DISTRIBUTING PROGRAM ENTITIES IN Ada[1]

Patrick Rogers
Charles W. McKay
High Technologies Laboratory
University of Houston
at Clear Lake

Introduction

In any discussion of distributing programs and entities of programs written in a high order language (HOL), certain issues need to be included because they are generally independent of the particular language involved and have a direct impact on the feasibility of distribution. Of special interest is the distribution of Ada program entities, but many of the issues involved are not specific to Ada and would require resolution whether written in Pascal, PL/1, Concurrent Pascal, HAL/S, or any language which provides similar functionality. The following sections will enumerate some of these issues, and will show in what ways they relate to Ada. Also, some (but by no means all) of the issues involved in the distribution of Ada programs and program entities will be discussed.

Justification

Before introducing such a subject, it is perhaps reasonable to provide a rationale for distributing any named resource of a HOL program in the first place. The reasons are straight-forward.

First, and probably most important, is the issue of reliability. Computers are increasingly used in applications which require high reliability, because they impact life and property (sometimes literally). Embedded applications which provide life support, control guidance and navigation, or manage weapons are examples. A failure of such an application can be disastrous. By decentralizing the software (and of course, the hardware), we can provide systems that not only do not have single points of failure, but that are fault-tolerant. Such systems can recover from

---

[1] Ada is a registered trademark of the U.S. Government (AJPO)

failures once they are detected. (This approach should not be confused with fault-avoidance, which attempts to prevent failures from impacting the system in the first place.)

The second reason is that of the decreasing cost of hardware, especially with respect to the ever-increasing cost of software. In order to make the most, economically, of the power of software, utilization of multiple processing resources is desirable. Parallel processing is an example.

The third reason is extensibility, in the domains of performance and functionality. When the software system is designed with distribution as a design criteria, the resulting modularity provides a design that does not necessarily have to be radically changed for increases in processing power (for performance) or for the addition of new modules (for additional functionality). In a system intended to have a long, evolving life cycle, this is a major issue.

Fourth, given limited resources of operational costs, hardware, communications, and information, when those resources are themselves distributed (as in Space Station), resource sharing implies that only those elements that require direct access and are to be held accountable for the integrity of the resource should be located in proximity to that resource. In this case, distribution of the software allows only that part which interacts with the resource to be present (with potential benefits of reduced communications costs and localization of accountability).

The fifth reason is the issue of the fidelity of modelling solutions to real world problems that are distributed in nature. Such problems are complex enough without adding additional complexity by distorting the solution model to fit a non-distributed HOL with no support for cooperating, parallel activities, or for recognizing both exceptions to normal processing and the context in which the exceptions occur (so that appropriate fault tolerance and fail-soft activities can be supported). For example, the Space Station Program will eventually involve ground support stations, free-flying platforms, the Station, orbital transfer vehicles, and other components. These components are intended to interact in an integrated, end-to-end information environment. (Put simply, any authorized user at any component of the environment who desires to access entities should be given timely access to such entities without regard for the location, replication, number of processors supporting the access, or means of providing fault tolerance.) Obviously, a model of the solution to these challenges involves a high degree of distributed parallel processing activities which must evolve

in a cost-effective, adaptable, and safe fashion.

Finally, the issue of performance should be addressed. It, too, is straight-forward. When the application demands the advantages and benefits of distribution, the price of decreased efficiency must be paid. It should be understood, however, that distribution will not automatically mean poor performance. In fact, distribution will in some cases improve performance by decreasing communication costs, taking advantage of remote hardware resources, and so on.

The above reasons should be sufficient for illustrating the need for distributed software. The general issues involved in distribution will follow.

Visibility

One of the primary underlying concepts in distributing a HOL program is that of "visibility". In this context, visibility means "the set of objects which may be potentially referenced at any particular point in a program". These objects include both data and code modules, such as variables and subroutines. Depending on the distribution scheme, these objects may or may not be locally available. In those instances where the object is remote, the Run Time Support Environment (RTSE) will be required to help fulfill the semantic requirements of a given reference. For example, the program may have some of its variables distributed across remote sites. A reference to such a remote object will require cooperation among the two RTSEs. The calling RTSE will have to contact the RTSE of the processing site at which the variable is located, with a request for the current value of the variable. The remote (called) RTSE must locate the variable, get its value, and send back a message containing that value. (The recovery of a failure of one of these messages is non-trivial.)

