Distributed Ada[1]: Methodology, Notation, and Tools

Greg Eisenhauer
Rakesh Jha
J. Micheal Kamrad, II

Honeywell Systems and Research Center

Abstract

The task of creating software to run on a distributed system brings with it many problems not encountered in a uni-processor environment. The designer, in addition to creating a solution to meet the functional requirements of the application, must determine how to distribute that functionality in order to meet the non-functional requirements such as performance and fault tolerance. In the traditional approach to building distributed software systems, decisions of how to partition the software must be made early in the design process so that a separate program can be written for each of the processors in the system. This design paradigm is extremely vulnerable to changes in the target hardware environment, as well as being sensitive to poor initial guesses about what distribution of functionality will satisfy the non-functional requirements. The paradigm is also weak in that no compiler has a complete view of the system. Many of the advantages of using a powerful language system are lost in a one-program-per-processor environment. This paper will present another approach to the development of distributed software systems, Honeywell's Distributed Ada program.

Our Approach

The goal of Distributed Ada is to develop methodology and tools which will significantly reduce the software design complexity for reliable distributed systems. We believe that the functional specification of a system (what it will do) can and should be separated from its non-functional specification (how it will be mapped onto the underlying system). The functional specification can be developed and expressed in Ada. To this is added the specification of the non-functional attributes of the system. Separating the problem space into two smaller problems means that the designer can concentrate on solving each of them in turn rather than attacking them together. It also allows software development to proceed before hardware final design is complete and enhances the portability of the functional specification.

[1]Ada is a registered trademark of the U.S. Government Ada Joint Program Office.

The software development paradigm we advance is described by the following scenario. The designer develops a functional solution to the problem in Ada using uni-processor development tools. With a functional solution in hand, she then creates a specification of the non-functional characteristics of the solution (more details on the nature of this specification will be given later). Using the tools being developed under our program, these two specifications can be used to create the distributed solution incorporating the non-functional attributes. At this point, the distributed solution can be tested for acceptability according to non-functional criteria and modified if necessary to meet non-functional requirements.

The advantages of this and similar approaches over the traditional approach of up-front distribution decisions are self-evident. When non-functional specification is separated from functional specification, software development can proceed with limited knowledge of final hardware configuration and will be little impacted by changes in the underlying system. We believe, however, that the granularity of distribution and the mechanism of specification employed in our approach separate our work from that done by other researchers.

As opposed to other projects which limit the unit of distribution to the Ada library unit and limit remote access to tasks and subprograms in the visible part of remote units [Inv 85, Sch 81, Sof 84, Vol 85], we believe that an effective and extensible non-functional specification should allow distribution of all subprograms, packages, tasks and objects in the Ada specification. A narrower stand on the objects of distribution requires the designer to be more conscious of the non-functional requirements while searching for the functional solution. While it can be argued that a designer who is aware of all the requirements of an application will produce a more efficient solution, we believe that the tools he uses to produce the distributed solution should impose as few constraints as is possible. Constraints imposed at this level directly impact portability and robustness of a given functional solution in the face of a changing hardware environment.

Many researchers argue that the PRAGMA construct in Ada should be used for non-functional specifications such as distribution of entities [Inv 85, Vol 85]. We have chosen another approach for several reasons. One concern is that an approach involving PRAGMAs will not be extensible to specification of non-functional attributes such as dynamic relocation of objects or fault tolerance strategies. Pragma-based schemes for specifying distribution are complicated already, attempting to extend these schemes to additional domains might prove unwieldy. We also consider it a disadvantage that the pragmas would be embedded in the source and scattered throughout the Ada specification. This

makes sharing of library units between applications difficult or impossible. It also impedes manipulation of the specification of distribution. If this specification were concentrated in one location rather than dispersed throughout the code, it would be easier to form a global picture of system distribution. We also observed that the function to be performed by these notations was to establish a structuring hierarchy distinct from that of Ada. This led us to create a separate specification notation, the Ada Program Partitioning Language (APPL) [Cor 84, Hon 85, Jha 86].

Ada Program Partitioning Language (APPL)

The goal of the APPL design process was to produce a compact, convenient notation for specifying the non-functional attributes of a program. APPL addresses issues of distribution of Ada entities, and dynamic relocation and replication of those entities. Extensions to APPL to cover fault tolerance specification are under consideration. For brevity, this discussion will consider only APPL in general and static distribution in specific. The reader is referred to the APPL Reference Manual for a more detailed and formal description.

It is useful at this point to introduce some terms.

> A FRAGMENT is a user-specified collection of entities, such as packages, subprograms, tasks and objects, from the Ada source program. Every entity belongs to one and only one fragment. Membership in a fragment is attained either implicitly, as a result of default rules, or explicitly, as a result of inclusion in an APPL fragment declaration.
>
> A STATION designates a computational resource in the underlying system. Typically, this is a node in a distributed system.
>
> MAPPING a fragment to a station causes all entities in that fragment to reside on that station at runtime.
>
> A PROGRAM CONFIGURATION refers to a specific partitioning of a program into a collection of fragments, and the specific mapping of the resulting fragments onto stations.

