TRW Systems Engineering &
Development Division

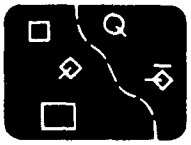
TRW-TS-89-02

Reliable, Reusable Ada Components for Constructing Large, Distributed Multi-Task Networks: Network Architecture Series (NSA)

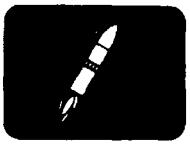
Walker Royce

December 1989

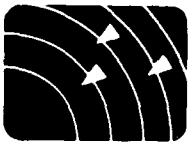
TRW Technology Series



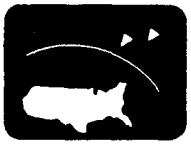
TRW Technology Series



TRW Technology Series

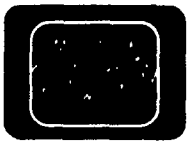


TRW Technology Series

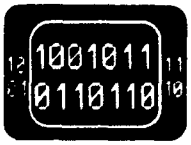


TRW Technology Series

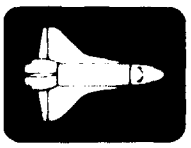
TRW Technology Series



TRW Technology Series



TRW Technology Series



echnology Series

Reliable, Reusable Ada Components For Constructing Large, Distributed Multi-Task Networks: Network Architecture Services (NAS)

Walker Royce
TRW Defense Systems Group
Redondo Beach, California

ABSTRACT

This paper will introduce the key concepts of TRW's Reusable Message Based Design Software (Network Architecture Services- NAS) which has proven to be key to the CCPDS-R project's progress to date. The NAS software and supporting tools have provided the CCPDS-R Project team with reliable, powerful building blocks that have been integrated into extensive demonstrations to validate the critical design approaches. The CCPDS-R PDR Demonstration consisted of 130 Ada tasks interconnected via 450 different task to task interfaces, executing in a network of 3 VAX nodes. The extensive reuse of NAS software building blocks and Ada generics resulted in the translation of 120,000 Ada Source lines into over 2 million lines of executable machine language instructions. The NAS software (about 20,000 Ada source lines) was conceived in a TRW Independent Research and Development project in 1985, and has since been refined and evolved into a truly reusable state. Although NAS reuse is limited currently to Digital Equipment Corporation VAX VMS networks, efforts are underway to provide heterogeneous NAS capabilities.

The advantages of NAS usage are twofold:

1. **Value added operational software** through reuse of mission independent, performance tunable components which support open architectures, and
2. **Overall project productivity enhancement** as a result of NAS support for rapid prototyping, runtime instrumentation toolsuite, and encapsulation of the difficult capabilities required in any distributed real-time system into a standard set of building blocks with simple applications interfaces. This isolation of the "hard parts" into an easily used standard software chipset, results in a large net reduction in applications software complexity, less reliance on scarce real-time programming experts, and a substantial reduction in overall project risk.

This paper describes the message based design techniques which drove us to the development of NAS, the capabilities and components inherent in the NAS product, and the CCPDS-R experience in using NAS in a stringent real time command and control environment.

PROJECT BACKGROUND

The Command Center Processing and Display System Replacement (CCPDS-R) project will provide display Information used during emergency conferences by the National Command Authorities; Chairman, Joint Chiefs of Staff; Commander in Chief North American Aerospace Command; Commander in Chief United States Space Command; Commander in Chief Strategic Air Command; and other nuclear capable Commanders in Chief. It is the missile warning element of the new Integrated Tactical Warning/Attack Assessment system architecture developed by North American Aerospace Defense Command/Air Force Space Command.

The CCPDS-R project is being procured by Air Force Systems Command Headquarters Electronic Systems Division (ESD) at Hanscom AFB and was awarded to TRW Defense Systems Group in June 1987. TRW will build three subsystems. The first, identified as the Common Subsystem, is 24 months

into development. The Common Subsystem consists of 325,000 source lines of Ada with a development schedule of 38 months. It will be a highly reliable, real-time distributed system with a sophisticated User Interface and stringent performance requirements implemented entirely in Ada. CCPDS-R Ada risks were originally a very serious concern. At the time of contract definition, Ada host and target environment, along with Ada trained personnel availability were questionable.

With cooperative innovation on the part of both TRW and ESD/MITRE over the last 5 years in preparation for the Air Force's first large Ada project, the execution of the first 24 months has been extraordinary. Key to success in these efforts are:

1. An innovative Ada Process Model for software development and test [Royce 89], and
2. A powerful set of real-time building blocks called Network Architecture Services (NAS).

The genesis of the Ada Process Model and NAS products was a TRW Independent Research and Development Project which pioneered the technology from 1984-1987 and provided the key software personnel for CCPDS-R startup.

