Code-Data Consistency in Ada

Vincent M. Celier
Drasko Sotirovski
Christopher J. Thompson

Hughes Aircraft of Canada
13951 Bridgeport Road
Richmond, B.C. V6V 1J6
Canada
tel: (604) 2313205
fax: (604) 2785625
(vmcelier, dmsotirovski, cjthompson)@ccgate.hac.com

Abstract. This paper describes the difficulties that the authors have experienced in maintaining existential consistency between Ada code and the various persistent storages that the code depends on. It identifies certain patterns in the existential relationship between the code and data, and describes how Ada's expressive power and ASIS (Ada Semantic Interface Specification) have been used to overcome these difficulties.

Keywords. Ada, Ada Semantic Interface Specification, Air Traffic Control

Introduction

One common expectation from the user's perspective of a software product — be it simple software components like stacks, lists or queues, popular desktop spreadsheets and text editors, operating systems or complex real-time control systems — it is an implicit expectation of flexibility. The software product is expected to easily adapt its behavior, capacity, or performance to suit the needs of a particular user, site, or installation. The Canadian Automatic Air Traffic System (CAATS) being built by Hughes Aircraft of Canada is no exception: one and the same software is required to support a dozen sites different in size, traffic patterns, operational procedures, etc.

Software adaptability is often a functional requirement: CAATS is, for example, required to adapt to the actual structure of airways, airspace partitioning between different controllers, etc. However, software adaptability is as often a designers' response to unknown or unstable phenomena: the actual text of an error message to be presented to the user or the actual size of an array is often better not to hardcode into the system. CAATS experienced some difficulties in managing this latter category of adaptability and this paper describes the nature of these difficulties and how CAATS is using Ada [1] and ASIS (Ada Semantic Interface Specification) [2] to overcome them.

The unique property of this kind of adaptability is that elements of the source code define the requirement for the existence of a corresponding data item in some persistent storage. This paper explores a technique for maintaining this existential relationship between the source code and data items in other persistent storage.

The Model Problem

To explore some of the difficulties and chosen solutions, we start with the following model problem. Correct sizing of various buffers and arrays is often difficult to predict and, therefore, software designers often opt to make the software flexible relative to the actual sizing: the actual value is stored in a persistent storage and used at elaboration time or at start-up. CAATS supports a notion of system parameters through the following (simplified) Ada design:

```
package System_Parameters is

  procedure Save_All_Collections;
  procedure Load_All_Collections;

end System_Parameters;

generic
  Name: in String;
  type Parameter is (<>);
package System_Parameters.Collection is

  generic
    type Value is (<>);
  package Discrete_Kind is
    procedure Define   (The_Parameter: in Parameter;
```

```
                          With_Value:    in Value);
      function  Value_Of (The_Parameter: in Parameter) return Value;
    end Discrete_Kind;
    ...
  end System_Parameters.Collection;
```

A “System Parameter Collection” is a set of “system parameters”. Each system parameter is identified by a value of the generic parameter discrete type Parameter. A system parameter can represent a discrete value or a real value. Here we only present the discrete case, but there could be similar generic packages such as Floating_Point_Kind and Fixed_Point_Kind.

To define “offline” the different collections of system parameters, one calls the procedure Define of an instantiation of Xxxx_Kind for each parameter in each instantiation of Collection, then calls the procedure Save_All_Collections. This stores all system parameters collections in some persistent form such as in a file.

To get “online” the current value of a system parameter one uses the function Value_Of for the parameter in the corresponding instantiation Xxxx_Kind of the instantiation of its Collection. Load_All_Collections must have been called before any call to a function Value_Of; one option is to call it in the body of package System_Parameters.

The actual sizing of various buffers can now be left unspecified until elaboration or later. One can define the sizing to be a system parameter:

```
type Number is range 0 .. 10_000;
type My_Parameter is (Number_Of_X, Size_Of_Y, ...);
package My_Collection is new
           System_Parameters.Collection
               (Name      => "My_Collection",
                Parameter => My_Parameter);
package My_Number_Parameter is new
           My_Collection.Discrete_Kind (Number);
```

Before running the subject application, the required values can be defined and saved offline:

```
My_Number_Parameter.Define (Number_Of_X,   25);
My_Number_Parameter.Define (Size_Of_Y,   4711);
...
System_Parameters.Save_All_Collections;
```

And the code will use them online for sizing purposes during initialization:

```
type Array_Of_X is array (Number range <>) of Number;
...
System_Parameters.Load_All_Collections;
```

```
...
declare
  X_Storage_Size: constant Number :=
      My_Number_Parameter.Value_Of (Number_Of_X);
  X_Storage: Array_Of_X (1 .. X_Storage_Size);
begin
  ...
```

The implementation of the package System_Parameters relies on a persistent storage: each and every system parameter that the software will ask for has to be in it. Consequently, there is now an existential relationship between declarations in the source code and items in a persistent storage.

