TRAINING SYSTEM BUILDING TECHNIQUES & THE POTENTIAL OF Ada

E.L.Averill Software Staff Engineer
T.M.Choy Principal Lead Software Engineer
Honeywell T&CSD West Covina Ca.

June 5, 1986

ABSTRACT

The acquisition of Training Systems is being stressed continually by the advent of **requirement change.** The cost and schedule impacts from such changes are felt throughout **the acquisition and fielding process.**

The use of FORTRAN places **constraints** upon the development process which limits ability to reduce the effect of requirement change. Preliminary experience with Ada [1] in the *training arena shows that an* **Ada environment does not share the** FORTRAN **limits.**

The paper examines the essential differences between the Ada and FORTRAN software environments in relation to both **requirements change** and to the introduction of **generic "standard" components** within Trainer Products. The concept of generic components is defined in terms to show how an Ada environment facilitates in ways a FORTRAN environment cannot.

Support software prerequisites (particularly for Training System products) are identified and are shown to be necessary to allow developers to exploit the potential within structures that are part of the Ada language.

INTRODUCTION

This paper identifies product differences between a FORTRAN and an Ada software environment. The paper relates these differences to the most desireable characteristics currently not provided by trainers. The paper then summarily indicates how the potential of Ada can be engineered to provide these characteristics for acceptable costs.

A major training school concern is training effectiveness. A primary cause of trainer in-effectiveness is the difficulty training schools have in responding to changes in trainer's requirements. Changes occur throughout the life-cycle of the trainer. A requirement change arises from such things as:

- changes to the equipment the trainer relates to,
- field operational changes (for example technical order changes which must be reflected in the trainer),
- a need to upgrade the trainer effectiveness based upon the experience of using the trainer,
- a need to improve the trainer's instructional capabilities,

Currently training schools are experiencing problems because of the conflict between the need to keep their trainers current, and the costs and frequency of change that are necessary to keep their trainers current.

The techniques used to develop and build a trainer establish a floor for the costs associated with modifying the trainer, *(in terms of time, dollars, and the prerequisite people skills).*

The techniques used to build the trainer are part of the design and development process, which collectively are the **Process Technology,** as opposed to the **Product Technology** which refers to the components out of which the trainer is constructed.

The paper compares the use of FORTRAN with the use of Ada in terms of the effect it has on the *Product Technology.* This comparison is then extrapolated to show how the potential of Ada can be exploited to make substantial changes to the *Process Technology* used to build and to modify training systems.

Changing the process technology is significant because experience with software estimating techniques ([1]) shows the most significant factor which increases cost and schedule of a product is the *Process Technology* used.

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

Need to Change Training System Requirements

Under the Figure 1 process technology, changes to requirements are applied to existing implementation detail (once a trainer is developed). Typically the pathway from requirements through design and development is different from the pathway used for making modifications to an existing trainer product. In the original development the "higher level" requirement and specification documents are interpreted by engineers into the "low level" implementation which becomes the delivered product. In the modification pathway shown in Figure 1, it is the "low level" implementation detail only that is changed.

However the Figure 1 technology can be upgraded if requirements are separated into fixed and changeable parts

- and if the **fixed requirements** [1] are engineered into a product architecture for the trainer type, and if
- the scope of things within the **changeable requirements** [2] are engineered into a modification pathway that (as shown in Figure 2) can serve both for the original development as well as for change through the product's life-cycle.

Figure 1: Currently Changes are made to the Low Level Implementation.