As can be seen, the visibility of objects plays a considerable part in determining the complexity of the RTSEs involved.

Distribution Scheme

A distribution scheme may often be described in terms of the visibility rules of the implementation language. Traditional block-structured languages, such as ALGOL and Pascal, use nesting to control visibility of locally declared data and subroutines. The visibility rules of these languages are such that the inner declarations of subroutines and data are visible to further nested units in

the same declarative region, but not to outer units at the same nesting level. A global section of data is directly visible, and of course outer-level subroutines are visible to successively declared subroutines at the same level, in a linear manner.

As previously shown, the visibility rules directly impact the complexity of the required RTSE by determining the set of entities that may be referenced at a particular point. This complexity represents a major factor in determining the feasibility of a distribution scheme itself. Those schemes which reflect visibility rules that restrict the size of the name space are easier to implement.

The distribution schemes form a spectrum based on the visibility rules and the constructs of the source language involved. For example, if the distribution is to be at the individual statement level, (representing one extreme), then any object referenced may be remote, including components of complex expressions. (The resulting RTSE requirements would be extensive. The instance discussed under "Visibility" above is an example.) If distribution is to be at the compilation-unit level, (the other extreme), then the set of all entities that may be referenced is reduced to globally visible entities, such as subroutines and their parameters. In effect, the distribution scheme controls the size of the distributable name space, and therefore the complexity of the RTSE.

Time

Another important concept is that of time, either expressed in the program directly, or in the underlying RTSE. The basic problem is that in order to provide correct semantic execution, distributed program units require the same effects as a consistent, unified version of time that would be provided in a non-distributed environment.

As an example of directly expressed timing, if one module requests a service of another remote module, with a specified amount of time allowed for the request to be fulfilled, the two modules must have a common view of time for the request to have any meaning. Note that this does not mean that the two modules' clocks are necessarily synchronized, only that they be mutually consistent while the request is being served.

In the underlying RTSE, certain operations and actions often need to be synchronized with respect to each other for correct operation and support of a source program. This will

also be required in a cooperative manner among the RTSEs supporting distributed programs.

Semantic Integrity

A critical concept is that of semantic integrity, which means that the meaning of constructs and program units must be maintained without regard for distribution. For instance, a call to a subroutine must have the same semantic effect, or meaning, regardless of the routine's actual location with respect to the caller. Note that this does not mean that the behavior is the same, especially with respect to temporal performance. (In other words, it has to work the same, but not necessarily with the same timing and space profile.)

A specific aspect of semantic integrity is that the semantics of a given construct are to be invariant over failures of the processors executing the corresponding object code. For example, the semantics of a subroutine call are such that, once the called routine is completed, execution resumes in the calling module. In a distributed context, in which the called routine is remote from the caller, if the called module's processor fails, the calling module will be suspended indefinitely. The semantics would thus be (incorrectly) different in the distributed environment. Semantic integrity, in this case, means that the caller must not be allowed to permanently suspend, since the semantics of a call do not include that situation. (Obviously, if the called routine is designed to never complete, due for example to an infinite loop, then the caller will never resume. However, that is not a result of the semantics of a subroutine call.) Similarly, if the processor(s) executing outer-level units in a nested structure fail, the inner-level units must not be allowed to proceed normally since they depend on the outer-level scopes for their execution context. This is, again, an issue that may be partially addressed by the distribution scheme, by constraining the units that may be distributed to those at the outer-level.

Resource Management

A more obvious issue than those above is the management of resources. These resources include storage, processors, and information (among others, such as devices). Specifically, storage management involves dynamic, static and temporary data, as well as the management of code (which may also be dynamic).

Processor management involves dispatching potentially remote processors to processes, as well as scheduling, which determines the units that are to be able to execute at a given moment. Both are, of course, requirements of the RTSE.

