Transitions to Ada: an Incremental Approach

D. H. J. Brown*
Logica Limited, Cobham Park, Cobham KT11 3LX, UK

Software producers currently use a wide variety of programming tools and management aids for software systems developments. The High Order Language Working Group of the US Department of Defense has placed as much emphasis on the provision of a co-ordinated Ada Programming Support Environment (APSE) as on the language design itself. A technique for ensuring programmer productivity during the transition period to APSE usage is outlined. The technique involves incremental replacement of functional components of the extant environment with those of the APSE.

INTRODUCTION

This paper introduces functional aspects of the transition from an extant software development environment to an Ada Programming Support Environment (APSE). The ‘big switch’ to a single host APSE is contrasted with an incremental approach gained by the introduction of a local area network to distribute the extant environment and then proceed by a series of changes toward the APSE.

The purpose of the incremental approach is to enable systems development teams to shift to the new environment by a gradual process, enabling them to remain productive during the transition. Naturally, this requires some sophistication from the transition process; it takes on responsibilities which would simply be lumped on users if a totally separate facility were provided. The use of a local area network (LAN) is suggested as a means of providing the technical framework for supporting the incremental transition process.

The architecture of a distributed APSE is illustrated in Fig. 1. In accord with Stoneman’s recommendations, the APSE is functionally divided into a kernel (KAPSE) plus toolset. The resource manager plays a central role, allocating tasks to servers. The user server provides the man-machine physical interface; the logical interface being provided by a command language interpreter which itself could run on any tool server. The database server provides the KAPSE database management facilities; it may be observed that use of a dedicated processor for database operations may be expected to achieve many of the aims of backend database machines. The test system station provides the interface between the APSE and a loosely-coupled target system. As such, it facilitates target system debugging operations; the debugger itself could run elsewhere on the LAN, communicating via the test station.

THE ‘BIG SWITCH’

Transferring from an existing—e.g. Coral/Mascot—environment straight to a single host APSE may be

---

* Current address: Dept of Computer Science, National University of Singapore, Lower Kent Ridge, Singapore 0511.
domain independent tools such as free text editors could simply be reimplemented in Ada while retaining their user interface.

**Host architectures**

Two problems for installations employing the big switch approach are: 'what happens to already developed systems requiring maintenance?' and 'what happens to partially developed systems?'. In either case, the answer hinges on the problem of whether two environments (namely the extant environment and the APSE) can co-reside on the same host. If not, hardware upgrades may be required or the forfeit of incurring additional overhead in bringing down and putting up environments will have to be paid. One way of providing the upgrade might be to tie existing hardware onto a LAN, using additional processors for APSE components. Given the better price/performance characteristics of multiple microcomputers over mainframes, a distributed system could be expected to provide greater throughput at lower cost than a single host system.

Designing a distributed APSE requires that the logical separation of APSE functions delineated in Stoneman be mirrored by physically separate processes. Logica describes a technique of distributing APSE functions among a number of processors arranged on a local area network which follows the basic principle of making the physical organization match the functional organization.

---

**THE INCREMENTAL APPROACH**

The essence of this approach is the provision of a co-ordinated composition of both the extant environment and the APSE. Compatibility between the two environments will not always be possible; in particular it is probable that they will decompose into different substructures. Nevertheless, certain functional areas can be identified which provide a transition path along which users and projects can proceed, each within its own timescale.

There are two stages by which the co-existence of environments can be achieved. The first is to provide various support facilities for developing Ada systems/subsystems within the extant environment; the second is to provide support facilities for features of the extant environment within the APSE. An incremental transition toward the APSE could be achieved by employing the first stage, followed by a shift to the second stage, dropping extant environment facilities as they become subsumed. This is outlined in Fig. 2.

Notkin and Habermann characterize the primary components of a programming support environment as:

1. A programming environment, which provides facilities for a programmer to turn specifications into a working program.
2. A system configuration environment, which provides facilities for the integration of several component subsystems into a version of a larger subsystem.
3. A management environment, which provides a means for co-ordinating the state of a system under development and for generating and proliferating system documentation.