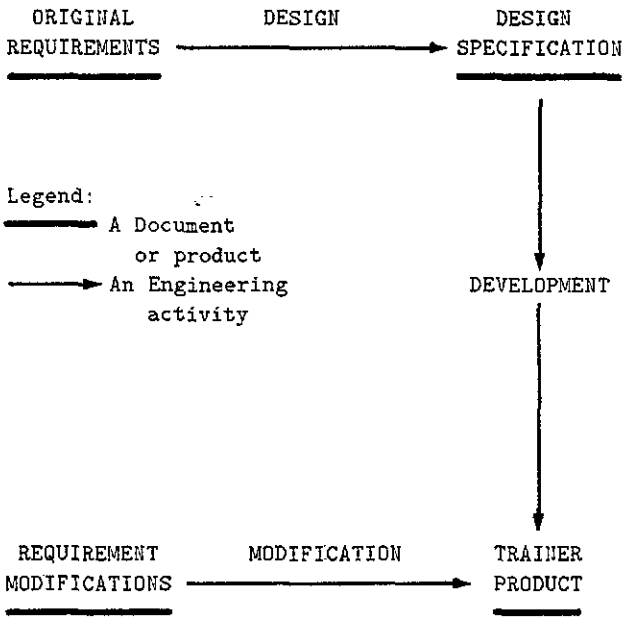
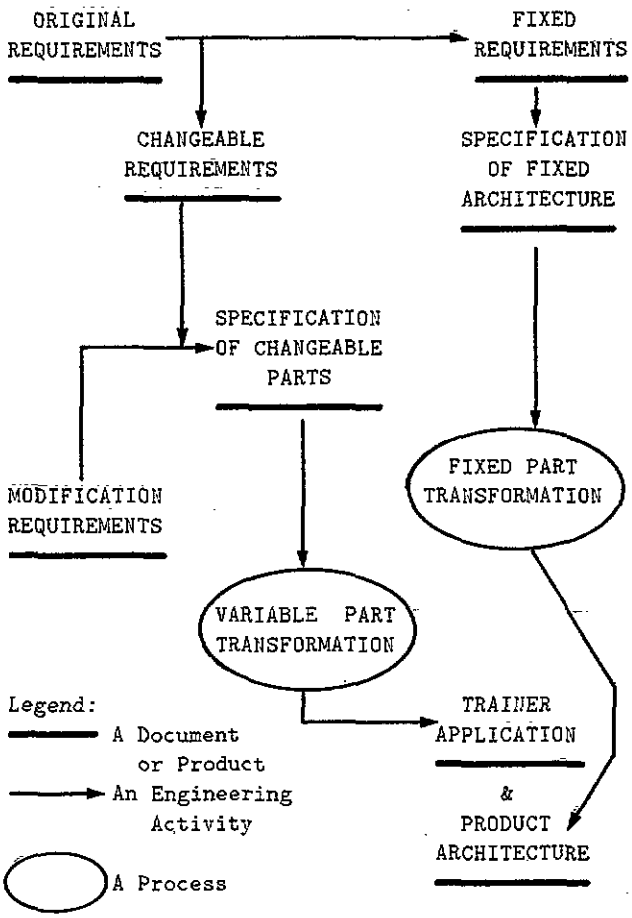


Figure 2 illustrates the process technology that can be achieved based upon requirement separation into fixed and variable parts. The paper indicates that this process technology is a practical possibility due to the potential of Ada.

[1] Fixed Requirements are represented by project independent specification.
[2] Changeable Requirements are represented by project dependent specification.

Figure 2: Ada Facilitates Design of Modification Pathways into the Product



A trainer product which provides capabilities for the training school to keep the trainer current has to have the Figure 2 type technology designed into the trainer.

The paper compares FORTRAN and Ada and states the opinion of the writers that Ada facilitates such a design while FORTRAN lacks essential characteristics to make such a design practical.

With a trainer designed to be kept current by the training school, changes will be made by changing the trainers' specification. The specification will have to be formalized so that the trainer software (which provides the "changeable requirements part") can be regenerated.

Experience ([2]) shows that, with current technologies, changes at design time cost a very small fraction (dollars & time) of the cost required by the same change at or after acceptance test time, (at which point under the Figure 1 process technology, it is the implementation detail that has to be modified). From this experience it is clear that the cost of change will be very significantly reduced if trainers can be kept current by regeneration instead of modification.

PM Trade's Standard Trainer Components

PM Trades Five Year Development Plan (1985-1989) ([3]) calls for the standardization of component items within a maintenance trainer. This introduces **a set of potential contraints** into the original design and development process. The writers feel that there is significant synergism between the two needs

1. to build in ease of changeability **via a modification transformation process** as per Figure 2,
2. to build trainer type products out of **standard components.**

Both introduce development costs at the front end in order to establish the technology. Both aim for long term life-cycle cost reductions. The PM Trade plan identifies

> "Problem: There are currently no means for obtaining compatibility among Army maintenance training devices or the computer software which supports them. ....
>
> *Objective: To design a specification which* will support acquisition of a wide-scope family of microcomputer-based maintenance trainer systems wherein hardware and software modules from any acquisition will be directly interfaceable and usable with modules from any other acquisition."

