
        m: in out Partial_Map
     );
   procedure Swap (
        m1: in out Partial_Map;
        m2: in out Partial_Map
     );
```

```
   procedure Define (
        m: in out Partial_Map;
        d: in out D_Item;
        r: in out R_Item
      );
   -- Similarly for Undefine, Undefine_Any_One, Test_If_Is_Defined,
   --   Get_Size

private

   type Partial_Map is
      record
         rep: Stack;
      end record;

end Partial_Map_Template_1;

-------------------------------------------------------------------------

package body Partial_Map_Template_1 is

   type D_R_Pair is new Record2;

   procedure Swap_Domain_Value (
        p: in out D_R_Pair;
        d: in out D_Item
      ) renames Swap_Field1;

   procedure Swap_Range_Value (
        p: in out D_R_Pair;
        r: in out R_Item
      ) renames Swap_Field2;

   procedure Pop_Until_Passed (
        s1: in out Stack;
        s2: in out Stack;
        d: in out D_Item
      ) is
   begin
      -- Code for Pop_Until_Passed
   end Pop_Until_Passed;

   procedure Combine (
        s1: in out Stack;
        s2: in out Stack
      ) is
   begin
      -- Code for Combine body
   end Combine;

   procedure Initialize_Package is
   begin
      null;
   end Initialize_Package;
```

```
   procedure Undefine (
        m: in out Partial_Map;
        d: in out D_Item;
        d_copy: in out D_Item;
        r: in out R_Item
      ) is
        catalyst: Stack;
        p: D_R_Pair;
   begin
        Initialize (catalyst);
        Initialize (p);
      Pop_Until_Passed (m.rep, catalyst, d);
      Pop (catalyst, p);
      Swap_Domain_Value (p, d_copy);
      Swap_Range_Value (p, r);
      Combine (m.rep, catalyst);
        Finalize (p);
        Finalize (catalyst);
   end Undefine;

   -- Similarly for Finalize_Package, Initialize, Finalize, Swap,
   --   Define, Undefine_Any_One, Test_If_Is_Defined, and Get_Size

end Partial_Map_Template_1;
```

Figure 5: Ada Generic Package for Realization "Unordered_Stack_Fixed" for Partial_Map_Template

```
with Standard_Boolean_Facility; use Standard_Boolean_Facility;
with Standard_Integer_Facility; use Standard_Integer_Facility;
with Record2_Template;
with Stack_Template_1;
with Partial_Map_Template_1;

generic

   type D_Item is limited private;     -- type D_Item
   with procedure Initialize (
        x: in out D_Item
      ) is <>;
   with procedure Finalize (
        x: in out D_Item
      ) is <>;
   with procedure Swap (
        x1: in out D_Item;
        x2: in out D_Item
      ) is <>;

   type R_Item is limited private;     -- type R_Item
```

```
   with procedure Initialize (
        x: in out R_Item
     ) is <>;
   with procedure Finalize (
        x: in out R_Item
     ) is <>;
   with procedure Swap (
        x1: in out R_Item;
        x2: in out R_Item
     ) is <>;

   with procedure Test_If_Are_Equal (  -- operation Are_Equal
        d1: in out D_Item;
        d2: in out D_Item;
        result: in out Boolean
     ) is <>;

package Partial_Map_Template_2 is

   procedure Initialize_Package;
   procedure Finalize_Package;

   type Partial_Map is limited private;
   procedure Initialize (
        m: in out Partial_Map
     );
   procedure Finalize (
        m: in out Partial_Map
     );
   procedure Swap (
        m1: in out Partial_Map;
        m2: in out Partial_Map
     );

   procedure Define (
        m: in out Partial_Map;
        d: in out D_Item;
        r: in out R_Item
     );
   -- Similarly for Undefine, Undefine_Any_One, Test_If_Is_Defined,
   --   Get_Size

private

   package D_R_Pair_Facility is new
      Record2_Template (
           Item1 => D_Item,
           Item2 => R_Item
        );
   use D_R_Pair_Facility;

   package Stack_Facility is new
      Stack_Template_1 (
           Item => Record2
        );
   use Stack_Facility;
```