The First Difficulty: Depending on other unknown system parameters

Designers are relieved by the fact that they don't have to make the sizing decisions up front. Nor to invent their own mechanism for configuring the software [4] [5]. Soon everyone is busy designing his own class(es) and setting up a number of system parameters to be used for sizing purposes. Writing the code goes smoothly but <u>running</u> it is not as easy:

- To run a test program, one has to prepare not only the values of his own system parameters but **all system parameters from the closure** of the unit being tested. In other words, **code-data consistency** has to be ensured: the persistent storage must contain the system parameters that the Ada code being executed depends on.

  First of all, for many parameters further away in the closure, the designer often has no idea what they are used for and what values to set.

  Second, because the system parameters are result of the design process, every release of the software tends to slightly change the set of parameters.

From the development point of view, each designer is interested in proving proper behavior of his/her code. This includes proving the appropriate behavior of the code for **any** set of values of the parameters that the code depends on. However, the very same designer is virtually never interested in the configuration data used in other classes her code depends on **as long as they behave** as specified. Hence, the problem can be resolved by means of:

Default Behavior

To ease the development process, if values for some system parameters are not found in the persistent storage, each class should use a default set of values found in the code:

```
package System_Parameters is
```

```
   type Data is private;

   generic
     type Value is (<>);
   function Data_Of (The_Value: in Value) return Data;

   procedure Save_All_Collections;
   procedure Load_All_Collections;

 end System_Parameters;
```

The type Data encapsulates any system parameter of any discrete or real type. For any type of system parameter, a function Data_Of must be instantiated. This instantiation can be used in several collections.

```
with System_Parameters;
generic
  Name: in String;
  type Parameter is (<>);
  type Default_Value is array (Parameter) of System_Parameters.Data;
  Default: in Default_Value;
package System_Parameters.Collection is
  ...
  procedure Define   (The_Parameter: in Parameter;
                      With_Value:    in System_Parameters.Data);
  ...
  generic
    type Value is (<>);
  function Value_Of (The_Parameter: in Parameter) return Value;
  ...
end System_Parameters.Collection;
```

Generic package Collection now has a generic parameter array Default that contains the default values of the system parameters of the collection. If the collection is not saved in the persistent storage, then its value is contained by default in the array Default.

This provides means for the designer to declare default values for each of the system parameters required by the implementation. (It should be noted here that although the designer felt uncomfortable in setting the final value of the system parameter, we found that designers have very little problems in setting up reasonable default values.)

Using the same example as before, the implementation of a class sets up default values

```
type Number is range 0 .. 10_000;
type My_Parameter is (Number_Of_X, Size_Of_Y, ...);
type Default_Value is array (My_Parameter)
                               of System_Parameters.Data;
function Data is new System_Parameters.Data_Of (Value => Number);
Default: constant Default_Value := (Number_Of_X => Data (  25),
                                    Size_Of_Y   => Data (4711),
                                    ...);
package My_Collection is new
           System_Parameters.Collection
               (Name          => "My_Collection",
                Parameter     => My_Parameter,
                Default_Value => Default_Value,
                Default       => Default);
function Get_Value is new My_Collection.Value_Of (Value => Number);
...
System_Parameter.Load_All_Collections;
declare
  X_Storage_Size: constant Number := Get_Value (Number_Of_X);
```

As the above code fragments show, the implementation of System_Parameters is refined: if the persistent storage does not specify a value for, say, Number_Of_X, the default value is assumed.

By providing default values for all system parameters, the designer removes the need for the clients to even know about them — for as long as the default behavior suits their needs.

The Second Difficulty: Gathering all system parameters for integration

Once equipped with default behavior, data dependencies between classes become equivalent to the Ada dependencies between compilation units and designers are busily refining their code and producing new system parameters and new releases of the software. At some point, however, all that software has to be integrated into a single system. And, in iterative software development [6], integration is not a singular activity at the end of the software development cycle but a continuous process: the software developed steadily flows into integration. The same problem reappears:

- To integrate the system, one has to prepare values for **all system parameters contained in the system**.