An APPL specification completes a Program Configuration and consists of two parts. The first of these, the configuration specification, specifies the fragmentation of the Ada program, while the latter, the configuration body, specifies the mapping of fragments to stations.

The configuration specification provides a mechanism for specifying Ada entities to be bundled together as a fragment. With a few exceptions, such as within unnamed blocks, these entities can be selected from within any declarative region in the program. As a convenience, APPL semantics implicitly declare a fragment for every library unit which make up a program. It also provides a mechanism for further bundling fragments into

fragment groups. Fragment groups, like fragments, are mutually exclusive and are treated like fragments in mapping.

The configuration body is a simple section specifying a correspondence between fragments and stations.

As an example to illustrate the use of APPL, consider the following Ada text.

```
with TEXT_IO, REAL_OPERATIONS; use REAL_OPERATIONS;
package EQUATION_SOLVER is
   procedure QUADRATIC_EQUATION;
   procedure LINEAR_EQUATION;
end;

package body EQUATION_SOLVER is
      .
      .
      .
end EQUATION_SOLVER;

with EQUATION_SOLVER;
procedure MAIN is
begin
      .
      .
end MAIN;
```

Also consider the following configuration specification.

```
with MAIN, EQUATION_SOLVER, REAL_OPERATIONS;
configuration PROTOTYPE is
   fragment QUAD_EQUATION is
      use EQUATION_SOLVER;
      procedure QUADRATIC_EQUATION;
   end QUAD_EQUATION;
end PROTYPE;
```

Recall that APPL implicitly declares a fragment for each library unit involved. Thus the implicitly declared fragments are: MAIN, EQUATION_SOLVER, TEXT_IO, and REAL_OPERATIONS. QUAD_EQUATION is an explicitly declared fragment containing the procedure QUADRATIC_EQUATION from the library unit EQUATION_SOLVER. An example configuration body is shown below.

```
configuration body PROTOTYPE is
   map EQUATION_SOLVER, MAIN, TEXT_IO onto STATION_1;
   map QUAD_EQUATION onto STATION_2;
   map REAL_OPERATIONS onto STATION_3;
end PROTOTYPE;
```

An APPL specification, together with the Ada source, constitute a description of a distributed software system. It is the function of the tools we are developing to actually produce this system.

Distributed Ada Tools

In order to avoid spending a large amount of development time on issues not strictly related to distributed systems, we have chosen the approach of modifying an existing Ada language system rather than creating one from scratch. Two major tools in any Ada language system are the compiler, which maintains the Ada program library and produces object code for strings of compilation units, and the linker, which must determine and generate code for library unit elaboration and actually assemble the final executable image. In the compilation environment, these tools are the most drastically affected by retargeting to a distributed environment.

Modifications to the compiler are perhaps the most dramatic. Obviously, the compiler must be made aware of the fragmentation and mapping specified by APPL. Therefore, the first phase of distributed compilation consists of modifying the intermediate representation (DIANA, for our purposes) of the Ada library units and their secondary units by adding a "fragment" attribute to the DIANA nodes. This allows the compiler to determine the station of residence for that entity.

From the modified DIANA representation of a compilation unit, a linearizer generates intermediate language (IL) code for the compilation unit. This linearizer, in particular, must be significantly more complex than it is required to be in a uni-processor compiler. It must now produce an IL code module for each of the stations to which fragments of the compilation unit have been mapped. Of even more significance, is the fact that it must generate proper code to reference entities on remote stations. This task is simplified somewhat, because the problems associated with distributed Ada tasking will be dealt with by the runtime environment (discussed in the next section) and will be invisible to the compiler. However, most every other aspect of IL generation is affected. Fetches and stores to remote variables, for example, will require calls to special runtime primitives for remote data access.

As another example of the issues involved in this stage, consider the problem of parameters to a remote subroutine call. In a uni-processor system, it is efficient to pass large parameters by reference rather than by value. A pass-by-reference mechanism could be employed in remote subroutine calls by adding a station address to the parameter address. But this would mean that every reference to the parameters of a procedure that could be called remotely would involve the remote data access mechanisms. Since parameters are likely to be heavily utilized in computation, this appears to be an undesirable situation. Our solution to this problem involves the generation of local 'stubs' whose purpose is to package the parameter values and transmit them to the remote system. The runtime environment on the remote station will disassemble the package and call the procedure in question. Since this call looks just like any purely local call, the code generated for the called procedure is unchanged. (Note: This mechanism cannot be applied to access types. They must be handled by a reference mechanism similar to that mentioned above.)