This PM Trade objective is distinct from the objective of keeping a training product current. However both involve standardization within the design. There exists sufficient commonality in the strategies needed to realize these objectives that a single design may be conceived to serve both purposes. Further, such a design is obtainable with an Ada software environment. With the FORTRAN language the same design may be possible, but the practical difficulties *arising from using FORTRAN probably will prevent realization* of the objectives.

DETAILED CONSIDERATIONS

In this section we first identify the types of requirement that training schools wish to change [3] and under what conditions. Then we examine the significance of FORTRAN and Ada to the trainer system product.

Changeable Requirement Examples

Changeable requirements for trainers are in the courseware, in the simulation, and in how the field equipment is represented to the trainee. Specific examples of change for *maintenance trainers,* can be changing a step within an existing procedure of a particular lesson, such as

- introducing at a new place a display from the visual system, or
- causing a specific panel output to occur, or
- the change can be more demanding such as changing an existing simulation module, or
- adding a new lesson, or
- replanning the lesson steps returned to in response to a "repeat task" command.
- if graphic displays are used to represent the field equipment, then the detail of the representation is open to change.

Cost Implications of Requirement Change

We know from ([1]) and ([2]) that the cost of making changes upon the implementation detail of a product is *in general much greater than the cost, per unit of functionality,* of the original development.

We can compare the cost/time characteristics of the modification of a trainer product with the charateristics of the original development, by considering the cost to add a floor to a multistory building. It can be readily appreciated the addition of a floor can require more work than the original building because the existing design did not include what was necessary to support the extra floor. Hence all the preparation and support for the change has to be included in the cost.

The essential causes of cost escalation are similar for both software and physical systems. The extra cost comes from rework that has to be done.

So in summary, if it is true that trainer systems

- will continue to be subject to changes in their requirements throughout their life-cycle, *then*
- considerations towards minimizing life-cycle cost will continue to support the need for cost effective **modification pathways** to be designed into the original trainer. Pathways to allow training schools to respond to the ongoing need for change within their operational resources and within their environment.

A Systems Engineering Comparison

FORTRAN and Ada are not compared in terms of their instruction repertoire or in terms of the environmental support to the using software engineer. The comparison made is in terms of the organization and structure that can be put into the product as a result of the language constructs. FORTRAN encourages a relatively low level and procedural formulation of the requirements specification and the top level design documents, *which introduces cost/time problems when it comes to making changes to the product after it is delivered.* A low level, procedural approach to the specification submerges the visibility of user's needs within the implementation detail. Ada allows the system's software requirements and top level design to be stated in data structure and data transform operational terms, which can make

[3] We recognize derived requirements will be affected also, but felt changes to derived requirements are not significant to the main discussion.

the user's needs more visible within the design, and accessible to change. Hence Ada designs can be engineered to facilitate changes for those requirements which are known to be subject to change.

Probably the most critical visibility for a trainer system product is the interface between the instructor and trainee users and the trainer. In FORTRAN there is no more capability to design a high level representation of the user's requirements than there is in assembly or machine language. Ada's package mechanism enables all user's world objects to be represented in the design with all their direct interfaces with the user. Few system modifications do not require a modification to how the system product interfaces with the user.

The objective of our comparison between Ada and FORTRAN is to compare them in terms of the ease of building into the system product Figure 2 type properties. Note the Figure 2 type properties must be designed into the product, and into the process which builds the product. The fact is that Figure 2 type properties rely on the internal organization and design of the software within the system product.

We, therefore, compare FORTRAN and Ada by examining the effect each has upon the specifics of the trainer product.

FORTRAN's Effect on the Product

FORTRAN's product is a hierarchy of functional procedures (routines and subroutines) which require a global set of variables to connect the routines into a meaningful whole. The root node of the hierarchy is firmly fixed at the implementation level with the operating system, the hardware configuration, and runtime environment. It contains no notion of environment beyond its single thread of control and memory space. A FORTAN system changes state by the routines making changes to the global variables. There is no provision or assistance (except an array) for organizing and managing the data structures by means of which the system state may be controlled.