Information management involves the maintenance of consistent, current status information regarding individual modules' contexts, processing status and workloads, the global program state for each executing program, descriptive information about data and code, and so on.

Different languages have varying degrees of resource management requirements, as well as varying degrees of programmer-level control over them. Thus the amount of RTSE support required varies. For instance, languages which allow the allocation and deallocation of dynamic objects from a heap will require different RTSE support from those languages which have no such capabilities (often intentionally, such as in HAL/S). Some languages have only static data, and thus require different storage management techniques that those which are stack-oriented. In a distributed context, where heaps may be effectively distributed and/or shared, the management of dynamic objects will require specialized RTSE capabilites.

ISA Homogeneity

The Instruction Set Architectures (ISA) of the processors that comprise the target environment are also an issue. If these processors are potentially heterogeneous, target dependencies become a problem. One such dependency is of course implicit in the object code itself, since the machine code was generated for a particular ISA. Also, the source code may contain explicit target dependencies. These could include references to absolute addresses and specific devices, as well as specific data representation requests, and so on.

Furthermore, the default representation of data may vary among ISA's with different capabilities. This difference in representation will be a problem when objects are visible to (two or more) remote modules on non-homogeneous ISAs, as well as when objects are passed as parameters between such modules.

Changes In Situ

In systems which are intended to have a very long, evolving life-span, such as Space Station, changes to the software are inevitable. These changes will occur as a

result of upgrades in technology, and as a result of changing requirements in functionality. The design of the software must, in its initial form, provide for such changes. (Alterations to the design after-the-fact present a much more difficult situation.) Currently accepted complexity-control methods of modularity and information hiding, along with the requirement for changing a system without first halting that system, dictate that separate programs be employed in the construction of the software. Each program is to be distributed as necessary, or not at all. This approach is in contrast to one in which a single, monolithic program is distributed across the network(s).

Issues in Distributing Ada Programs & Program Entities

Justification for Selecting Ada

Provably Correct Constructs

Older HOLs were designed in an era of single monolithic processors that were typically expected to execute programs that were small (by current standards), and that were developed by one programmer. The three oldest high order languages, FORTRAN, LISP, and COBOL, were developed (in 1957, 1958, and 1959, respectively) before the development and wide recognition of the concepts of building "structured" software from a small set of provably correct constructs. Thus it is understandable that natural reenforcement for consistent use of such constructs is lacking. In fact, those who use early languages in building solution models for many of today's complex problems often find themselves penalized for such use. In contrast, the Ada language provides direct support for developing solutions to large, complex problems that are demonstrably correct, maintainable and adaptable.

Support for Parallel Activities with Fault Tolerance

These early languages are called sequential because they have no support for modelling concurrent or parallel actions. Additionally, they provide support for normal processing only, with no means for expressing the response to run-time errors. Again, The Ada language provides direct support for such activities. To distort the solution model with such a language as FORTRAN or Pascal would require extensive programming in assembly language and use of operating system calls in order to compensate for the inadequacies of the language. The resulting software system

would be too expensive to build, much more difficult to maintain and operate, and far more difficult to adapt to changing requirements. Similarly, to distort the solution model by failing to support distributed program entities, as well as distributed programs (when appropriate), would be to add rather that to reduce complexity, since the resulting model would be far less representative of the problem.

Distribution Scheme

The central theme in the following discussion is that of the distribution scheme. As demonstrated, its control over visibility has a considerable impact on the complexity of the underlying RTSE, and thus the feasibility of distribution. In Ada, the spectrum of distribution begins with constants and variables, continues to nested program units (blocks, subprograms, packages and tasks), and ends at the other extreme of compilation units. (It should be noted that Ada provides greater control over the name space via packages.) Compilations units in this case would be Ada's "library units": specifically, subprograms and packages. At this level, the only visible entities are these library units, parameters for these units when they are subprograms, and declarations in the visible parts of library unit packages. Distribution at this level is the easiest to support. Distribution at the nested program unit would limit some visibility, (i.e., the declarations local to nested units), but not globally visible data and routines. Thus it would not result in less RTSE complexity. Obviously, the simpler the requirements for the RTSE the better, since the implementation of distribution support is simpler.

