Automatic Serialization of Dynamic Structures in Ada

Technical Report IC/2003/63

Rodrigo García García[1], Alfred Strohmeier[1], and Lukas Keller[1]

Software Engineering Laboratory,
Swiss Federal Institute of Technology Lausanne (EPFL),
CH-1015 Lausanne, Switzerland
`{rodrigo.garcia, alfred.strohmeier, lukas.keller}@epfl.ch`

Abstract. Serialization is the process that transforms the state of a software object into a sequence of bytes. Serialization is useful, for example, to store the value of an object in persistent memory or to send it through a network channel. Ada provides object serialization by means of the `Streams` package. However, default Ada serialization does not address the problem of serializing dynamic structures automatically. The streaming of an access variable in Ada flattens the value of the access variable itself and not the value of the object referenced by it. It is the responsibility of the programmer to write custom serialization routines for the lists, trees, stacks or any other dynamic structure requiring serialization. In this paper, we present an algorithm for the automatic generation of serialization subprograms for dynamic structures in Ada. We will use ASIS for analyzing the structure of the types to be serialized.

1 Introduction

Serialization, also known *flattening* or *marshalling*, is the process that takes the data contained in a software object and outputs it as a sequence of contiguous elements, usually bytes. A sequence of bytes is a convenient format for storing the state of an object in a file or for sending it through a serial communications channel, among other possible uses. The data in its serialized form can later be used to reconstruct an object with the same state than that which was stored or transmitted.

1.1 Serialization in Ada

Ada has an elegant and adaptable object-oriented solution to the serialization problem: the package `Streams` [1]. The abstract type `Root_Stream_Type` is declared in this package, along with the abstract procedures `Read` and `Write`. These procedures must be implemented for every concrete (non abstract) descendant of `Root_Stream_Type`. The code in these procedures is adapted to read from and write to a particular kind of media, represented by the concrete descendant of `Root_Stream_Type`.

Ada also defines stream-oriented attributes for all non limited types: `Read`, `Write`, `Input` and `Output`. These attributes convert values to their sequential representation (a stream of elements) and reconstruct values from a stream. An access to the stream is passed as a parameter to the stream-oriented attribute subprogram. In this way, the subprogram will be able to use the `Write` and `Read` procedures specific to that concrete stream. The stream-oriented attributes are defined for class-wide types as well. In this latter case, the call to an attribute subprogram will dispatch to the corresponding attribute subprogram of the appropriate specific type (the way to determine this specific type depends on the attribute used).

The stream-oriented attributes can be specified for any type by using attribute definition clauses and thus override the default serialization subprograms. This allows the programmer to control the way objects of a certain type are serialized.

In addition to package `Streams`, Ada provides one concrete implementation of it for reading from and writing to files: the package `Streams.Stream_IO`. Thanks to this package, the programmer can get access to a stream representing the file and then write values to the file or read values from the file using the stream-oriented attributes, as with any other kind of stream.

1.2 Default Serialization of Access Types in Ada

We have just seen that Ada proposes a clean and extensible mechanism for streaming. However, the default serialization of access types in Ada is rather simplistic and not very useful. Access types are considered non limited elementary types in Ada. Therefore, the serialization of their value is implementation defined (RM 13.13.2 (36)). The fact that their serialization is implementation dependent is not relevant for us at this point. The important issue about serialization of access types is that it is the value of the access variable which is serialized and not the value of the object pointed to by the access variable.

In order to serialize a user defined dynamic structure, the programmer will have to specify the stream-oriented attributes for it. Default attribute implementation would be useless, since references to other objects would not be followed. It is the programmer who must control the serialization process of the whole structure, putting in the stream all the necessary information for being able to reconstruct the structure later.

In our opinion, this puts a lot of burden on the programmer. Many modern languages already provide the mechanisms to make the programmer's life easier by automatically serializing all objects referenced by the object to be streamed (see [2] and [3]). With a bit of run-time support, we think that this overhead can be removed from the work of the Ada programmer as well. In Sect. 2, we present an algorithm based on the ideas found in [2] that performs the automatic serialization of dynamic structures.

2 The Algorithm

One of the issues that have to be considered when serializing dynamic structures is that they can have circularities. If we just ignore the circularities, the serialization process falls into infinite loops. In order to avoid loops, we keep a list of the identifiers of all the referenced objects[1] already serialized. We identify each referenced object using the value of the access variable that references it. This value should be unique for each

[1] By referenced object, we mean one that is pointed to by an access variable.

referenced object, because of the properties of the equality operator for access types. Thus, if two access variables are equal, they point to the same object; if not, they point to different objects. Obviously, this condition holds only if the access variables are of the same access type, otherwise they are not comparable.

