A Generative Approach to Reusing of Ada Subsystems

James Solderitsch, Timothy Schreyer
Unisys, Electronic and Information Systems Group
Center for Advanced Information Technology
PO Box 517
Paoli, PA 19301-0517
215-648-2831, 215-648-2475
jjs@prc.unisys.com, schrey@prc.unisys.com

Abstract

Software reuse occurs on many levels, such as reuse of simple abstract data types, reuse of application subsystems, and the generation of system components and interfaces. Three prototypes developed at the Center for Advanced Information Technology — a black-box Ada software testing tool, an Intelligent Librarian for Ada reuse libraries, and a knowledge-based Tool Utilization Assistant in the document-processing domain — all successfully reuse the same underlying knowledge representation subsystems written in Ada. These subsystems are AdaTAU, a distributed rule base inference engine; AdaKNET, a structured inheritance network tool; and a hybrid knowledge representation scheme used to integrate them. This paper will discuss issues of reuse at the subsystem level and will address the role of the Ada language and generated language-based interfaces in making subsystem level reuse easier and more practical.

1. Introduction

This paper describes some experiences in the reuse of relatively large Ada subsystems in three different applications. Our approach has been to try and overcome some of the problems inherent in *Reuse in the Large* (*RitL*) compared to *Reuse in the Small* (*RitS*). In fact, the Ada language was designed in part to explicitly address the latter and the use of Ada for this purpose has been the subject of many papers and presentations (e.g. [Berrett87]).

Ada has not solved the reuse problem however well it has aided *RitS*. We begin by discussing several approaches to *RitL* ranging from a completely constructive approach to a completely generative one. The main body of this paper presents a partially generative approach based on the use of Interface Specification Languages (ISLs). We discuss the design and use of such languages, their development, and the philosophy and methods used in our approach. The paper closes with a lessons-learned section and a summary.

2. Approaches to Reuse

Ted Biggerstaff and Charles Richter in a paper that originally appeared in 1987 [Biggerstaff87] (and reprinted in[Biggerstaff89]) used several illustrations that nicely capture some of the basic issues and approaches fundamental to the subject of Software Reuse. Adaptations of two of these illustrations appear as figures 1 and 2 below. Although the figures themselves do not explicitly reference the distinction between the *RitS* and *RitL* categories, the methods and approaches shown in the figures naturally tend to one or the other of these two reuse categories.

From an Ada perspective, *RitS* is package level reuse. Math packages and simple data structures such as stacks and queues are elementary examples of this kind of reuse. Applications typically use such packages as a whole, and while individual applications may need to tailor parts of packages that are reused, the most economical form of reuse occurs when the package can be reused as-is. For this kind of as-is reuse to occur, the packages must be designed carefully with reuse in mind. If proper design principles are used, packages can be reused widely, far outside the scope of the application wherein the packages were originally developed.

Through the use of the package concept, Ada supports *RitS* but does not assure it. The incorporation of packages by an importing application illustrates the constructive approach to reuse shown in the left side of figure 1. *RitL* occurs when entire collections of modules or packages are used out of the context of the application(s) for which the collection was originally developed. There are circumstances where large systems can be composed from such smaller subsystems, but only when there is nearly an exact fit between the services provided by the collection and the needs of the

Features	*Approaches to Reusability*				
Component Reused	Building Blocks		Patterns		
Nature of Component	Atomic and Immutable Passive		Diffuse and Malleable Active		
Principle of Reuse	Composition		Generation		
Emphasis	Application Component Libraries	Organization & Composition Principles	Language Based Generators	Application Generators	Transformation Systems
Typical Systems	* Libraries of Subroutines	* Obj. Oriented * Pipe Archs.	* VHLLs * POLs	* CRT fmtrs * File Mgmt	* Lang. transfrmrs

Figure 1. Alternate Reuse Approaches

importing application. This phenomenon is quite rare and most often occurs when a system is a newer, enhanced version of an earlier one which supplies the subsystems that are reused (such examples exhibit the "use" rather than reuse of the subsystems). In such examples, the functionality of the subsystem component is exactly what is desired by a using application.

RitL, in general, has many facets and our experiences provide evidence that a generative approach has much to offer when there is an inexact fit between the operations and services provided by the subsystem and the needs of the using application. In such cases, one needs to reuse selected parts of the subsystem or to use adaptations of the subsystems for the new application. Managing the complexity that may exist within the subsystem and recognizing relationships that may cross module boundaries within the subsystem are often difficult. Subsystem components can have complex interfaces and subsystem operations may have hidden interdependencies. Careful design can lessen these sorts of problems but the reuser must become expert in the semantics of the subsystem in order to take maximum advantage of the subsystem.

An alternative to forcing the user to acquire the needed expertise is to package the subsystem capabilities within a generative framework. Generation technology can effectively support both of two basic scenarios. In one scenario, a complete, tailored subsystem is generated from scratch using a high level specification of the system. Such specifications are given in a high-level language, called an Application Specific Language (or ASL), that is constructed to support the application domain being served by the subsystem. ASLs [Simos87] as used and developed by Unisys have the following characteristics:

- they are often non-procedural
- they are implemented using generators
- they can be used to produce multiple outputs, such as source code in a high-level language (e.g., Ada) for a subsystem, as well as documentation for the generated subsystem.

In addition, Unisys has found that the use of ASLs and ASL generators provides a number of distinct advantages including:

- tailorability
- compactness
- domain orientation
- a higher level of correctness
- target (language and machine) independence.