  First of all, the default values of many system parameters do not meet the needs of the integrated system and many of them will have to be tuned to balance system performance and resource utilization.

  Second, different integration tests may require different sets of system parameters. (For example, the maximum load test is likely to require its own set of system parameters.)

  Finally, once the system has been delivered, many of the system parameters will yet have to be tuned to meet the needs of the particular site or user.

And so, the same problem of **code-data consistency**, postponed by means of default values, strikes again. And we ask ourselves: wouldn't it be nice if the consistency rules of Ada can be extended to the persistent storage so that it is guaranteed that the persistent storage contains all required system parameters?

ASIS Comes in Handy

Ada based systems offer an ideal opportunity for automating collection of data that originate from the code — ASIS. Unlike any other language, Ada provides through ASIS a means for constructing a multitude of tools for collecting information contained in the code. In our example, all system parameters contained in the released code can be collected together with their default values.

In our example, ASIS is used to find all instances of System_Parameters collections, the enumeration of parameters in the collection and their default values.

Because integration is a continuous activity on CAATS, ASIS tools will not only collect the required information from the code but also compare it with the footprint of the previous integration and issue difference reports. E.g. a report should be issued that lists all system parameters that have disappeared from the code since previous integration as well as all new system parameters introduced into the released Ada code. These reports provide insight for the integrators into the nature and scope of the changes and a starting point for understanding how to tune the system. (Figure 1)

In some cases, if system parameters are changed often or the end-user is allowed to change the system parameters, fancier and more friendly front-ends need to be built. However, the pattern remains the same, ASIS based tools are built to extract the necessary information from the Ada code and transform it into the format that the more elaborate front-end requires. For example, the information collected from the Ada code can be transformed into SQL format that many user friendly tools can manipulate.

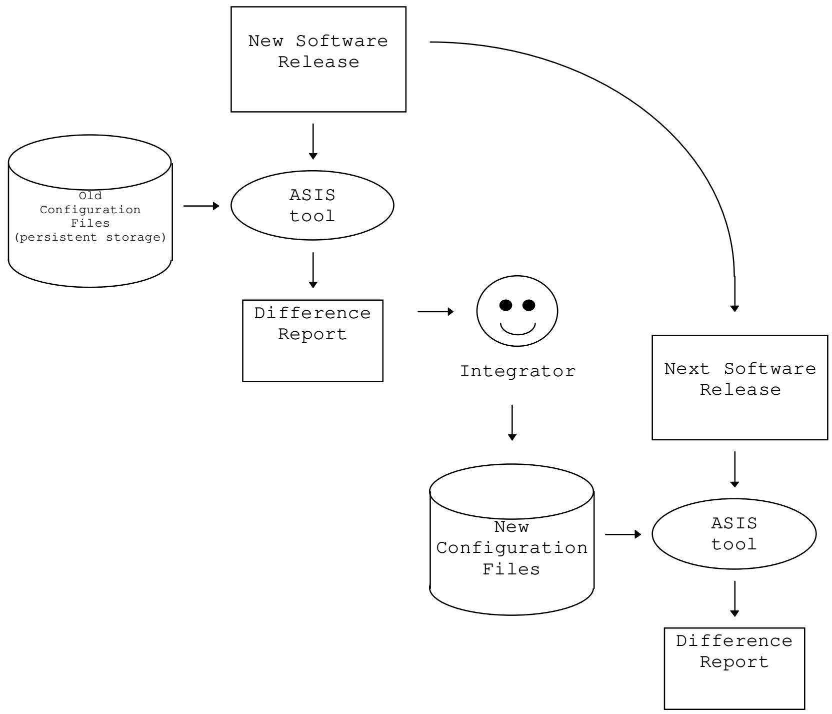

Figure 1: The role of ASIS

In CAATS, we keep most of our data in a relational database. From the database we produced offline “fast load files” that are used online by the deployed software. Because we are quite confident that the fast load files produced represent exactly what is in the database, our tools for data-code consistency checks the code against the database. To access the database, we use SAMeDL [7], both for the production of the fast load files and for the data-code consistency checking. To access the code, we use ASIS in an environment that has visibility on all the developed code.