However, other factors besides RTSE complexity must be considered in the choice of distribution level support. Specifically, the amount of fault-tolerance required must be seriously considered. If little fault-tolerance is required, the system may be allowed to deal with it transparently (in very deterministic ways), such that the programmer is not directly involved with the response to failures. As such, the programmer has no need to express aspects of distribution dynamically in the source language. However, in some applications only the programmer can know what is to be done in response to failures. The appropriate response may be a specific reconfiguration of the program units involved. Since the only dynamic program unit is the task, the distribution scheme may have to support distribution of tasks in order for the programmer to specify the reconfiguration.

Time

The concept of time in Ada may be expressed explicitly in several ways, based on the delay statement. An example of the need for consistency across remote units is, of course the timed entry call, which requests a service to be provided to the caller in a specific amount of time. If the server is to respond meaningfully, it must perform the request for rendezvous in the amount of time indicated by the call. However, since the clocks of the two processors will not be synchronized, and there will be an indeterminable communication lag, difficulties will exist. Specifically, the server may respond too late, such that the caller will have timed-out and continued on as if the service was never provided. If not handled by the RTSE, the program would then be in a logically inconsistent state.

An example of timing issues in the underlying RTSE is the activation of remote tasks. The parent task must not begin execution until all tasks declared in its declarative region are successfully activated. If one or more of these activations fail, then Tasking_Error must be raised in the parent.[2]

Another example is the elaboration of the library units named in the context clauses of a main (sub)program. These must be elaborated in an order that is consistent with the transitive dependencies. As a result, distributed library units cannot simply be elaborated when the remote host site is ready. Rather, there must be communication and cooperation among the sites.

Semantic Integrity

Ada subprogram calls will exhibit the behavior described under the general section on "Semantic Integrity" with respect to failure of the called unit (i.e., they too will not return). Furthermore, an entry call will exhibit those same characteristics when the processor supporting the called entry fails. Conditional and timed entry calls can protect the caller from permanent suspension prior to the start of the rendezvous. However, these calls do not protect the caller once the rendezvous has begun.

---

[2] Note that in a distributed context, the activation status messages may be lost. The resulting indefinite suspension of the parent would be an example of failed semantic integrity.

It should be noted that in a distributed execution environment, the conditional entry call is not the same as a timed entry call with a zero delay. The reason is as follows. In the Language Reference Manual (LRM)[3], the phrase "immediately possible" in the discussion of the conditional entry call refers to the readiness of the called task to accept the call, (not to an amount of time). The conditional caller is dependent upon the called task to indicate whether or not it can accept the call. If not, the caller will resume under the "else" part of the call. If the called task indicated that it could perform the rendezvous (resulting in the caller being suspended), and then failed, the caller would be indefinitely suspended (unless fault tolerant programming techniques are applied). This is not the case with a timed entry call. Under a timed call, the caller is not dependent on the called task. (The caller does the timing.) If the call is not performed in the specified delay, then the caller continues on, without regard for the status of the called task. Thus, the semantics are not the same.

Resource Management

Distributed Ada will require all the resource management activities outlined in the general section on resource management, and specifically those for a stack-oriented language. One aspect that has received attention is the subject of dynamic data, supported in Ada by the "access type". Some implementations of distributed Ada restrict parameters such that values of access types are not passed between remote program units.[4] This is an expedient approach, but not an absolutely necessary one. In Ada, dynamic objects are referenced as abstractions, which is why they are called "access" types rather than "pointer" types. The value gives "access" to the dynamically allocated object. This is of course typically implemented (on uniprocessors) as an actual address. The common reaction to distributing access types is then that such distribution is not possible. However, in keeping with the abstraction concept, in passing an access value to a remote site, rather than passing an address which will be meaningless to the remote site, a "token" should be passed which uniquely identifies the dynamic object. The identifier will have to

---

[3] Ada Language Reference Manual, ANSI Mil-Std-1815A, Section 9.7.2

[4] A Feasibility Study to Determine the Applicability of Ada and APSE in a Multi-microprocessor Distributed Environment (Final Report, March, 1983) TXT, CISE, SPL

be unique over the entire target environment, and may be passed at will among distributed units.