```
   package Partial_Map_Facility is new
      Partial_Map_Template_1 (
           D_Item => D_Item,
           R_Item => R_Item,
           Record2 => Record2,
           Stack => Stack
         );

   type Partial_Map is
      record
         rep: Partial_Map_Facility.Partial_Map;
      end record;

end Partial_Map_Template_2;

-------------------------------------------------------------------

package body Partial_Map_Template_2 is

   procedure Initialize_Package is
   begin
      D_R_Pair_Facility.Initialize_Package;
      Stack_Facility.Initialize_Package;
      Partial_Map_Facility.Initialize_Package;
   end Initialize_Package;

   procedure Finalize_Package is
   begin
      Partial_Map_Facility.Finalize_Package;
      Stack_Facility.Finalize_Package;
      D_R_Pair_Facility.Finalize_Package;
   end Finalize_Package;

   procedure Initialize (
        m: in out Partial_Map
      ) is
   begin
      Partial_Map_Facility.Initialize (m.rep);
   end Initialize;

   procedure Undefine (
        m: in out Partial_Map;
        d: in out D_Item;
        d_copy: in out D_Item;
        r: in out R_Item
      ) is
   begin
      Partial_Map_Facility.Undefine (m.rep, d, d_copy, r);
   end Undefine;

   -- Similarly for Finalize, Swap, Define, Undefine_Any_One,
   --   Test_If_Is_Defined, and Get_Size