For the sake of simplicity, but without loss of generality, we will assume that we have only one access type defined while presenting our serialization algorithm. We will see later how to generalize the result. The algorithm for serializing an object is as follows (see Fig. 1 for an example):

1. Serialize the elements that compose the object using default Ada serialization, except for the elements which are access variables.
2. When an access variable is reached, first check whether its value is already in the list of identifiers.
   (a) If the value is `null` or it is already in the list, serialize just the value of the access variable.
   (b) If the value is not in the list, serialize it and then also serialize the object to which the access variable is pointing. Put the access value in the list of identifiers.
3. Repeat from the beginning until there are no more objects to serialize.

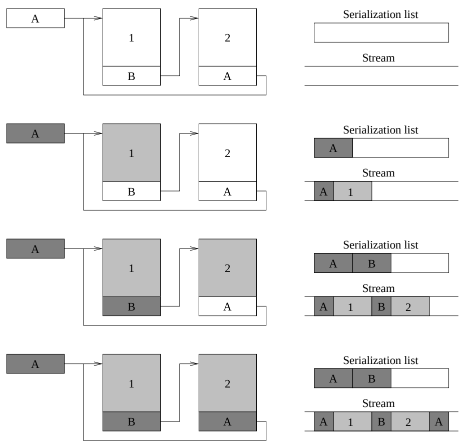

Fig. 1. Example of serialization.

Once all the objects have been serialized into a stream, we have to think about how to retrieve them. Much like the list of identifiers used for serialization, the deserialization process needs to keep another list with the objects that have been already deserialized. When a referenced object is deserialized, a new image of it has to be created in memory. This new copy of the object will be stored at a memory address that may differ from the one that it had when it was serialized. Therefore, the value of an access variable pointing to this new image of the object may also differ from the original one. For this reason, each element of the deserialization list is a pair of access values. One is the value of the access variable that referenced the object when it was serialized (old value) and the other one is the value of the access variable that holds the new copy of the object (new value). The algorithm for deserialization works as follows (see Fig. 2 for an example):

1. Deserialize the elements that compose the object using default Ada deserialization, except for the elements which are access variables.
2. When an access variable is reached, deserialize its value and check if it corresponds to one of the old values stored in the list.
   (a) if the access value is `null`, assign `null` to the new access variable.
   (b) If the access value is not in the list, save it in the list as an old value. Create a copy of the new object with an allocator and deserialize in it the elements of the object. Save this newly created access value in the list as the new value corresponding to the old one.
   (c) If the access value is already in the list, the object has already been deserialized. Assign the corresponding new value to the new access variable.
3. Repeat from the beginning until there are no more objects to deserialize.

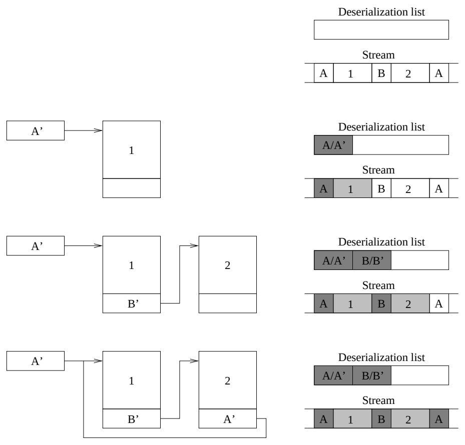

Fig. 2. Example of deserialization.

3 The Implementation

3.1 The Serialization Lists

As we have seen, the algorithm above is only applicable if the dynamic structure to serialize uses just one kind of access type. In the general case, the access variables used by a dynamic structure can be of different access types. We studied different options to overcome this problem. The first three approaches are based on storing the value of the access variables in a common representation format, whereas the fourth approach is based on the use of generics:

1. Convert all access variables to `System.Address` using unchecked conversions and implement the lists with `System.Address` as the type of their elements.
2. Convert all access variables to `System.Address` using the package `System.Address_To_Access_Conversions` instead of using unchecked conversions. Then proceed as in the first case.
3. Create a kind of stream that keeps the value of an object in memory. Transform all access variables to their stream representation using this special kind of stream. If two access variables are equal, they will have the same stream representation. We can then build our lists with these memory streams as elements.
4. Use a different list for each different access type. Lists are generic and they are instantiated with each particular access type, that becomes the type of the elements in the instantiated list.