If only a portion of a potentially large subsystem is required, only that portion can be generated from the specification. If full generality is not required, a simpler version with better performance and simpler interfaces can be produced from the same specification. A subsystem description written in an ASL is typically much shorter than the source code required to actually implement the system. For example, an ASL specification of a few hundred lines can produce a target language source file of ten thousand lines or more. In general, this reduction in size permits an ASL description to be completed in much less time than the subsystem it generates. A well-designed ASL uses the terminology of the application area, permitting the ASL user to ignore irrelevant details. Thus, the productivity of the ASL user measured in lines per day (to use one traditional measure) will be much greater than a programmer working directly on the subsystem source code.

The algorithms generated from ASL input are built in to the generator itself, and once verified, can be relied upon to be successfully and faithfully translated into the final application code. Thus, the programmer is prevented from making (and re-making) many mistakes in this code. ASL-based program generators are easily targetted to arbitrary programming languages and execution environments. Having a choice of target languages means that integration with existing systems written in a multitude of languages is easier. Moreover, once a generator has been implemented for one target system, retargeting to another has, in practice, proven to be inexpensive and straight-forward.

Unisys experiences with this sort of complete generation are best illustrated in the design and application of the Message Format Processing Language (MFPL) [Pollack87]. Complete and efficient message processing subsystems are generated from the specification of the message formats that the subsystem must be able to process. To reuse such a subsystem for a new or modified message format, one writes or changes the corresponding

MFPL specifications and the MFPL generator produces the application. More specifically, software engineers use MFPL to generate software modules that validate and generate messages within a larger message processing system.

Message Switching Systems (MSS) have been an important applications area for Unisys over the past two decades. Recently, MFPL has been a key component (along with the use of commercial off-the-shelf (COTS) database, user interface and communication packages) in the production of a new MSS with the following features:

- The MSS can be hosted on a PC whereas previous MSSs required mainframes.
- The MSS is *generic*, in the sense that it can be easily modified to use different formats or communication protocols.
- The MSS was developed at only a small fraction of the typical effort required to implement older message switches.

In particular, MFPL was used to increase productivity by at least ten times for the development of C language software to process formatted headers and trailers of standard NATO message formats. System changes resulting from modified message format specifications that used to take weeks or months to complete and debug, now require only a few minutes of MFPL generation and source code compiling time. A paper now in preparation will document the completely generative approach to subsystem reuse that MFPL exemplifies.

Such a totally generative approach is most effective when confined to a narrow, highly specialized domain. Another kind of reuse support through generation occurs when the subsystem is constructed by hand, but the interfaces to the subsystem are generated to meet the needs of the application. The partially generative approach outlined in the rest of this paper supports the effective reuse of existing handwritten subsystems whose full generality needs to be maintained. Rather than generate the entire subsystem, modules are generated which reference the handwritten subsystem and which contain validated operation calls on portions of the subsystem. The main goals of this approach are to reduce user errors and frustration while simultaneously increasing efficiency in the use of such subsystems. As such, the approach represents an enhancement to the use of abstract data types (ADTs) whose use embodies well-respected software engineering practice, and for which Ada was expressly designed and intended. The goals of the approach are accomplished through the use of abstract interface specifications that are used to generate actual subsystem operation calls. Such specifications provide an intermediate, supportive layer between the subsystem reuser and the subsystem being reused thereby providing less error-prone and more productive reuse.

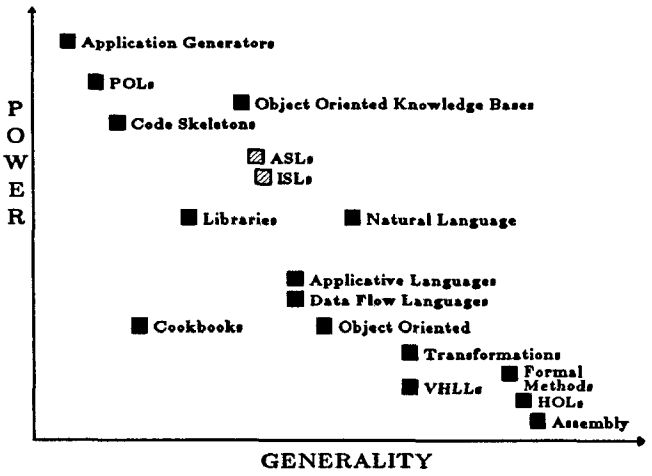

Figure 2. Spectrum of Reuse Technologies

As shown in figure 2, the problem with a generative approach is the lack of generality. Each problem domain requires its own generator and its own descriptive notation that captures the semantics of that domain. One would expect that the development of generators and associated languages would be an expensive proposition.

Unisys has overcome this economic argument through the use of a meta-generation system called SSAGS (Syntax and Semantics Analysis and Generation System) [Payton82]. SSAGS combines the generality of the use of VHLLs with the power of application generators (see figure 2). SSAGS is the basis for the full generation approach used in MFPL to support the domain of message processing as well as the generation of the Interface Specification Languages (ISLs) described in this paper. The next two sections discuss the design of subsystems to support the ISL methodology and the use of SSAGS to produce ISLs.

3. Designing Subsystems for Reuse

No matter whether a constructive or generative approach to *RitL* is taken, up-front design of the subsystem for reuse is desirable. In our case, we combined the principles of a layered system design along with data abstraction to promote the careful design of a family of abstract data types that comprise each of the two basic knowledge representation subsystems which are to be reused at the subsystem level (see figure 3).