Once these multiple intermediate code modules have been produced, object code generation on each of them should continue in a fairly normal manner and the final object files can be passed on to the linker. In the Distributed Ada environment, our scenario involves the production of multiple executable images, one for each station in the system. This will require modifications to the linker, which will have to resolve symbols between multiple executable images, something which no uni-processor linker would ever have to do. Fortunately, these linker modifications are not conceptually difficult and represent only an engineering problem.

Runtime Environment

The execution environment considered consists of a network of stations and a copy of the runtime system on every station. The runtime system makes the underlying hardware appear to the distributed application as an Ada virtual machine.

There is a minimum set of facilities that must be provided by the distributed runtime system, independent of the granularity with which an Ada program is partitioned. It must provide reliable inter-station communication and synchronization, a consistent view of distributed state information at each station, a globally consistent view of time, and means to deal with partial failures in the underlying system.

The overall complexity of the distributed runtime system depends on the support it provides for binding the application fragments together dynamically, for making the application fault-tolerant by masking station and network failures from it, and for representation conversion between heterogeneous stations.

There is a spectrum of possible binding times. If binding is done statically before execution time, it is not possible to reconfigure an application during execution by remapping one or more of its fragments. Dynamic binding is the most flexible. The mechanism can be used effectively by the runtime system to reconfigure an application as a means of providing fault-tolerance, or of changing the configuration as resource requirements change during execution. If the underlying system is heterogeneous, the binder must also insert representation conversion filters for values that are passed between the remote fragments that it binds together.

The complexity of the runtime system is only marginally affected by the choice of the set of Ada entities that can be distributed. The apparent similarity between concurrency in Ada tasks and concurrency of execution on a network of processors may initially suggest that tasks be made the unit of distribution. However, a close examination will quickly show that this restriction does not really simplify the runtime support needed.

The allowed granularity of partitioning has a greater impact. For the sake of an example, consider the case where Ada library units are the unit of distribution. The runtime system must support calling of remote subprograms, reading and writing remote data, and tasking operations on remotely located tasks. Since Ada task dependencies do not cross library unit boundaries, the semantics for task termination can be implemented in a manner that gainfully uses the knowledge that the task dependencies cannot cross station boundaries. This simplification is not available if a finer granularity of partitioning is allowed. In application areas where the size and efficiency of the runtime system are critical, we think that the specific requirements of the application domain should be taken into consideration when deciding the granularity of program partitioning.

Project Status and Plans

Honeywell's Distributed Ada project was started in 1982. A preliminary version of APPL was defined in 1983 [Cor 84]. A prototype implementation based on source-to-source transformation and an unaltered uni-processor compiler was built during the following year. In 1985, the structure of APPL was changed and the language revised and formalized [Hon 85, Jha 86]. Current development focuses on creating the specialized tools and runtime environment described above. In order to manage the implementation, we have divided its development into several stages. Phase 1, which we are currently working under, calls for a fully functional system, limited to homogeneous systems and static distribution of objects declared in the visible portion of library units (and the units themselves). We hope to complete this phase of development by the end of 1986. We are

using the VERDIX Ada Development System (VADS[2]) as the baseline compiler from which to create the Distributed Ada system and are operating in a simulated network environment using processes under Unix[3]. Future development phases call for support for heterogeneous systems, dynamic reconfiguration, object replication and fault tolerance.

References

[Cor 84] D. Cornhill, "Partitioning Ada Programs for Execution on Distributed Systems," **IEEE 1984 Proceedings of the International Conference on Data Engineering.**

[Hon 85] "Honeywell Distributed Ada Project," 1985 report.

[Inv 85] P. Inverardi, F. Mazzanti, and C. Montangero, "The use of Ada in the design of distributed systems," **Ada in Use Proceedings of the Ada International Conference**, Paris, 14-16 May, 1985.

[Jha 86] R. Jha, J.M. Kamrad, D. Cornhill, "Ada Program Partitioning Language: A Notation for Distributing Ada Programs," submitted for publication.

[LRM 83] "Reference Manual for the Ada Programming Language," ANSI/MIL-STD-1815A, U.S. Department of Defense, 1983.

[Sch 81] S. Schuman, E.M. Clarke, and C. Nikolau, "Programming distributed applications in Ada: A first approach," **Proceedings of the 1981 International Conference on Parallel Processing.**

[Sof 84] Softech, "Programming distributed applications in Ada," December 1984.

[Vol 85] R.A. Volz, T.N. Mudge, A.W. Naylor, and J.H. Mayer, "Some problems in distributing real-time Ada programs across machines," **Ada in Use Proceedings of the Ada International Conference**, Paris, 14-16 May, 1985.

[2]VADS and VERDIX are registered trademarks of the VERDIX Corporation.

[3]Unix is a registered trademark of AT&T Bell Labs.