In the database, each collection is identified by its name: it should be the same name as the generic parameter Name of the corresponding instantiation of Collection. Each system parameter in a collection is identified by its name. All generic parameter Parameter are enumerated types. The name of system parameter in the database is the identifier of the corresponding type Parameter.

Here are the different steps of a tool that checks the system parameter collections:

- From the database, using SAMeDL generated subprograms, load into memory all system parameters of all collections.
- Using ASIS, open the Ada environment context.
- Identify the library unit System_Parameters.Collection.
- Find all library units that are "direct dependents" of System_Parameters.Collection.
- For each dependent library unit:
    - Iterating on its elements, find the instantiation of System_Parameters.Collection.
    - Get the value of the actual parameter Name of the instantiation: this is the name of the collection: if the collection is in the database, mark it as found; if it is not, report that the collection is not in the database, and continue with the next library unit.
    - Get the identifiers of the generic parameter Parameter. For each of them, if it is the name of a system parameter in the database, mark it as found. If it is not in the database, report this fact.
    - If there are any system parameters in the database that is left unmarked, report them.
- If there is any collections in the database that is left unmarked, report them.
- Close the Ada environment context.

It is envisioned that such a tool will make use of ASIS in a standard fashion as described in [3]. The CAATS system currently uses a System Parameter mechanism similar to the simplified one described in this paper. In the over 400K SLOCS currently integrated, there are over two hundred system parameters in use and we expect to have more than twice that many by the end of the project.

Conclusion

The development of the CAATS system has seen the following pattern again and again:

- Ada design in support of behavior adaptable through configuration data
- requires default behavior during software development and
- ASIS tools to maintain code-data consistency during system integration and maintenance.

Other examples that we believe are common for many systems include the following:

- CAATS software is capable of detecting a number of erroneous system events (e.g. storage problems, disk availability or some other problems) that need to be reported to the maintenance personnel. The actual text of the message is adaptable and the fast load file requires that each and every event that the system can detect is provided with appropriate text. During development a default text can be used for as long as the developers can understand the message being issued. Before delivery, ASIS tools are used to ensure that fast load files with text messages for the end-user are in fact consistent with the actual set of events that the software can report.
- When a user request cannot be accomplished, CAATS application software returns a semantic response, a description of what went wrong and why. Different semantic responses have to be presented to the user as different text messages. Again, during development, a default textual representation will suffice as long as the developers can understand it. And again, late in the integration phase, configuration data that describe each and every semantic response that the system can return has to be prepared for the end-user.

In the past, CAATS has made use of a variety of ad-hoc development tools to help manage this kind of configuration data. It is only through the experiences of iterative software development and working to keep the system integrated at all times, that we have discovered the need for such tools. Our current focus is on enhancing the “integration toolset” to exploit the capabilities offered by ASIS.

References

[1] ISO/IEC 8652:1995, Information Technology - Programming Languages, their environments and system software interfaces - Programming Language Ada.

[2] Association for Computing Machinery (ACM) Special Interest Group in Ada (SIGAda) ASISWG/ISO/IEC JTC 1 SC 22/WG 9 ASISRG, ASIS WOrking Draft, Version 2.0.j, 1 November 1996, available World Wide Web site at => hppt://www.acm.org/ sigada/WG/asiswg

[3] Architecture of ASIS - A tool to Support Code Analysis of Complex Systems, Colket, et el. ACM Ada Letters, Volume XVII, Jan/Feb 1997

[4] Philippe Kruchten & Christopher Thompson, "An Object-Oriented, Distributed Architecture for Large Scale Ada Systems," Proceedings of the TRI-Ada '94 Conference, Baltimore, November 6-11, 1994, ACM

[5] Ch. J. Thompson and V. Celier, “DVM: an object-oriented framework for building large distributed Ada systems,” Proc. of Tri-Ada’95, Anaheim, Ca., Nov. 1995, pp.

[6] Philippe Kruchten, Christopher J. Thompson, "Iterative Software Development for Large Ada Programs," Alfred Strohmeier (Ed.), Reliable Software Technologies - Ada-Europe'96: proceedings, Lecture Notes in Computer Science, vol 1088, Springer, Ada-Europe International Conference on Reliable SoftwareTechnologies Montreux, Switzerland, June 10-14 1996, ISBN 3-540-61317-X

[7] ISO/IEC 12227:1994, Information Technology - Programming languages, their environments and system software interfaces - SQL/Ada Module Description Language SAMeDL