Several design description levels were identified including the conceptual, data description, data structure,

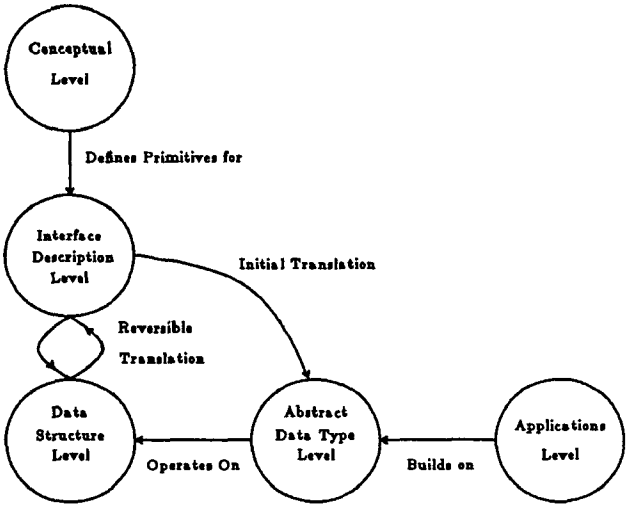

Figure 3. Design Levels

abstract data type and applications levels. The highest level is the conceptual level in which the basic concepts and operations to be provided by the subsystem are enumerated. This level can provide the seed for the descriptive language (the ISL) used to generate the interfaces to the actual package operations available at the abstract data type level. Moreover, the interface description level provides a means to abstractly refer to the objects and relationships being managed at the data structure level. In fact, one of the most common applications of an ISL specification in our work is generation of a routine to initialize the basic data structures required by a reusing application. This initialization is accomplished through the execution of operations contained in the actual abstract data types that make up the handwritten system. Actual manipulation of data structures is accomplished through the execution of operations contained in the abstract data type (ADT) level. Finally at the applications level, ADT operations are used to manipulate and modify the data structures required by the application.

Our experience has also shown the usefulness of having a two-way translation capability between the actual data structures and an ISL description of the contents of these data structures. The application may support the interactive access and modification of these structures, but having the ability to take a snapshot of these structures and recording them in the descriptive syntax of an ISL based on the conceptual level can lead to optimizations and modifications of the data structures that are hard to achieve interactively. Because an ISL specification provides an ASCII description of the application's data, it can also aid portability of the application's data structures to new operating platforms.

4. An ISL-Based Approach to Reuse

In order to effectively use ISLs to support the reuse of complex subsystems, there must be a mechanism in place to promote economic design of the ISL and the efficient production of the ISL processor that produces the desired interface to the services provided by the subsystem being reused. ISLs are in fact special kinds of ASLs. We have been most interested in the use and integration of such languages in the context of Ada [Simos87] and have explored the application of SSAGS in the production and application of ASLs. In fact, SSAGS itself is another example of an ASL and SSAGS is currently maintained in its ASL form.

Figure 4 is an adaptation of a figure that appeared in the MFPL paper referenced earlier showing the application of SSAGS to produce an ISL and the eventual use of the ISL in a reuse application. The ISL designer begins by describing the syntax and semantics of the ISL in terms of SSAGS input files. Some auxiliary Ada source files must also be provided by the ISL designer. As mentioned earlier, the conceptual level of the subsystem being described with the ISL can supply the basic vocabulary for the ISL.

The ISL input when run through SSAGS produces a collection of Ada source files that are compiled by an Ada compiler to produce the ISL processor system as an executable file (the middle oval in figure 4). The application design then proceeds along two fronts. ISL

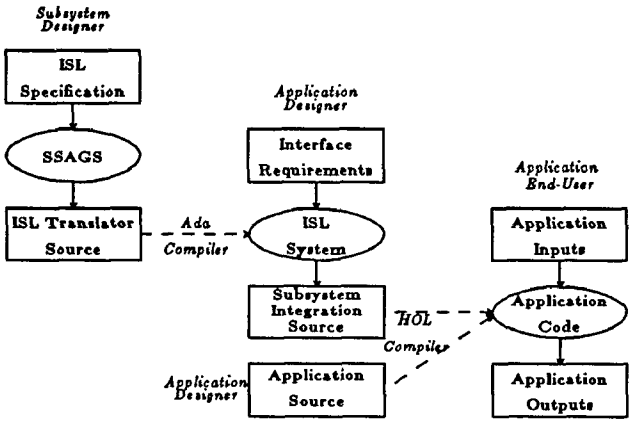

Figure 4. Developing and Using ISLs

specifications are written which capture the intended usage of the subsystem to be reused. These specifications are processed by the ISL processor to produce application code that calls upon the services provided by the subsystem being reused. The application developer also produces handwritten code to complete the body of the application. The complete collection of source files are then compiled to produce the resulting application.

In our case the HOL compiler indicated in the figure is also Ada. Note that in some cases, there may be several pieces to the application that are to be compiled and executed separately. For example, the ISLs described in the next section generally are used for data structure specification and initialization and the programs generated from the ISL descriptions are executed before the application code is actually run. The application outputs from this phase are the application's data structures stored in a persistent form as Ada files.