<table>
<thead>
<tr>
<th>Stage I</th>
<th>Stage II</th>
</tr>
</thead>
<tbody>
<tr>
<td>extant language</td>
<td>Ada language</td>
</tr>
<tr>
<td>+ Ada subprograms</td>
<td>extant language subprograms</td>
</tr>
<tr>
<td>extant environment</td>
<td>access via APSE DBMS</td>
</tr>
<tr>
<td></td>
<td>APSE database</td>
</tr>
</tbody>
</table>

*Figure 2. Outline of the two-stage approach.*

Most APSE tools, as defined by Stoneman, fall within the domain of the programming environment, with facilities such as version control belonging to the system configuration environment.

**Programming environment transitions**

Including Ada language programming within the framework of an extant programming environment could be achieved by a compiler alone, which would permit training exercises to be performed. The inclusion of a linker, together with facilities for language mixing, would permit mixed language system development to be performed. In the first stage of transition, the tasking facilities of Ada could be omitted with extant tasking facilities (e.g. MASCOT activities/channels) used to implement concurrent processing. In the second stage of transition, Ada tasking would be used with extant language subprograms subordinated to Ada tasks.

**Language mixing.** Ada provides the INTERFACE pragma (Ref. 6, p. 13-11), as a means for specifying the interface to foreign language subprograms. Similarly, a means of calling Ada subprograms from within foreign programs is required. This may require the use of a preprocessor; for example, Coral permits assembly code insertions but does not explicitly provide interfaces to foreign high-level languages. Consequently, a preprocessor comprising an Ada compiler capable of generating compatible assembly-level code would be required to process source texts before they are input to the coral compiler, which itself need not be modified. Obviously, the preprocessor would need to be capable of producing compatible data structures; elementary types present no serious problems but the language interface would be unable to handle complex Ada data types beyond those available within the foreign language. For example, Coral permits only hierarchically structured records which are a subset of those definable within Ada.

The separate compilation of Ada and foreign language subprograms would be feasible provided the interface mechanisms described above are available. In addition, the extant environment’s linker would need to be modified to accommodate the run-time environment needed to support Ada subprograms; features such as dynamic allocation of space for Ada data structures are unlikely to be able to be included in the foreign language run-time environment.

The second stage of transition involves the incorporation of foreign language subprograms as the bodies of
private Ada packages. Their specifications would provide the interface to other Ada packages, with the package body being defined using the INTERFACE pragma. It is conceivable that certain frequently used functions could be described within an APSE as generic package declarations; this would assist portability between different foreign languages.

**Debugging facilities.** Debugging tools are source language dependent; it would not make sense to attempt to debug at the machine language level—software instrumentation has to be performed at the source language level to be meaningful. It is therefore desirable that—in a multi-language scenario—multi-language (i.e. composite) debuggers are available. Ideally, a composite debugger would provide its users with a common interface for source code instrumentation. It is perhaps more likely that two debuggers would be used, one for each language. This is awkward for the programmer, but poses no technical difficulties, assuming each module of the system being developed is written in a single language. Instrumentation at the inter-module level would be performed by the debugger of the dominant environment.

**Tasking issues.** Clearly, it is not feasible to upgrade an existing system which employs concurrent processing by adding on a few Ada tasks. The tasking environment should be consistent throughout any version of a given system. Although it may be technically possible to replicate a given set of tasking facilities within Ada, such an approach would make the system less maintainable.

In the first stage of transition, the use of the extant environment for control purposes implies the use of its tasking facilities; Ada subprograms could be written in as part of foreign language tasks. In the second stage, Ada tasking would be used throughout with foreign language subprograms subordinated to Ada tasks. In this way, language mixing can be freely employed without introducing unnecessary complexities.

**System configuration environment transitions**

Here too, the basic philosophy of having one environment dominate applies. Again a two-stage approach is feasible; initially the system configuration concepts of the extant environment could be used, with Ada being used only at the module level of system implementation.

The central stage of a transition to an APSE’s system configuration environment (SCE) is the introduction—to system builders—of the KAPSE database and its retrieval language. In particular, formalizing such notions as horizontal and vertical versions and creating database objects for system components is an activity which must precede the introduction of the APSE SCE. The facilities could equally well be used for systems implemented in languages other than Ada.