Table 1: Product Differences for FORTRAN and Ada coded Systems

ITEM	Ada	FORTRAN	Environment Effect
Threads of Control	Multiple	Single	'Single' forces dependence upon operating system, and constrains scope
Memory	Dynamic	Static	'Static' inhibits virtual memory (pointers require Assembly Code)
Built-in Data Structures	Full Set	Array Only	The lack of built-in structures mandates low level design thinking
User Objects as Data Structures	Full Set	None	A Data Structuring capability is prerequisite to ease of change
Built-in I/O Potential	Full I/O support	No I/O Potential	Full support facilitates ease of change & wider scope
Algorithmic Structuring	Full Set	Primitive	Primitive implies a wide gap between requirements and implementation
Recursion	Provided	None	Recursion requires well developed structuring
Exception Handling	Full Support	None	Exceptions always exist, 'handling' provides formal interface to main algorithmic logic
Writing of I/O Drivers	Possible	Not Possible	For FORTRAN Assembly code is needed: Ada provides machine representation capabilities
Task Types	Full Support	None	Task Types allow for maximum parallel execution pathways at run time

Ada's Effect on the Product

Ada's data structure specification can start in the user's operational theater with real world objects. There is a set of objects in the user's operation that the trainer system must have represented in its memory.

Ada not only facilitates the specification of the data structures, but it creates a structure in the product which is an accurate representation of the objects within the user's operation.

The data specifications become a collection of separately defined data packages for which the Ada language provides considerable language support.

For each separate system object represented by a data structure, (in Ada a *Package*), Ada directly associates the operations needed to operate upon that structure. The scope of the logic within the operations (in Ada a *subprogram*), is clearly defineable in terms of pre- & post-operation data states.

Data structure independence from implementation strategy and configuration, means that there is, within the product, a data level with a clean interface to the lower levels in the product structure. (This support is in direct contrast to the amorphous data mass provided by FORTRAN's common data block of global variables.)

More Detailed Ada/Fortran Comparisons

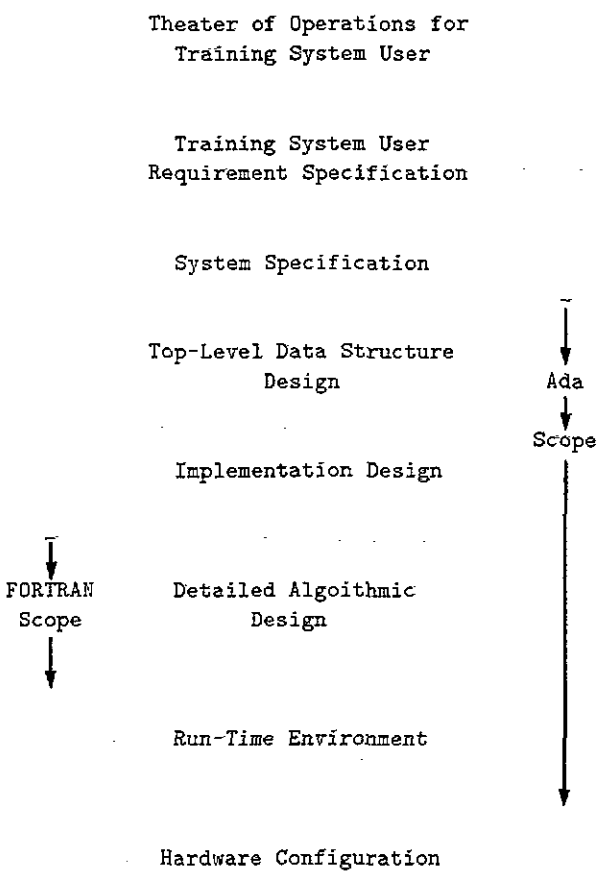
Table 1 compares those features in the Ada and FORTRAN languages which have significant effect upon the compiled code within the system product.

FORTRAN is a language which has proved its effectiveness and limitations over more than 20 years. Although it has been upgraded to include some algorithmic structuring, it still relies on informal use of "go to's" to carry out its functioning. The problem here is that there is no way to relate the "go to" connections to the function being performed except by detailed examination of the "go to" network, *this may be compared to analysing wiring without the aid of a wiring diagram.* Further FORTRAN allows use of subroutines, but again the use is informal and requires detailed instance by instance examination to comprehend the system behavior implications.

FORTRAN's scope is too limited to provide a product structure that facilitates the design of a product architecture with the characteristics portrayed in Figure 2. FORTRAN allows only one thread of hardware control and so cannot properly relate to the needs of managing its underlying hardware configuration. All peripheral device drivers and interrupt handling have to be written in assembly level language. A software engineer user has to rely heavily on specific non-standard operating system functionality to fullfil basic needs for establishing the software for a system.

On the other hand Ada has been designed upon the experience of more than twenty years use of dozens of different languages. It has a much wider scope. A comparison of the scope differences is illustrated in Figure 3.