5. Some Examples of ISLs

The AdaTAU and AdaKNET knowledge base subsystems [Wallnau88] use ISL's to make specification of their knowledge bases simpler for the knowledge engineer. These ISL's are the Rule Base Description Language (RBDL) for AdaTAU and the Semantic Network Definition Language (SNDL) for AdaKNET. Both languages use an Ada-like grammar and syntax to define the data structures which form the knowledge base for each subsystem. The RBDL and SNDL translators take knowledge base specifications in their respective ISL and generate the Ada code which performs the calls on the subsystem ADT's to instantiate the knowledge bases. There is a third ISL called the Hybrid Knowledge Base Description Language (HKBDL) which is used to define the manner in which the knowledge bases of AdaTAU and AdaKNET will interact. For example, the HKBDL specification could specify which AdaTAU rule-based inference engines were associated with particular AdaKNET network concepts. This section will briefly describe the AdaTAU and AdaKNET subsystems and discuss the role of the RBDL, SNDL, and HKBDL ISL's in the reuse context of the three applications utilizing AdaTAU and AdaKNET.

AdaKNET is a structured inheritance network knowledge modelling system. The heart of each of our applications is an AdaKNET semantic network which captures and structures the information found in the domain of the application. For the Gadfly testing tool documented in [Wallnau88], AdaKNET captures knowledge about the structure of an Ada program and the methodology of constructing test cases. In the Intelligent Librarian [Solderitsch89, McDowell89], AdaKNET provides the hierarchical structure of an Ada reuse library and describes the attributes of library components. The Tool Utilization Assistant application was based in the domain of document-processing and the UNIX `roff` family of tools. Here, AdaKNET embodies knowledge about the structure of documents and the variety of `roff` mark-up commands.

```
concept ada_benchmark ( subprogram ) is
   local roles
      measured_features( 0 .. infinity ) of benchmark_feature;
   end local;
end concept;

concept acec_benchmark ( ada_benchmark ) is
   local roles
      control_measurement_component( 1 .. 1 ) of library_unit;
      instrument_package( 1 .. 1 ) of acec_instrument_package;
   end local;
end concept;

concept acec_composite_benchmark ( acec_benchmark,
                                   composite_benchmark ) is
end concept;

concept acec_performance_benchmark ( performance_benchmark,
                                     acec_benchmark ) is
   restricted roles
      subunits( 0 .. 0 );
      iteration_schemes( 1 .. 1 ) of fixed_loop_within_test;
      parameters( 0 .. 0 );
   end restricted;
end concept;

concept piwg_benchmark ( ada_benchmark ) is
end concept;

concept piwg_composite_benchmark ( piwg_benchmark,
                                   composite_benchmark ) is
end concept;

concept piwg_performance_benchmark ( performance_benchmark,
                                     piwg_benchmark ) is
   local roles
      io_package( 1 .. 1 ) of piwg_io_package;
      iteration_control_package( 1 .. 1 ) of
                                 piwg_iteration_package;
      optimization_control_package( 1 .. 1 ) of
       piwg_optimization_control_package;
   end local;
   restricted roles
      subunits( 0 .. 0 );
      iteration_schemes( 1 .. 1 ) of
                         clock_resolution_based_loop;
      parameters( 0 .. 0 );
   end restricted;
end concept;

individual wheta2 ( acec_composite_benchmark )
  is
end individual;
```

Figure 5. Fragment of Benchmark Library SNDL Specification

Knowledge is presented in a semantic network by describing objects and classes of objects in terms of relationships that exist between them. The three principal relationships provided by AdaKNET are specialization, aggregation and individuation. The *specialization* relationship declares that one class of objects (called a concept in AdaKNET) is a sub-class of a more general class where the sub-class inherits properties (in particular, all aggregation relationships) possessed by the parent class.

For example, in figure 5, an ada_benchmark is a kind of subprogram. Note that a given class may be a sub-class of several parent classes as illustrated by the piwg_performance_benchmark which is a kind of performance_benchmark and piwg_benchmark.

The *aggregation* relationship (called a role in AdaKNET) defines the properties or attributes of a class in terms of related classes. In figure 5, the class ada_benchmark is declared to have a role that defines the features being measured by the benchmark. The range (0 .. infinity) means that any individual benchmark may measure any number of features including the case of no features. Any subconcept lying below ada_benchmark in the network will automatically inherit this role. Subconcepts can also declare their own aggregation relationships (such as acec_benchmark which adds the roles for control_instrument_component and instrument_package). Moreover, a subconcept can further restrict both the number and type corresponding to an inherited role. The restricted roles declared in figure 5 provide examples of this behavior. The third kind of relationship is *individuation* which establishes that a given object is an instance of a class of objects. Figure 5 shows that wheta2 is an actual acec_composite_benchmark.

The SNDL ISL was used in the development of each of the clients of AdaKNET to define the AdaKNET semantic network underlying the application. SNDL can be used to fully describe an AdaKNET structured inheritance network. The knowledge engineer transfers his knowledge of the domain into a SNDL description of the domain. This SNDL specification is then translated into the Ada code calls to the AdaKNET subsystem which instantiate the semantic network. AdaKNET semantic networks can be browsed and built interactively with a tool from the AdaKNET subsystem, but a large and complex network can be more easily and quickly defined and understood from its SNDL specification. The SNDL specification allows understanding of the domain which would be much harder to gather from either browsing the network interactively or wading through the Ada code used to instantiate the network. Also, network information models of different depth, style, or complexity can be modelled with SNDL in the same way, without having to know the intricacies of the AdaKNET ADT calls which distinguish the different varieties of networks. SNDL grants the application programmer and knowledge engineer the full power of the AdaKNET structured-inheritance network model while keeping the interface simple and uncluttered so that concentration can be centered on modelling the domain knowledge.