ISA Homogeneity

Ada programs will have the same problems of data representation that any HOL program would, when the processors comprising the target environment are heterogeneous. These problems will be exhibited when global objects are referenced by two or more remote program units on different ISAs, and when parameters are passed between such program units via subprogram and entry calls. The specific incarnation of the problem is package Standard, which logically encloses the units comprising a program. (Package System is also a problem to a lesser extent.) Package Standard defines type Integer, Float, Character and so on, for an entire program. The question then is how different ISAs can efficiently represent those common types.

One approach is to resort, in all cases, to representing passed data at the level of the common denominator: type String. This is considered too extreme, since not all communicating program units will be on heterogeneous processors. However, the concept of a common format, a "canonical data format", may be the most expedient approach. A promising alternative is the concept of "self-defining data structures", in which the passed data includes a description of its representation.

Changes In Situ

As stated in the general section, changes to the software in a system with an long, evolving life cycle will be required. It may often be the case on Space Station that the subsystem being changed is critical and cannot be stopped in order for the changes to be installed. Also, good design, maintenance aspects, and the sheer volume of software involved mandates that multiple Ada programs be utilized in the construction of the software system. This is not in conflict with the LRM, although a casual reading might imply that the LRM requires only one program to be "in existence" at a time. Nothing in the LRM has been found to require such a restriction.[5]

Each program would be distributed if the requirements dictated that approach. Each would be only as distributed as

---

[5] The issue of multiprogramming is (appropriately) not addressed in the language reference manual.

necessary, to reduce the costs of distribution support. Furthermore, if the RTSE is constructed in a layered, modular fashion, those programs not requiring distribution support would not pay an overhead penalty since the RTSE would be configured to the minimum support necessary. A non-distributed program would then be supported by a traditional configuration of runtime support services.

Although the details of supporting the integration of a new subsystem without first stopping that subsystem are not clear, it is felt that such an activity is impossible if separate programs are not employed.

Conclusion

As shown, many of the issues in distributing Ada programs are common to distributing any high-order language. The distribution scheme, because of its impact on the underlying RTSE complexity, should be carefully chosen when implementing distribution of the language. In making the choice, special consideration must be given to the amount of fault-tolerance required, and the level of programmer response. In Space Station, such issues will be critical.

Bibliography

A Feasibility Study to Determine the Applicability of Ada and APSE in a Multi-microprocessor Distributed Environment (Final Report, March, 1983) TXT, CISE, SPL

American National Standards Institute
Reference Manual for the Ada Programming Language
ANSI/MIL-STD-1815A-1983

Cornhill, Dennis
A Survivable Distributed Computing System for Embedded Application Programs Written in Ada
Ada LETTERS, vol. 3, no. 3, pp. 79-87

Cornhill, Dennis
Four Approaches to Partitioning Ada Programs for Execution on Distributed Targets
Proceedings of the IEEE Computer Science Conference on Ada Applications and Environments, St. Paul, MN (Oct. 15-18, 1984) pp. 153-162

DeWolf, Barton, Nancy Sodano, Roy Whittredge
Using Ada for a Distributed Fault-Tolerant System
Draper Labs Report No. CSDL-P-1942 (dated Sept. 1984)

Dapra, A., S. Gatti, S. Crespi-Reghizzi, et al
Using Ada and APSE to Support Distributed Multimicro-processor Targets
Ada LETTERS, vol. 3, no. 6, pp. 57-65

Gehani, N. H.
Concurrent Programming in the Ada Language: the Polling Bias
Software Practice and Experience, vol. 14, no. 5 pp. 413 - 427

Grover, Vinod, and Reuben Jones
Programming Distributed Applications in Ada
SofTech, Inc. Report No. 9076-3 (Dec. 1984)

Knight, John C. , and John I. A. Urquhart
On the Implementation and Use of Ada on Fault-Tolerant, Distributed Systems
Ada LETTERS, vol. 4, no. 3, pp. 53-64

Rossi, G. F., and Zicari, R.
Programming a Distributed System in Ada
Journal of Pascal and Ada, Sept/Oct 1983