The first three approaches are necessary if we want to use only one list for all access types. In Ada, there is no global access type that can reference any object[2]. This is consistent with the strong typing philosophy of Ada. We need thus a common internal representation for the values of different access types. This conversion to a common representation comes with a price. The first three approaches make some assumptions that are not necessarily true for all implementations of Ada.

The first approach reduces every access type to a `System.Address` by means of unchecked conversions. We are assuming here that all access types can be represented by a `System.Address`, but this is not true in general, since the representation of an access type in Ada is implementation defined. For instance, we tested this implementation in GNAT and it worked for most access types. However, access types that reference occurrences of objects of unconstrained types have a different representation in memory: they occupy double the size of a `System.Address`. GNAT uses the first address to reference the object and the second to point to the bounds of the object. In this case, a solution based on unchecked conversions will still work for serialization, since the address of the object is enough to determine it and it is saved in the serialization list. However, for deserializing them, we are losing the information about bounds in the list of the new addresses. This example is enough to realize that this approach is not a general solution.

We could use, alternatively, the package `System.Address_To_Access_Conversions`. This option suffers from the same problem as the previous one: it relies on the compatibility of addresses and access types. The standard recognizes this problem saying that an implementation may place restrictions on instantiations of this package (see RM 13.7.2 (6)). Moreover, this approach introduces other inconveniences. Since the type `Object_Pointer` of the package is a general access type, only explicit conversion to general access types is allowed. When working with the serialization lists, several conversions between the type `Object_Pointer` and the access types defined by the

[2] In C, for instance, a pointer of the type `void*` can point to any object.

user, which can be pool-specific, are required. We would force then the user to always work with general access types which is a big disadvantage.

The third approach is independent of the size in memory of an access variable. The value of an access variable is transformed into a special kind of stream which is stored in memory. Streams built up in this way can be compared and used for our algorithm. The problem with this approach (which also affects the first two approaches) is that it assumes that the stream representation of two different access values will also be different. This is not necessarily true, since an implementation could use the same underlying representation for two access variables of different access types. This is allowed by the standard, since variables of different access types cannot be compared. In any case, we have not detected such behaviour when using the GNAT compiler.

The fourth approach seems to be the cleanest solution. It uses different serialization lists for each different access type. The idea is that serialization lists are implemented as generic lists. We will have one reading-writing pair for each access type used by the dynamic structure to serialize. This approach does not rely on any type conversion or internal representation of access types like the other three. Besides, having different lists for different access types accelerates the search of elements on them. In order to check if an access value of a certain type is in a list, we will look only in the list instantiated by that particular access type. In the previous approaches, all values were stored in the same list. The disadvantages of this solution are the large number of files generated and the added complexity in the creation and deletion of the lists (see Sect. 3.8).

3.2 Selecting the Tool

For implementing the serialization algorithm presented above, we need to know the structure of the data type to serialize: which are its components and which of them are access variables. For being able to know the internal representation of a data type, we need a tool that can recognize it. We explored three different options:

- Build a Ada parser on our own.
- Use an Ada compiler.
- Use the Ada Semantic Interface Specification (ASIS).

Clearly, building a custom Ada parser would be a rather inconvenient solution. It would demand a great amount of work just for reinventing the wheel. Since Ada parsers are already found in compilers and ASIS allows the analysis of a compilation environment, this option was soon abandoned for impractical.

3.3 GNAT, a Free Ada Compiler

The next option that we studied was to modify GNAT in order to perform serialization in the way described by our algorithm. GNAT is, up to our knowledge, the only free implementation of a complete Ada compiler. Its distribution license (a slightly modified GPL) allows the use, study, modification and redistribution of its source code. These properties make it a specially valuable tool for academic projects such as this one. In the case of modifying the GNAT compiler, we still have to decide between two possibilities:

1. Modify the part of the expander dealing with stream attributes so they would use the new access type serialization.
2. Add new stream attributes that would serialize access types in this special way and keep the original ones unmodified.