To illustrate the power of using AdaKNET through SNDL, the complete SNDL spec shown in figure 6 was processed by the SNDL processor to produce the Ada output which is listed in figure 7. The first part of the listing illustrates the complexity of the necessary Ada environment that must surround the use of AdaKNET and a set of "housekeeping" declarations required to execute the rest of the generated code. The latter part of the listing shows the actual calls to the services provided by the semantic network packages. These services are not only called with the correct arguments, but have been arranged in the proper calling sequence.

```
network Animals is

  root concept thing is
  end root concept;

  concept higher_animal (thing) is
    local roles
        A_head (1..1) of head;
        A_front_leg (0..2) of front_leg;
        A_hind_leg (0..2) of hind_leg;
    end local;
  end concept;

  concept head (thing) is
  end concept;

  concept leg (thing) is
  end concept;

  concept front_leg (leg) is
  end concept;

  concept hind_leg (leg) is
  end concept;

  concept elephant ( higher_animal ) is
    local roles
        A_trunk (1..1) of trunk;
    end local;
    restricted roles
        A_front_leg (2..2) of front_leg;
        A_hind_leg (2..2) of hind_leg;
    end restricted;
  end concept;

  concept trunk (thing) is
  end concept;

  individual Joe ( elephant ) is
  end individual;

end Animals;
```

Figure 6. Complete SNDL Specification before Translation

AdaTAU is a rule-based inferencing system with partitionable distributed fact bases and the ability to interactively gather information. In our three applications, AdaTAU was used primarily to tailor the application for a particular instance, user, or run of the program. AdaTAU's ability to ask questions to gather information and then reason upon that information made it very suitable for gathering dynamic information not easily recorded in the static structure of the AdaKNET semantic network. In the testing tool application AdaTAU acquired information about the unit under test by asking questions of the user, and then inferenced to deduce a proper test case for the unit.

```
with Text_IO; use Text_IO;
with Fixed_Strings;
with Adanets;
with CommonIO; use CommonIO;
with Sndl_Support; use Sndl_Support;
with Roleset_Ranges; use Roleset_Ranges;
with Adanet_name_types;
procedure instantiate_sndl is
   SNDL_INSTANCE_NOT_GENERATED: exception;
   INSTANTIATE_ERROR: exception;
   New_Network: Adanets.Adanet;
   Network_Root: Adanets.Generic_Concept;
   Temp_gc: Adanets.Generic_Concept;
   Temp_ic: Adanets.Individual_Concept;
   New_Subroles: Adanets.Roleset_Spec_Sets.Set;
   Destroy_on_error: Boolean:= False;
   type ty_concept_k is (generic_k, individual_k, uninit_k);
   type concept_var(concept_k: ty_concept_k:= uninit_k) is
      record
         case concept_k is
            when uninit_k =>
              dummy_slot: Adanets.Generic_Concept;
            when generic_k =>
              generic_c: Adanets.Generic_Concept;
            when individual_k =>
              individual_c: Adanets.Individual_Concept;
         end case;
      end record;
   Concept_Array: array(1..9) of concept_var;
   function pad(
      Str: String;
      Len: Integer:=
                Adanet_Name_Types.Max_Object_Name_Length)
      return String
   renames Fixed_Strings.Pad;
begin
   begin
      Set_Boolean_Environment(
         Yes_No_Question => "Destroy network on error? y/n",
         Answer          => Destroy_on_error);
      Create_AdaKNET(
         New_Network  => New_Network,
         Network_Name => "animals");
      begin
         put_line("Renaming Root Concept");
         Temp_gc:= Adanets.Root_Concept(New_Network);
         Adanets.Rename(
           New_Network,
           Temp_gc,
           Pad("thing"));
         Concept_Array(9):= (generic_k, Temp_gc);
      exception
         when others =>
           put_line("Error Renaming Root");
           raise;
      end;
      Add_Generic_Concept(
         Network       => New_Network,
         Super_Concept => Concept_Array(9).generic_c,
         Concept_Name  => "higher_animal",
         New_Concept   => Temp_gc);
      Concept_Array(1):= (generic_k, Temp_gc);
      Add_Generic_Concept(
         Network       => New_Network,
         Super_Concept => Concept_Array(9).generic_c,
         Concept_Name  => "head",
         New_Concept   => Temp_gc);
      Concept_Array(2):= (generic_k, Temp_gc);
      Add_Role(
         Network          => New_Network,
         Role_Owner       => Concept_Array(1).generic_c,
         Role_Name        => "a_head",
         Min_Range        => 1,
         Max_Range        => 1,
         Value_Restriction => Concept_Array(2).generic_c);
      Add_Generic_Concept(
         Network       => New_Network,
         Super_Concept => Concept_Array(9).generic_c,
         Concept_Name  => "leg",
         New_Concept   => Temp_gc);
      Concept_Array(3):= (generic_k, Temp_gc);
      Add_Generic_Concept(
         Network       => New_Network,
         Super_Concept => Concept_Array(9).generic_c,
         Concept_Name  => "trunk",
         New_Concept   => Temp_gc);
      Concept_Array(7):= (generic_k, Temp_gc);
      Add_Generic_Concept(
         Network       => New_Network,
         Super_Concept => Concept_Array(1).generic_c,
         Concept_Name  => "elephant",
         New_Concept   => Temp_gc);
      Concept_Array(6):= (generic_k, Temp_gc);
      Add_Role(
         Network          => New_Network,
         Role_Owner       => Concept_Array(6).generic_c,
         Role_Name        => "a_trunk",
         Min_Range        => 1,
         Max_Range        => 1,
         Value_Restriction => Concept_Array(7).generic_c);
      Add_Generic_Concept(
         Network       => New_Network,
         Super_Concept => Concept_Array(3).generic_c,
         Concept_Name  => "front_leg",
         New_Concept   => Temp_gc);
      Concept_Array(4):= (generic_k, Temp_gc);
      Add_Role(
         Network          => New_Network,
         Role_Owner       => Concept_Array(1).generic_c,
         Role_Name        => "a_front_leg",
         Min_Range        => 0,
         Max_Range        => 2,
         Value_Restriction => Concept_Array(4).generic_c);
      Restrict_Generic_Roleset_Range(
         Network          => New_Network,
         Role_Owner       => Concept_Array(6).generic_c,
         Role_Name        => "a_front_leg",
         Min_Range        => 2,
         Max_Range        => 2);
      Restrict_Generic_Roleset_Type(
         Network          => New_Network,
         Role_Owner       => Concept_Array(6).generic_c,
         Role_Name        => "a_front_leg",
         Value_Restriction => Concept_Array(4).generic_c);
      Add_Generic_Concept(
         Network       => New_Network,
         Super_Concept => Concept_Array(3).generic_c,
         Concept_Name  => "hind_leg",
         New_Concept   => Temp_gc);
      Concept_Array(5):= (generic_k, Temp_gc);
      Add_Role(
         Network          => New_Network,
         Role_Owner       => Concept_Array(1).generic_c,
         Role_Name        => "a_hind_leg",
         Min_Range        => 0,
         Max_Range        => 2,
         Value_Restriction => Concept_Array(5).generic_c);
      Restrict_Generic_Roleset_Range(
         Network          => New_Network,
         Role_Owner       => Concept_Array(6).generic_c,
         Role_Name        => "a_hind_leg",
         Min_Range        => 2,
         Max_Range        => 2);
      Restrict_Generic_Roleset_Type(
         Network          => New_Network,
         Role_Owner       => Concept_Array(6).generic_c,
         Role_Name        => "a_hind_leg",
         Value_Restriction => Concept_Array(5).generic_c);
      Add_Individual_Concept(
         Network       => New_Network,
         Super_Concept => Concept_Array(6).generic_c,
         Concept_Name  => "joe",
         New_Concept   => Temp_ic);
      Concept_Array(8):= (individual_k, Temp_ic);
      Save_AdaKNET(New_Network);
      Close_AdaKNET(New_Network);
   end;
exception
   when SNDL_INSTANCE_NOT_GENERATED =>
      put_line("rename your network");
   when others =>
      if Destroy_on_error then
         Adanets.Destroy_Adanet(New_Network);
      end if;
      raise;
end;
```