Mixing of SCEs should not be undertaken in respect of a single system; obviously different systems could be configured within different environments but any one project should consistently use only one SCE.

**Management environment transitions**

Traditional approaches to the management of software development employ non-technical means. Neither Stoneman nor the terms of reference of the UK APSE Study7 address this component directly. Technical facilities, such as those suggested by Notkin and Habermann,5 rely heavily on the existence of a structured system database. The introduction of technical solutions to management problems such as interprogrammer communication could only follow the introduction of the APSE SCE.

**IMPLEMENTATION CONSIDERATIONS**

The key to the incremental approach is the ability of the host to support both environments. In the first approach—that of moving to a single host APSE initially—accommodation of both environments by a single processor architecture places certain demands on its immediate access and direct access storage facilities. It is likely that many installations would be unable to meet these requirements and would be forced to adopt the ‘big switch’. Because of these difficulties, a distributed architecture offers considerable advantage. In particular, it allows the two environments to coexist without having to reside on a single processor. In this way, each environment could use certain of the other’s facilities in a controlled manner, without requiring substantial recoding of either.

**Distributing the extant environment**

The easiest way to achieve this would be to separate out the user interface of the extant environment to the extent that APSE-like user commands could be submitted via the extant host. The architecture is illustrated in Fig. 3. Ada compilations could be provided on a separate processor dedicated to the purpose. The results of

![Figure 3. A step towards the APSE.](https://academic.oup.com/comjnl/article-abstract/27/1/37/418724/5134551)
compilations would of course have to be routed back to the server hosting the extant environment.

A useful feature of a local area network is that it provides a means of coupling host and target machines with high bandwidth connection; thus a side advantage is that it facilitates the development of systems—within the extant environment—which require different host and target machines. The debugger would not necessarily have to reside on the target machine; messages could be sent from instrumentation points in the executing code to the user server. Similarly, user commands can be routed to the target via the host.

Distributing the extant environment should ideally have little impact as far as the interface to the user is concerned. Obviously, software supporting object program executions would need to be created in the case of an extant environment which did not previously support different host and target machines. The introduction of Ada as an implementation language brings with it the need for a variety of training aids. For example, although optimizing compiler would be used for production software, test compilers with comprehensive diagnostic facilities would be useful to programmers during their Ada apprenticeship. In a distributed system such compilers could be run on different machines to the production compilers, so that throughput is not adversely affected. To be sure, bottlenecks may still arise—such as access to the database server(s)—but at least they will be less severe than would be the case if a single processor were performing all functions.

Separation of the user interface could be achieved by providing a new command language with a syntax similar to that of the APSE’s command language. The purpose of this is to facilitate transition to the latter. The Ada compiler, run-time environment and debugger would have to reside within the extant environment’s file store so as to be retrievable by the same mechanisms. At this stage of the transition, the existing terminal driver(s) could continue to be employed to facilitate communication between the user and the tool he has invoked.

As a general philosophy, unless major rewrites are required, it would seem reasonable that system developers continue to use the extant implementation language. Until the introduction of Ada tasking it would not be reasonable to expect Ada to be used as the primary language for maintenance purposes; that is, it would be inefficient to rewrite faulty foreign language modules in Ada. Systems which have been designed, but not implemented, could use Ada as the primary implementation language. In this situation the configuration facilities of the extant environment continue to be used but the provision of tasking would be achieved using the facilities provided by Ada. In the case where the extant environment was MASCOT, this would require identifying activities and channels as Ada tasks, and pools as library packages whose specifications would be visible to tasks requiring access to them. Reference 8 describes a method by which this can be achieved. MASCOT queues and the kernel would be subsumed by the Ada run-time environment implementing the rendezvous mechanism.

Introduction of the APSE DBMS

The user interface is now somewhat different. Access to programs and tools of the extant environment is now provided by the database server, with consequent changes in nomenclature. It should be emphasized that the internal structure of objects would not change in any way. Indeed, it is feasible that the interim database schema could simply play the role of a file directory, with the extant environment’s file store being used as the repository for the objects. However, the transition is not contingent upon such a method; indeed it might be counter-productive—in terms of processor use—since the database server provides all the necessary facilities to retain objects, as well as reference them. The distributed architecture requires no serious disruption of extant operating systems as the database interface can be provided by a special-purpose processor hooked into the network as illustrated in Fig. 3.