The modification of the compiler requires an understanding of its internals which was outside the scope of this project. Besides, a solution based on the modification of one compiler would be incompatible with any other Ada compiler. Modifying the compiler was, therefore, not further studied although it could be the subject of future projects. We finally used ASIS to validate our serialization solution. Nevertheless, the inclusion of the solution into the Ada standard and, consequently, its integration into Ada compilers is regarded as our final goal. Code generation is an important part of the project and this is a natural task of a compiler expander. A compiler implementation of the solution would also provide a tighter integration into the compilation chain than the implementation with ASIS as we will see below.

We have to notice as well that the modification of the compiler would also imply a modification in the run-time library, which would be in charge of the creation and management of the serialization lists.

3.4 ASIS

ASIS is a procedural interface for communicating with an Ada compilation environment. ASIS provides a set of syntactic and semantic queries that allow an Ada tool to extract information about the structure of an Ada program from its environment.

In GNAT, ASIS works with the abstract syntax tree of the compiled program in the form of a file with extension **.adt*. We note that, in order to use ASIS, the program must be compiled first. We have to figure out how to compile the program without the serialization routines, then generate the routines and, finally, produce the executable program with serialization included. Note that, if we modify the compiler instead of using ASIS, this steps are avoided. Thanks to Ada's abilities for separate compilation, these steps can be carried out. The process is described graphically in Fig. 3.

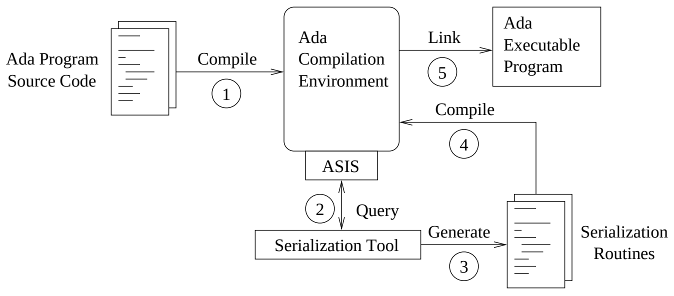

Fig. 3. Compilation chain.

3.5 Child Packages vs Subunits

Once it was decided that we were going to use ASIS as the primary tool for analyzing the program code, we had to think about how we were going to provide the generated serialization subprograms.

The subprograms have to know the internals and the structure of the data type in order to serialize it. For this implementation, we limited ourselves to data types declared in package specifications. The reasons for this choice have to do with the visibility rules of Ada. Usually, dynamic structures are declared as private types, so the complete type definition is located in the private part of the package. A private part of a package specifications is only visible from the body of the package or from its child units. Therefore, the serialization subprograms should be placed in these regions, where the internals of the data type are visible.

In order to apply the separate compilation concepts required by our ASIS-based implementation (see Sect. 3.4), we cannot insert directly the serialization subprograms in the body of the package that defines the data structure. Besides, we wanted to generate the serialization subprograms in separate files from those of the user. We wanted our solution to be as litle intrusive as possible. The first solution that we thought of was to use child units for implementing the serialization routines. The second approach is based on the use of subunits[3]. Subunits can be written in separate files and be compiled separately, but they have the same effect as inserting code in the body of a package. Both approaches have their advantages and their drawbacks.

3.6 Using Child Packages

The first strategy that we followed was to generate the serialization subprograms in child packages. As we said before, this option was interesting because even the private declarations of a package are visible from its children. Within this approach, one child package is generated for the type of interest. Then, one additional child package is generated for each composit type or access type found in the definition of the type of interest. These child packages implement the operations `Read`, `Write`, `Input` and `Output` of their corresponding type.

However, the serialization subprograms generated in this way cannot be primitive operations of the types to serialize. The subprograms are declared outside the package specification where the type of interest is declared. At this point, the type is already frozen and no primitive operations can be added. The serialization subprograms generated in child packages cannot thus be used to override the default serialization attributes. The user has to call the subprograms in the child package as any other ordinary operation.

The use of child packages has other implications as well. Since the serialization subprograms are not primitive operations of their types, all the object-oriented features (namely polymorphism) are lost. There is no equivalent to class-wide streaming attributes with this approach.

Data types declared in generic packages are also affected by this approach. The tool is unable to analyze the structure of formal types, since their actual definition is not

[3] Idea first proposed by Jörg Kienzle (`joerg.kienzle@mcgill.ca`).

available until the generic is instantiated. The only thing we can do in this case is to apply the default serialization attributes to the formal type. This happens even in the case when the actual type which instantiates the generic has custom serialization subprograms generated. We cannot call the procedures of an actual type from the generic, since only the formal type is known at this point.