Figure 7. Ada Source produced by SNDL

In the Intelligent Librarian, a distributed AdaTAU guides the user through the hierarchy of the reuse library by asking questions about what the user is searching for and then steering the user in the right direction. For the Tool Utilization Assistant, AdaTAU was targeted for determining the location of help files and providing a user-tailored presentation of information about the tool.

The RBDL ISL defines the different flavors of the AdaTAU inferencing context in our three applications and generates the Ada code necessary to build the inferencer constructs which provide the functionality of the subsystem. RBDL contains syntax for describing fact and rule bases, and an association of fact and rule bases which compose an AdaTAU inferencer. It also describes the interface necessary to have the AdaTAU inferencers act in a cooperating, distributed manner, changing an inferencing focus and passing facts between different inferencers. In our three applications, using RBDL to define and generate the data structures for AdaTAU allowed the knowledge engineers to use the power of the inferencing subsystem without having to tediously code the calls to AdaTAU that build inference context structures. The AdaTAU subsystem calls necessary to build the complex inter-weaving of distributed rule base inferencers composing an application's inferencing context would be very hard to follow in their Ada format; but the same knowledge base specified with RBDL can be followed logically and clearly. Figure 8 shows an example of the RBDL ISL from the Intelligent Librarian application. The complexity and length of the Ada interfaces to the AdaTAU subsystem generated from a RBDL specification are comparable to those shown in the SNDL output file (figure 7).

All three of our applications which reuse the AdaKNET and AdaTAU subsystems use them in a hybrid knowledge representation scheme. This scheme allows objects to be associated with the concepts of an

```
fact base schema Acec_Composite_Benchmark_Facts is

   numerical_computation_benchmark_type : one_of
      (dhrystone, whetstone, henessy);
   benchmark_use : one_of
      (numerical_computation, capacity_test, multiple);
   uses_pragma_suppress : one_of
      (yes, no, immaterial);

end Acec_Composite_Benchmark_Facts;

initial fact base Acec_Composite_Benchmark_IFacts is
   null;
end Acec_Composite_Benchmark_IFacts;

question base Acec_Composite_Benchmark_Questions is

   question Ask_Benchmark_Uses is
      text : {Will this benchmark be used for
              numerical computations?};
      type : one_of;
      responses :
        "yes" => (benchmark_use, numerical_computation);
        "no" => (null_attribute, null_value);
        "not solely" => (benchmark_use, multiple);
   end question;

   question Ask_Kind_Of_Computation is
      text : {Which response best describes the
              kind of computation done?};
      type : one_of;
      responses :
        "systems and program" =>
           (numerical_computation_benchmark_type, dhrystone);
        "science and math" =>
           (numerical_computation_benchmark_type, whetstone);
        "classic and various computations" =>
           (numerical_computation_benchmark_type, henessy);
   end question;

   question Ask_Uses_Pragma_Suppress is
      text : {Does the benchmark make use
              of pragma SUPPRESS?};
      type : one_of;
      responses :
        "yes" => (uses_pragma_suppress, yes);
        "no" => (uses_pragma_suppress, no);
        "doesn't matter" => (uses_pragma_suppress, immaterial);
   end question;

end Acec_Composite_Benchmark_Questions;
```