As all currently used environments are non-integrated collections of tools, little functional impact will be caused by the introduction of the APSE DBMS. Different nomenclature, access conventions, and (perhaps) facilities for describing versions will be introduced, all of which impact upon the user interface. The system interfaces remain unchanged; tools operate in precisely the same manner as before. The APSE DBMS does not significantly impact systems being developed. When the full range of APSE functions—including configuration control—is made available, there will be an impact, but the database itself only affects the development system.

Bringing in the APSE DBMS paves the way for the introduction of the APSE database and toolset. It is desirable that extant environment utilities continue to be available throughout the transition. At this point, it is worthwhile to consider what the architecture of the incoming APSE might be like. There are essentially two possibilities:

1. A loosely knit collection of tools, each user-driven
2. A highly integrated facility comprising a coordinated set of tools which use a common internal representation, so that

   ... The tools ... understand each other’s functions and can collaborate toward a common goal. The integrated environment has knowledge about the objects it manipulates and their current state. It is therefore able to respond to incorrect or undesirable user actions.

The former—more conventional—approach is much simpler from the transition point of view. Ada tools would not differ in their abstract specification from those of the extant environment and could be accommodated with ease. In the latter case, with many operations being triggered from a single user activity, and with many controls and conventions to be applied, it is likely that the APSE database interface would be quite different. Using a two-stage approach—in which an extant environment schema is first employed—enables users to become familiar with the DBMS interface before being presented with the APSE schema.

Introduction of the APSE database and toolset

The physical architecture (cf. Fig. 3) remains unchanged; however, from the user’s point of view the scene shifts from one in which the extant environment dominates to
one in which the APSE is the primary support environment.

The principal contribution provided by the APSE not presented to the user in earlier transitions is the system configuration environment. This provides a means for organizing and controlling the development of systems in which multiple versions of subsystems may be required. All such meta-information (i.e. information about information) will reside in the APSE database, either in the form of a formal means of identifying system components, or in various documentation facilities, depending on the degree of integration of the APSE. At any rate, it is likely that the user interface will be different from that of the extant environment.

From the system development system point of view, this stage constitutes the major transition. The path has been smoothed by the introduction of the APSE database. The provision of additional APSE tools, in particular those associated with system configuration creation and control, is readily achieved as the database should already contain the superstructure necessary to accommodate them.

The introduction of a configuration environment strongly impacts ongoing developments. Each project team will be faced with the choice of whether to move to the new environment or to continue using the facilities of the previous one. To facilitate the latter choice, it is important that the extant environment remains available to systems developers requiring it. Moving to the APSE would involve rewriting system descriptions, and creating the necessary database objects to contain them. For near-complete systems such an activity is unlikely to be cost-effective.

**SUMMARY**

Transitions are always complicated affairs. The simpler the technical facilities supporting an environment transition, the more complicated it is for users of the environments. In the extreme case, a ‘big switch’ is thrown and users turn from one environment to another.

The usual technique to alleviate the problems a big switch causes is to provide access to both environments, one for continuing work and the other for training in the new system. It is debatable whether this does in fact improve productivity, for it places additional cognitive loads on the users. An alternative, incremental, approach has been outlined. To be sure, it involves a great deal more preparation on the part of the suppliers of ‘transition aids’, but it may be hoped that development teams—who are the users going through the transition—might profit by it.

In conclusion, there is an inherent complexity in making a transition. The approach suggested here is an attempt to transfer some of that complexity from the users’ minds to the technical support facilities.

**Acknowledgements**

Work reported here was partially supported by the UK Royal Radar and Signals Establishment (RSRE) under contract A71c/6023. The opinions and ideas expressed herein are entirely those of the author; they do not necessarily represent the official view of either RSRE or Logica Limited. The concept of using a local area network to implement a distributed APSE was conceived and developed by Logica Limited. Significant contributions to the network architecture—which provides a basis for the ideas reported here—were made by Andrew Arbaster, Annette Hughes, John Wilson and Robert Worden.

**REFERENCES**


Received February 1983