Figure 3: Scope Comparison Between Ada & FORTRAN



The Potential of Ada

The Ada potential that we are referring to depends upon the conceptual notions which gave birth to the language. We draw attention to specific features.

The Ada language provides the software engineer with a complete set of descriptive and specification capabilities. It is not necessary to go outside the language to make a complete specification of the low level functionality required for the hardware configuration.

This completeness is a very significant facilitator to the successful development of system/software generation capability for system engineers. So the potential of Ada for training systems is in its ability to support future system specification languages.

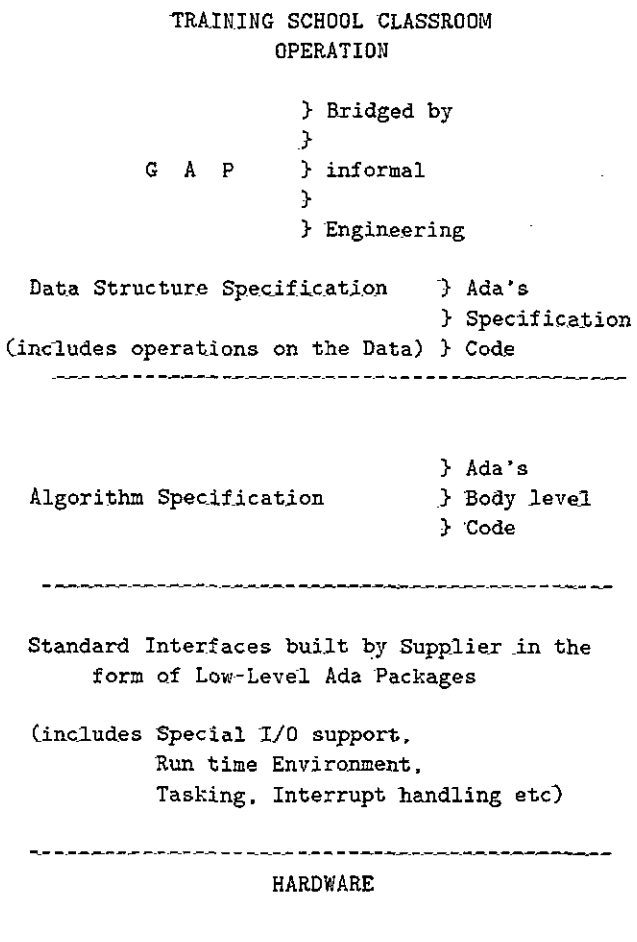
Figure 4 depicts the different levels within the Ada language, and relates these levels to both the user's world and underlying Hardware.

When examining Figure 4 we need to recognize that all functionality is achieved by data transforms.

Ada provides very considerable data typing features which preserve, the intent of the user's operation, in the implementation specifics. For instance if a particular switch is defined to be a particular type, then it would not be possible to perform operations upon that switch which were not defined as applicable in the data specification for that type.

Hence with Ada the systems analysis of the training need can be accurately represented in the Ada data specifications. These data specifications are sure to be accurately translated into executeable memory environment by the compiler.

Figure 4: How Ada Relates to a Training System Environment



SOME PREREQUISITES TO EXPLOIT THE POTENTIAL OF ADA

This section of the paper identifies some prerequisites, besides the Ada language potential, to make the updating of training systems possible and within the training school budgets. The same prerequisites facilitate the acquisition of training systems with inter-operable components.

Generics and separate compiling are two Ada features not mentioned in Table 1 because they do not change the execution of code. However both are prerequisite for the reusability of engineering labor in the generation of Training systems. Generics enable very high level macros to be developed, which can then be specialized for a specific need without having to recreate the common framework. Both are necessary to support a Figure 2 process technology.

Separate compiling allows for the management of Ada code "pieceparts", and allows the pieceparts to be compiled on their own, and then added in to other compilations by making reference to the reuseable part.