Figure 8. Sample of the RBDL ISL

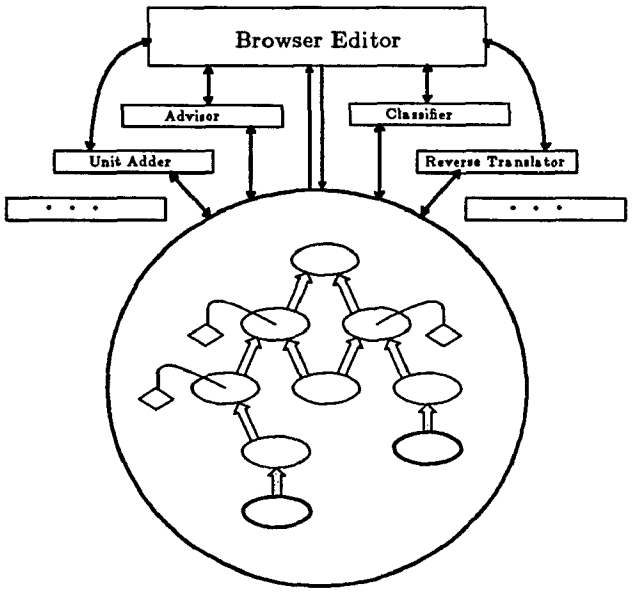

Figure 9. Hybrid Knowledge Representation in the Intelligent Librarian

AdaKNET network. For example, AdaTAU inference bases are attached to AdaKNET concepts to provide localized guidance about the network surrounding the concept. The association between concepts and objects is established with a "hybrid state layer" describing which particular concepts are bound to which objects. Figure 9 shows the hybrid knowledge representation scheme as applied in the Intelligent Librarian application. The diamond shapes indicate attached inference components within a larger hierarchical collection of concepts.

At first, during development of the Gadfly unit test application, the Ada code establishing the hybrid state layer was hand-coded. As the size of the hybrid knowledge representation grew and the variety of objects that could be associated with AdaKNET concepts was enhanced, it became apparent that the procedure to build the hybrid state layer was becoming long and repetitive, and hand-coding it was prone to error. To meet this problem, the Hybrid Knowledge Base Description Language was rapidly developed to allow a simplified specification of the hybrid state layer which would generate the increasingly complex procedure which instantiates it.

HKBDL is a small ISL which simply describes what objects are associated with concepts of the AdaKNET semantic network. Using HKBDL, like RBDL and SNDL, helps the knowledge engineer concentrate on modelling by removing complex details of the Ada implementation. HKBDL also makes reuse of cooperating AdaKNET and AdaTAU knowledge bases easier to manage and establish. Many possible hybrid knowledge representation schemes are possible, and HKBDL's syntax will be adapted accordingly as multiple schemes evolve. Notable about HKBDL is the speed with which it was developed and implemented to remove a growing problem area. Figure 10 shows an example of the HKBDL ISL from the Intelligent Librarian.

```
hybrid benchmark_library ( benchmarks ) is

   text file "pruaa2DE" at ind000_pruaa2_design;
   text file "pruaa2AB" at ind000_pruaa2_abstract;

   text value "Simtel-20" at simtel_20;
   text value "/library/demo_dir" at demo_dir;
   text value "p000007.a" at p000007_a;
   text value "p000006.a" at p000006_a;
   text value "bgt.o" at bgt_executable;

   inference base composite_benchmark;
   inference base acec_composite_benchmark;
   inference base piwg_composite_benchmark;
   inference base bgt_generated_composite_benchmark;

end benchmark_library;
```

Figure 10. Sample of the HKBDL ISL

6. Lessons Learned

One of the goals of the RLF project is the general promotion of reuse practices and concepts including the use of generation technology. By choosing to use this technology ourselves to promote the reuse of internal RLF subsystems, we hope to learn more about the effectiveness and capabilities provided by the technology. To date, we have reused the basic RLF knowledge representation subsystems in three different applications and developed three ISLs. In so doing, we have accumulated a number of observations.

First and foremost, the use of ISLs really does aid reuse. This fact was demonstrated by the use of the RBDL language by a non-Ada programmer to build knowledge bases for the Gadfly testing tool application. Without RBDL, the individual would have been forced to become an expert on the AdaTAU ADTs which would have greatly delayed progress in developing usable knowledge bases for the Gadfly application. Through the use of UNIX shell scripts, the operation of the Ada compiler in compiling and executing the programs generated from a RBDL specification can be completely hidden from the novice user. Moreover through RBDL, the operations called upon from AdaTAU are used in a safe, reliable manner. This reason promotes the use of RBDL even by programmers experienced in the AdaTAU ADTs.