These are the steps that a user should follow to create an Ada executable by using this approach:

1. Write the package where the data structure is defined.
2. Write, manually or automatically, a child package specification with the subprogram specifications for the serialization operations for the data structure. ASIS is not needed for this part.
3. Write the rest of the program source code. When serialization of the data structure is required, use the subprograms specified in the child package.
4. Compile everything to produce an environment for ASIS.
5. Use our ASIS tool to analyze the data structure generate the body of the child package. As explained, additional packages will be created if the data structure contains access types or composite types.
6. Compile the code generated and link it to the rest of the compilation environment to produce the executable program.

The use of child packages has several disadvantages derived from the fact that they cannot use the standard streaming mechanism (the serialization attributes) because subprograms declared in a child package are not primitive operations of their controlling type. In addition, we have to provide serialization subprograms not only for access types, but for every composite type (record or array). A composite type could hold elements whose type is an access type. Since we are not using the default serialization mechanism, a call to a serialization attribute of the composite type would imply the default serialization of the access types it contains and that is not what we want. Therefore, we have to generate the serialization subprograms for composite types too and serialize them element by element.

3.7 Using Subunits

A solution based on subunits eliminates the problems related to child units. We decided thus to abandon the strategy of generating child packages in favor of the production of subunits. By using subunits, we can write the serialization subprograms of a data type in separate files, but with the same effect as if we wrote them in the body of the package where the data type is declared. Thus, serialization subprograms can be primitive operations of the data type if their specification is placed in the same package specification. For the same reason, serialization attributes can be overridden by the generated serialization subprograms.

The only inconvenience of using subunits is that this solution requires more intervention from the programmer. Indeed, it is the programmer who will have to insert the headers of the serialization subprograms in the package specification and use attribute definition clauses to override the default serialization attributes. Although this can be done automatically, it implies the modification of the source code of program so we preferred to leave the responsability to the user (better not to mix hand made code with automatic generated code). In any case, that is a small nuisance compared to the advantages over the child package approach.

With subunits, only the serialization subprograms for access types have to be generated. Composite types can still use default serialization. For instance, if a record contains a component whose type is an access type, the default serialization mechanism of the record will call the correct serialization subprogram for the access component; that is, the one generated by our tool. This was not the case in the child package approach.

The problem that we had with generics in the previous solution also disappears. Usual serialization attributes can now be applied to formal types. If the actual type used for instantiating the generic has its serialization attributes overridden, the appropriate serialization subprograms will be called. We also keep the object-oriented properties of class-wide serialization attributes.

The steps that the user has to take in this case are the following:

1. Write the package where the data structure is defined.
2. Add to the package the specifications of the serialization subprograms for the access types used in the data structure and override the default serialization attributes by using attribute definition clauses.
3. Write the rest of the program. When serialization is needed, use serialization attributes as usual.
4. Compile everything to create an environment for ASIS.
5. Use our ASIS tool to generate the code for the subunits. One subunit will be created for each access type.
6. Compile the code generated and link it to the object files from the rest of the program to produce the executable.

3.8 Serialization Lists Management

As we have seen, the serialization lists keep track of the objects that have already been serialized for avoiding infinite loops during the serialization process. Every time serialization or deserialization of a dynamic structure has been completed, we have to decide what to do with the lists.

In Java, these lists are associated with a stream object. Objects are serialized by calling the methods of this stream object. Whilst the stream object is not closed, the content of the lists is mantained. As a consequence, if one object is serialized several times without closing the stream, its state will only be stored in the stream the first time. All the modifications made to the state of the object after the first serialization will not be reflected in the stream. To solve this problem, the stream object has to be closed and then reopened, so the lists will be empty and the whole state of the object to serialize will be saved [2].

For our implementation, we decided to give the user the possibility to decide between automatic or manual management of the serialization lists[4].

[4] The solutions implemented up to now are not thread safe, since it relies on a global package which assumes that only one serialization is taking place at any given time.

1. **Automatic:** The lists are erased each time serialization of a dynamic structure is completed. Since we only modify the attributes of access types, the variable that represents the dynamic structure should be an access variable. Otherwise, our algorithm does not work properly.
2. **Manual:** The program will call the procedure `Start_Writing` of the `Serialization_Control` package each time it needs to serialize one or several objects. Once the serialization is finished, the program will call procedure `End_Writing` in the same package. Procedures `Start_Reading` and `End_Reading` are also provided in the package for their use during deserialization. If the objects share references, manual serialization has some advantages over automatic serialization.