end Partial_Map_Template_2;
```

2. C++ Principles

The RESOLVE/C++ discipline illustrated in this section is newer and less-tested than the RESOLVE/Ada discipline, in no small part because of the unavailability (until recently) of compilers that properly handle "templates" — the C++ mechanism for generic parameters. We have explored several variants of the particular discipline illustrated here, in which we use the traditional imperative language style for operation invocation (e.g., Push (s, x)) as opposed to the pseudo-monadic object-oriented programming style (e.g., s.Push (x)). We hope that those who expect C++ code to use the latter style are not blinded to the benefits of the discipline by this apparent *faux pas*. The variants that use more typical C++ style are alive and well and still under investigation.

A C++ class template corresponds to a RESOLVE realization header plus the concept it implements (including both conceptual and realization parameters), and the code for the class methods corresponds to a RESOLVE realization body. The reason for this is that unless you use inheritance in the "abstract base class" style (we do not in the discipline illustrated here) the C++ component model is the same as Ada's traditional "one implementation per specification".

There are two primary sources of difficulty in developing a RESOLVE/C++ discipline. The first problem — which fortunately is fairly rare — deals with concepts that export multiple types. We handle this using the "friend" construct, but none of the examples here illustrates this. The second problem is that, in effect, an ordinary C++ template class (a class obtained by instantiating a C++ class template) defines both a facility *and* a type. So we adopt conventions that help maintain the distinction between facility and type where possible.

First, we define some pseudo-keywords: FACILITY_AND_TYPE (for "typedef"), FACILITY (for "typedef"), and OBJECT (for " "). These help to clarify intent when making declarations. Second, we divide concepts into those that define a type and those that define only one or more operations. Instances of the former are declared using FACILITY_AND_TYPE , and instances of the latter using FACILITY. Public routines are declared "static" in either case, so operations are invoked with the traditional imperative style but with a fully-qualified name, e.g., "Facility::Operation (arguments)". The example code illustrates this in several places. Proposed "parameterized namespaces" will help here when they eventually appear in C++.

Some areas where C++ code under the RESOLVE/C++ discipline is a bit unusual are:

- *Swap operator* — There is no ":=:" operator in C++, so we use the slightly suggestive "&=" operator, which has no other use in RESOLVE/C++ code.

- *Initialize and Finalize procedures* — Initialize and Finalize are done using C++ constructors and destructors. Initialize_Facility and Finalize_Facility cause some problems, however. Because all similarly-declared instances of the same class template are considered to be the same instance in C++, there is no single "right"

place to call Initialize_Facility and Finalize_Facility. So in realizations where these operations actually need to do something, they are handled in the constructor and destructor by using a reference count to determine when the first object of a type is constructed and when the last object of a type is destroyed. This slightly tricky business is not illustrated in the examples.

- *Scalar types* — Seamless integration with the built-in scalar types is provided through "wrapper" classes called Standard_Boolean, etc. These classes (not class templates) permit a uniform treatment of the special scalar types by defining the "&=" operators that are needed when composing these types with other ADTs, and by defining other appropriate operations and type conversions between the wrapper classes and their built-in counterparts. A nice side-benefit of this approach is that the special scalar types have the same names as in RESOLVE; e.g., you use Integer instead of int.

- *Record types* — When a record type is needed, the obvious C++ struct is declared with public data members for each field, and a public "&=" operator that is implemented in the class declaration; see Figure 10. We declare a struct only to emphasize that the intent is to create a record; declaring the same thing as a class would be equivalent.

- *Prohibited copying* — Assignment and copy construction are explicitly prohibited by declaring them to be private. The equality-testing operator "==" need not be prohibited because the C++ compiler does not generate default equality-testing based on the representation.

- *Routines* — Most RESOLVE operations are implemented as C++ void routines. For a RESOLVE functional operation that returns anything but a special scalar type, an extra parameter is added to convey the return value from the corresponding void routine. The naming convention is as follows: for RESOLVE procedural operation "P", the C++ routine is "void P"; for RESOLVE functional operation "F" returning a special scalar type "scalar", the C++ routine is "scalar F"; for RESOLVE functional operation "F" returning any other type, the C++ routine is "void Get_F".

- *Parameter passing* — All C++ formal parameters are declared to be passed by reference (using "&"), except for the counterparts of RESOLVE **`preserves`**-mode parameters of the built-in scalar types, which are declared to be passed by value.

- *Types and facilities as template parameters* — Both RESOLVE type and facility parameters are imported as class parameters to class templates.

- *Multiple implementations* — The exports of a C++ class template are determined entirely from the RESOLVE concept interface, and never depend on realization-related information. The only differences between two C++ class templates for the same concept are in the class template names, in the context declarations

("include" files), and in the template parameter lists where realization parameters occur.

Figures 6-10 illustrate how some RESOLVE examples [Parts II and III] look when coded in C++ following the RESOLVE/C++ discipline. One thing that stands out here is that inheritance is not employed for the usual purpose, i.e., subtyping. In Figure 8 we inherit from template parameters to do a realization by bundling. But overall, templates are much more important and useful to us than inheritance.

Another more subtle consequence of the RESOLVE/C++ discipline is that some errors caught by Ada at compile time (because facility parameters to generic package specifications are "expanded" into their exports) are not caught by the C++ compiler, but must be caught by the linker. For example, in Figure 7 the C++ compiler has no way to know the types of the parameters to Push, Pop, and Get_Replica.

Finally, note that the code for routines such as Get_Top in Figure 7 actually resides in a header file for which object code is not generated until an instance is declared. Most compilers try to keep compiled class templates in an intermediate form to speed this process, but the fundamental problem remains. To offset it, you can create wrapper classes (not templates) that are implemented by instantiating class templates as needed by the application, then compiling them and placing their object code in the library.

Figure 6: C++ Class Template for a Realization of Stack_Template

```
#include "Standard_Integer.h"

template <
      class Item
   >
class Stack_1
{
   public:

      Stack_1 ();
      ~Stack_1 ();
      void operator &= (
            Stack_1& rhs
         );

      static void Push (
            Stack_1& s,
            Item& x
         );
      static void Pop (
            Stack_1& s,
            Item& x
         );
      static Integer Length (
            Stack_1& s
         );

   private:

      Stack_1 (Stack_1 s);
      Stack_1& operator = (const Stack_1& rhs);

      /* Representation goes here */
};
```

Figure 7: C++ Class Template for Realization Obvious for Stack_Top_Capability

```
template <
     class Item,
     class Stack,
     class Item_Replica_Facility
   >
class Stack_Top_1
{
   public:

      static void Get_Top (
           Stack& s,
           Item& x
         );
};

////////////////////////////////////////////////////////////////////////

template <
     class Item,
     class Stack,
     class Item_Replica_Facility
   >
void Stack_Top_1<Item, Stack, Item_Replica_Facility>::
Get_Top (
     Stack& s,
     Item& x
   )
{
   OBJECT Item x_copy;

   Stack::Pop (s, x);
   Item_Replica_Facility::Get_Replica (x, x_copy);
   Stack::Push (s, x_copy);
}
```

Figure 8: C++ Class Template for Realization "Bundled" for Replicable_Stack_Template

```
template <
     class Item,
     class Stack,
     class Stack_Replica_Facility
   >
class Replicable_Stack_1:
   virtual public Stack,
   virtual public Stack_Replica_Facility
{ /* Empty */ };
```

Figure 9: C++ Class Template (and One Routine Body) for Realization "Unordered_Stack" for Partial_Map_Template

```
#include "Standard_Boolean.h"
#include "Standard_Integer.h"

template <
     class D_Item,
     class R_Item,
     class D_R_Pair,
     class Stack,
     class D_Item_Are_Equal_Facility
   >
class Partial_Map_1
{
   public:

      Partial_Map_1 ();
      ~Partial_Map_1 ();
      void operator &= (
           Partial_Map_1& rhs
        );

      static void Define (
           Partial_Map_1& m,
           D_Item& d,
           R_Item& r
        );
      static void Undefine (
           Partial_Map_1& m,
           D_Item& d,
           D_Item& d_copy,
           R_Item& r
        );
      static void Undefine_Any_One (
           Partial_Map_1& m,
           D_Item& d,
           R_Item& r
        );
      static Boolean Is_Defined (
           Partial_Map_1& m,
           D_Item& d
        );
      static Integer Size (
           Partial_Map_1& m,
        );

   private:

      Partial_Map_1 (Partial_Map_1 s);
      Partial_Map_1& operator = (const Partial_Map_1& rhs);
```

```
     static void Pop_Until_Passed (
          Stack& s1,
          Stack& s2,
          D_Item& d
        );
     static void Combine (
          Stack& s1,
          Stack& s2
        );

     OBJECT Stack rep;
};

////////////////////////////////////////////////////////////////////////

template <
     class D_Item,
     class R_Item,
     class D_R_Pair,
     class Stack,
     class D_Item_Are_Equal_Facility
   >
void Partial_Map_1<D_Item, R_Item, D_R_Pair, Stack,
   D_Item_Are_Equal_Facility>::
Undefine (
     Partial_Map_1& m,
     D_Item& d,
     D_Item& d_copy,
     R_Item& r
   )
{
   OBJECT Stack catalyst;
   OBJECT D_R_Pair p;

   Pop_Until_Passed (m.rep, catalyst, d);
   Stack::Pop (catalyst, p);
   Combine (m.rep, catalyst);
   d_copy &= p.d;
   r &= p.r;
}

// Similarly for Pop_Until_Passed, Combine, constructor, destructor,
//   operation &=, Undefine_Any_One, Is_Defined, and Size
```

Figure 10: C++ Class Template (and One Routine Body) for Realization "Unordered_Stack_Fixed" for Partial_Map_Template

```
#include "Standard_Boolean.h"
#include "Standard_Integer.h"
#include "Stack_1.h"
#include "Partial_Map_1.h"

template <
     class D_Item,
     class R_Item,
     class D_Item_Are_Equal_Facility
   >
class Partial_Map_2
{
   public:

      Partial_Map_2 ();
      ~Partial_Map_2 ();
      void operator &= (
           Partial_Map_2& rhs
        );

      static void Define (
           Partial_Map_2& m,
           D_Item& d,
           R_Item& r
        );
      static void Undefine (
           Partial_Map_2& m,
           D_Item& d,
           D_Item& d_copy,
           R_Item& r
        );
      static void Undefine_Any_One (
           Partial_Map_2& m,
           D_Item& d,
           R_Item& r
        );
      static Boolean Is_Defined (
           Partial_Map_2& m,
           D_Item& d
        );
      static Integer Size (
           Partial_Map_2& m,
        );

   private:

      Partial_Map_2 (Partial_Map_2 s);
      Partial_Map_2& operator = (const Partial_Map_2& rhs);
```

```
     struct D_R_Pair
     {
        D_Item d;
        R_Item r;
        void operator &= (D_R_Pair& lhs, D_R_Pair& rhs)
        {
           lhs.d &= rhs.d;
           lhs.r &= rhs.r;
        }
     };

     FACILITY_AND_TYPE Stack_1<D_R_Pair> Stack;
     FACILITY_AND_TYPE Partial_Map_1 <D_Item, R_Item, D_R_Pair,
        Stack, D_Item_Are_Equal_Facility> Partial_Map;

     OBJECT Partial_Map rep;
};

////////////////////////////////////////////////////////////////////

template <
     class D_Item,
     class R_Item,
     class D_Item_Are_Equal_Facility
   >
void Partial_Map_2<D_Item, R_Item, D_Item_Are_Equal_Facility>::
Undefine (
     Partial_Map_2& m,
     D_Item& d,
     D_Item& d_copy,
     R_Item& r
   )
{
   Partial_Map::Undefine (m.rep, d, d_copy, r);
}

// Similarly for constructor, destructor, operation &=, Define,
//   Undefine_Any_One, Is_Defined, and Size
```

3. Conclusion

There is little difference for a RESOLVE programmer between the RESOLVE/Ada and the RESOLVE/C++ disciplines. Both are relatively cumbersome compared to programming in the RESOLVE realization language, and both disciplines probably seem awkward to typical Ada and C++ programmers as well. But after a short time it is easy to feel somewhat comfortable using either one, especially with support from tools such as rapp.

We plan to extend these tools in several ways. One direction that seems attractive is to generate Ada and/or C++ code from RESOLVE code. This approach permits us to

reason about program correctness in RESOLVE itself, factoring off the problem of correct code generation.

We have used the (manual) RESOLVE/Ada discipline to develop a significant component library and some classroom-complexity applications. Major parts of the discipline have been taught in CS2 at three different institutions, in three upper-division undergraduate/graduate courses at two different institutions, and in two TRI-Ada full-day tutorials. From this experience we are confident that the discipline is sound and that it can be used to develop non-trivial Ada components and applications that use them. The impact of Ada9X additions and changes remains to be investigated, as do various performance-related questions such as the effects of inlining in conjunction with heavy use of generics.

We have used one version of the RESOLVE/C++ discipline to develop a less extensive but still quite practical component library, and this also has been used to develop classroom-level applications. This version is being used to build a commercial software package for Windows — a realistic task that is more ambitious than the ones for which we have used the RESOLVE/Ada discipline. We hope to be able to report on the results of this effort in the future. The RESOLVE/C++ discipline has not been well-tested in the classroom yet, nor have the alternative versions been fully explored. So our future plans include investigating the relative advantages of these variants, teaching one or more of them, and exploring the same kinds of performance-related questions as for Ada.

The power and flexibility of Ada and C++ language constructs — just the positive attributes that let us shoehorn most of RESOLVE into these languages — must be harnessed in order to achieve modular reasoning about program behavior. Apparently, programming in what seems like a straight jacket is the price you need to pay for the invaluable ability to reason modularly about software components written in these languages. At least, we know of no published alternative styles that accomplish this.