Secondly, while there is some overhead in learning to design an ISL and to use SSAGS to generate the ISL processor, once this price has been paid it becomes very natural and efficient to modify and enhance the ISL. Since the language designer is working at a very high level, a new ISL processor can be produced from a changed specification in a very short period of time (the length depending on the number and severity of the changes made to the language definition). Overall, we found production of ISLs to be economical and practical.

On the other hand, there has been at least one negative result. The Ada files produced by the ISL translators may not be very palatable to today's generation of Ada compilers. The basic problem is that compiler writers develop compilers to handle programs that people would write, not those that might be generated. Generators, however, just produce legal programs that can exceed compiler internal limits. We have observed problems especially with the size of generated compilation units. Compilers just don't expect units with hundreds or thousands of lines per unit. Such large units take an unusually long time to compile (if they compile at all) and often have inefficient or erroneous executions as well. Of course, the language designer can adjust the code generation process employed, but the resulting code can often be less readable or traceable to the original ISL specification. While it is possible for the language

designer to accommodate several compilers, it will be impossible to account for a majority of them, thereby impacting portability of the generated code.

Summary

In this paper we have shown how *RitS* and *RitL* are quite different in nature and how the latter can be supported through the use of generation technology. In particular, the use of a meta-generation system such as SSAGS provides a cost-effective solution to the problem of reusing existing complex subsystems based on partial generation. Three distinct ISLs were introduced and their application in support of reuse at the subsystem level was illustrated. We believe that with careful design of the subsystem, the use of specialized interface description languages promotes the visibility and understandability of the subsystem and thus improves the reusability of the subsystem. We plan additional work in the area of language-based interfaces in support of *RitL*.

Acknowledgements

The design and implementation of the Reusability Library Framework was originally funded under the STARS Foundations program as contract number N00014-88-C-2052 administered by the Naval Research Laboratory. This paper is a revision of the paper "Experience in the Reuse of Ada Subsystems" presented at the Third Annual Unisys Software Engineering Symposium, January 1990, in Houston Texas. Parts of this paper were inspired by discussions with fellow researchers at the Unisys Center for Advanced Information Technology (CAIT). In particular, the authors wish to thank Bob Pollack, Don McKay, Ray McDowell and John Thalhamer of CAIT for their contributions.

References

[Berrett87] C. Berrett, "Engineering for Reuse: The Goal of Ada," *Proceedings of the First Annual Unisys Defense Systems Software Engineering Symposium*, McLean, VA, September 1987.

[Biggerstaff87] T. Biggerstaff and C. Righter, "Reusability Framework, Assessment, and Directions," *IEEE Software*, 4(March 1987), pp. 41-49.

[Biggerstaff89] T. Biggerstaff and C. Righter, "Reusability Framework, Assessment, and Directions," in *Software Reusability Volume 1 Concepts and Models*, T. J. Biggerstaff and A. J. Perlis (editors), ACM Press, 1989, pp. 1-17.

[McDowell89] R. McDowell and K. Cassell, "The RLF Librarian: A Reusability Librarian Based on Cooperating Knowledge-Based Systems," *Proceedings of RADC 4th Annual Knowledge-Based Software Assistant Conference*, Utica, NY, September 1989.

[Payton82] T. F. Payton, S. E. Keller, J. A. Perkins, S. Rowan, and S. P. Mardinly, "SSAGS: A Syntax and Semantics Analysis and Generation System," *Proceedings of COMPSAC '82*, 1982, pp. 424-433.

[Pollack87] R. Pollack, W. Loftus, and J. Solderitsch, "A Generative Approach to Message Format Processing," *Proceedings of the First Annual Unisys Defense Systems Software Engineering Symposium*, McLean, VA, September 1987.

[Simos87] M. A. Simos, T. F. Payton, and R. H. Pollack, "Integration of Fourth Generation Languages and Ada," in *Proceedings of the USAISEC Technology Strategies '87 Conference*, Alexandria, Virginia, February 1987.

[Solderitsch89] J. Solderitsch, K. Wallnau, and J. Thalhamer, "Constructing Domain-Specific Ada Reuse Libraries," *Proceedings of Seventh Annual National Conference on Ada Technology*, March 1989.

[Wallnau88] K. Wallnau, J. Solderitsch, M. Simos, R. McDowell, K. Cassell, and D. Campbell, "Construction of Knowledge-Based Components and Applications in Ada," *Proceedings of AIDA-88, Fourth Annual Conference on Artificial Intelligence & Ada*, November 1988, pp. 3-1 through 3-21.

Biographical Sketches

James Solderitsch is the chief programmer of the Reusability Library Framework (RLF) project. Dr. Solderitsch joined the Unisys Defense Systems Software Technology Laboratory in January 1986 after having been an assistant professor of Computer Science at Villanova University for 8 years. His primary interests are software reusability and the impact of very high level, domain-oriented, specification languages on software productivity and reusability. He holds a B.S. degree in Mathematics from Villanova University and an M.S. and Ph.D. degrees in Mathematics from Lehigh University.

Since joining Unisys in June, 1988, Timothy Schreyer has spent most of his time with the Reusability Library Framework (RLF) project involved in design and implementation tasks. He also helped integrate the STARS program ACE/CAIS-A Baseline Software Engineering Environment (SEE). Currently, he is working under the STARS program implementing an Ada Xt toolkit. Mr. Schreyer graduated from Bucknell University in May, 1988, with a B.S. degree in Computer Science.