3.9 General Access Types

Along this document, we assumed that all access types used in our dynamic structures were pool-specific access types. General access types can be serialized using our mechanism as well but, in the case of a general access pointing to an aliased object, the state after deserialization is not exactly the same as before serialization. The reconstruction of such an object performed by deserialization requires its new creation in a storage pool, even if it did not belong previously to a pool.

It is also possible to have duplicates of an object, since it can be aliased and contain an access to itself. If we serialize the object following our algorithm, we will write its elements one by one until we reach an access to itself. The value of this access type will not be in the serialization list, so the algorithm will erroneously assume that the object was not yet serialized and it will serialize it again. During deserialization, two copies of the object will be created in memory.

This problem can also affect pool-specific objects. Let us suppose that, in the example shown in Fig. 1, we start serializing the object instead of the access to the object. The result would be a duplicate of the first object in the stream.

4 Conclusion

During this project we worked a lot with Ada streams. As a consequence of this work, we missed one declaration of a general access to stream type in the package `Ada.Streams` as explained below.

Streaming attributes have an access to the class wide type `Ada.Streams.Root_Stream_Type'Class` as their controlling parameter. However, there is no access to this type defined in the package `Ada.Streams`. It seems that the standard delegates this responsability to concrete implementations of the streams package. For instance, `Ada.Streams.Stream_IO` (the only standardised concrete implementation of streams) does declare the type `Stream_Access`. In our opinion, this declaration should be placed in the parent package `Ada.Streams`, since it is an access to the whole class of streams and it does not make sense that each concrete implementation of streams should define its own. We could not find an explanation to this design decision in the annotated version of the Ada reference manual [4].

4.1 Future Work

We have presented the implementation of a mechanism for the automatic serialization of dynamic structures in Ada based on the use of ASIS. The ultimate goal of this project would be, however, to produce a modification to the Ada standard in order to include this serialization mechanism in the language. We would propose to add two new serialization attributes named `Dynamic_Input` and `Dynamic_Output` for serializing objects as described by this document. If this idea is well received inside the Ada community, a formal AI will be stated and proposed. It is also foreseen to provide a reference implementation of the new attributes by modifying GNAT. The GNAT implementation would also address some improvements of list management.

We will have to study as well the implications of this new serialization method for the Distributed System Annex (DSA). The DSA uses Ada streams for serializing the parameters of remote procedure calls. It would be interesting to use our new serialization mechanism for exchanging complete dynamic structures among active partitions.

This project also opens the door to the exploration of new ways of object serialization in Ada. Although Ada allows to create different kinds of streams for different kinds of media, the representation of variables in the stream is always the same, although it can be changed for user defined types by overriding stream attributes. It would be interesting to have the possibility of using standardised attributes for generating streams in a common representation format such as XML, even for predefined data types. XML serialization would be a great supporting mechanism for introducing Ada into the emerging world of Web services.

References

1. S. T. Taft, R. A. Duff, R. L. Brukardt, and E. Ploedereder, Eds., *Consolidated Ada Reference Manual: Language and Standard Libraries. International Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2001, vol. 2219, ISBN 3-540-43038-5.
2. (2001) Java object serialization specification. Sun Microsystems. Santa Clara, CA. [Online]. Available: ftp://ftp.java.sun.com/docs/j2se1.4/serial-spec.pdf
3. (2001) Serializing objects. Microsoft Corporation. Redmond, WA. [Online]. Available: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconbinaryserialization.asp
4. *Annotated Ada 95 Reference Manual*, ISO/IEC Std. 8652:1995(E), 1995.
5. J. Kienzle, A. Romanovsky, and A. Strohmeier, “A framework based on design patterns for providing persistence in object-oriented programming languages,” EPFL, Tech. Rep. DI/2000/335, 2000.
6. J. Kienzle and A. Romanovsky, “On persistent and reliable streaming in ada,” in *International Conference on Reliable Software Technologies - Ada-Europe’2000, Potsdam, Germany, June 26-30, 2000*, H. B. Keller and E. Plöderer, Eds., no. 1845, 2000, pp. 82–95.
7. S. Crawley and M. Oudshoorn, “Orthogonal persistence and ada,” in *TRI-Ada*, 1994, pp. 298–308.