Ada provides only the potential. To realize the desired training system behaviors there has to be investment in the technology of system generation. Creating the means for the generation of a system, *such as a training system*, requires very similar antecedents to the creation of a hardware factory. The following antecedents are seen to be necessary:

1. The specification of a **PRODUCT TYPE**, (which sets out the scope of a particular software factory,)

2. A clear and rigorous separation of fixed from changeable requirements for products belonging to the product type,

3. The full development and maturation of an Ada partitioning language along the research and development lines documented by a Honeywell Report ([4]), (this will allow both the Ada specifications and the Ada bodies to be truly independent of both hardware and configuration),

4. The development of a high level language *(which is dedicated to the product type,)* and which allows specification of the variable (so far called changeable) requirements, via graphic and tabular means after the style of Teledyne Brown Engineering TAGS ([5]),

5. The development of interactive graphical tools for the transformation of specifications in the "product language" into standard databases (which can be used to recreate the diagram on many monitors, printers, and plotters); databases which can be transformed via Ada tools into lower level engineering products/data bases, some of which may act as frameworks of diagrams or tables for completion by an engineer who adds further specification... which gets put into an enriched database and so on until by automation assisted stepwise refinement and decomposition the product is fully defined.

   (The Introduction to an Army report ([6]) clearly defines the connection between a specification and one of its possible implementations, the report lays a formal foundation for the use of specification languages).

6. The preservation of all the intermediate products (i.e. the databases) each of which represents a particular level of specification for one product,

7. The development of a special tool set which allows an end-user to make changes to the content of these databases, which then can be directed to regenerate the product with changed requirements.

SUMMARY

The paper shows how the Ada language facilitates a process technology in which there is a separation of requirements into fixed and changeable camps while FORTRAN does not.

The paper identifies what the Ada language brings to a training system product and to the development process that is not available with the FORTRAN language. In particular it focuses upon those differences which affect our ability to build modification pathways (as per Figure 2) and standard interfaces (as per PM Trade's Generic Trainer) into trainer product types.

The paper has only summarily defined the generation process, and cannot go further because of the size of the topic. It has to leave to the experience of the reader to see the connection between the Ada's potential and the demand for a product generation process.

FURTHER WORK

It is our viewpoint that the following extra research work will speed up the realization of Ada's potential to the training system community.

1. Industry and the training system System Program Offices to work upon the pre-requisites for adopting a standard product type approach to replace the current method of individual total specification for a trainer.

2. The exploration of a product line specification language development:

   - the specification language to become the means of specifying the variable requirements, and
   - the semantics of the language to be formally directed towards a product architecture,
   - a product architecture to satisfy the fixed requirements.

   (Such architectures could include the standard interfaces for the generic trainer which the Army seeks to bring into being.)

3. Standard database design tools for graphic diagrams and parameter tables for integration into the engineering process described in item 5 above.

REFERENCES

(1) R.W.Jensen, "An Improved Macrolevel Software Development Resource Estimation Model", Proceedings of the Fifth International Society of Parametric Analysts Conference, St. Louis, MO, April 26-28, 1983.

(2) B.W.Boehem, "Software Engineering Economics", Englwood Cliffs, N.J.(1981); Prentice Hall.

(3) Project Manager for Training Devices (PM TRADE) 1985-1989 Simulation and Training Device Technology, Five Year Exploratory Development Plan, Approved by James W.Ball Colonel ORDC Orlando Florida

(4) September 1985 Preliminary Report on "The Ada Program Partitioning Language" and "Ada Program Partitioning Language - Reference Manual" Honeywell System and Research Center Distributed Ada Project,

(5) January 1986 "Technology for the Automated Generation of Systems (TAGS) - Reference Manual" Teledyne Brown Engineering, Cummings Research Park, Huntsville, Alabama 35807.

(6) Leslie Lamport, SRI International, ARO 20628.5-EL "Specifying and Verifying Concurrent Programs" Final Report Feb 1985.

ABOUT THE AUTHORS

EDWARD AVERILL is a Software Staff Engineer for Honeywell T&CSD. He started his software career in 1955 working with machine language and paper input and output. For many years he interfaced directly with the users, and to satisfy their needs, worked all parts of the life-cycle; requirements, design, production, sell-off, and follow-up maintenance.

His current challenge is leveraging the on-going hardware and software support system advances to enable competitive software engineering development within the production of real-time system products. The target is the application software within the Training and Naval Combat System products made at the T&CSD operation within Honeywell Inc.'s Aerospace Defense Group.

THOMAS CHOY received BSCS and MSCS degrees from the University of Southern California in 1977 and 1979 respectively.

Currently, at Honeywell T&CSD, he is the lead software engineer of two Ada projects for the Air Force and Navy. His industrial experiences are in the spacecraft real-time data systems, maintenance, operator, and visual trainers. His research interests are in the areas of real-time parallel processing systems, programming language translators, and programming language theory.