Message Based Design

TRW's definition of *Message Based Design* is a methodology for constructing a distributed software solution using a predefined set of network objects and a predefined set of operations on those objects. In message based software architectures, the top level executable software components are tasks which are restricted to using explicit message passing as the sole means of data and control flow. At the top-level, our message based design approach is very object oriented as described in [Booch 1987]. The fundamental requirements which drove the development of NAS are:

> NAS should provide a mission independent set of reusable software components which supports the construction of multi-task Ada networks which are (compilation) independent of the underlying hardware network. NAS building blocks should provide a uniform set of tailorable control structures which provide a standardized set of control interfaces. The control interfaces should be encapsulated so that applications task networks can be constructed by defining their intertask interfaces through data coupling only. NAS should not be compilation dependent on any applications software. Finally, NAS components should support fault tolerant network operation, and provide the instrumentation to monitor network health.

The Network Architecture Services (NAS) product provides the objects and operations needed to construct robust real time networks which support flexible, open architectures. These networks consist of the following logical objects:

(1) Tasks (Software Processing Engines)

(2) Task to Task Communication Sockets (logical input/output ports)

(3) Task to Task Communication Circuits (logical connections of sockets)

(4) Messages (intertask stimuli and responses)

Given infinite processing power and memory, a system architect would need only to define the above objects to architect a specific application. Since real systems must be built out of distributed hardware components with finite performance and reliability constraints, the software architect's solution can become significantly more complicated. To ease this complexity, the NAS components must isolate the knowledge of the underlying hardware architecture so that the application tasks can communicate with each other through logical names without knowing their physical residence. This allows the software to be allocatable to an arbitrary network of processing nodes.

To further complicate the software architecture, reliability/availability requirements introduce software redundancy and reconfiguration needs. Since Ada provides no facilities for compiling programs across multiple processors, nor any for communicating between tasks on different processors, the unit of software redundancy must be an Ada main program, which we will call a *process*. The facility for

communicating between tasks within an Ada Program via task rendezvous or shared data is defined in the Ada language. The ability to communicate between tasks residing in different processes (whether residing on the same node or not) is provided by NAS.

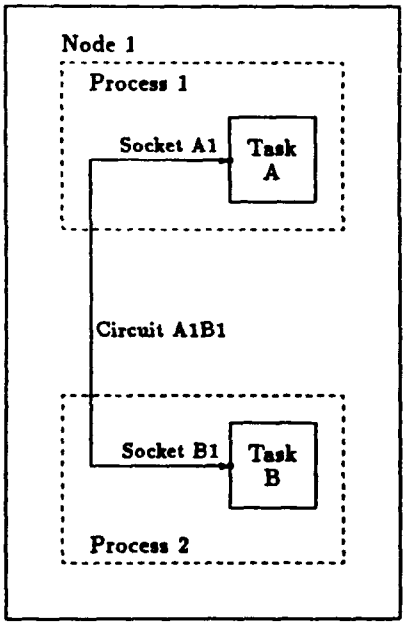
The added dimensions of performance and redundancy introduces a new set of objects which must be defined in addition to the objects listed previously:

(5) Processes (collections of tasks within an Ada Main Program)

(6) Nodes (collections of processes)

The advantage of a message based design solution is increased software flexibility and processing architecture independence. This flexibility and independence can be exploited in a software architecture to provide high reliability, maintainability and performance.

Figure 1 identifies graphically the objects available for constructing a NAS network. Without elaborating on implementation details, the flexibility and power of these objects is best portrayed by the "operations" provided for manipulating the various network components. The table identifies some of the more important operations and descriptions.



Object	Example Operations
Network	Create, Initialize, Reconfigure, Monitor, Shutdown, Freese, Applications unique
Node	Add, Delete, Initialise, Reconfigure, Monitor, Shutdown, Freese, Applications unique
Process	Add, Delete, Initialise, Reconfigure, Monitor, Shutdown, Freese, Restart, Applications unique
Task	Start_Up, Initialise, Reconfigure, Monitor, Shutdown, Abort
Socket	Create, Connect, Buffer, Flush_Buffer, Delete, Initialise, Reconfigure, Monitor, Close, Query Status, Deliver Self Stimulus, Cancel Self Stimulus
Circuit	Connect, Disconnect, Query Status
Message	Read, Write, Get_Next, Flush

Figure 1: Message Based Design Objects

The operations listed in the table illustrate the mainstream capabilities provided by NAS. There are many other off-nominal operations and attributes provided for added applications tailoring, flexibility, performance and robustness. For example, tasks and processes can be created with different priorities. Sockets can be created with several attributes including: priority, CRC checking , overwriting, buffer timeout, buffer size, direction (in, out, in/out), etc.

Advantages of Message Based Design With NAS

The NAS implementation of message based design provides complete encapsulation of the multiprocessor architecture details within the InterTask Communications (ITC) component. This means that application tasks send and receive messages of predefined types without the applications tasks requiring compile time knowledge of the location of the receiver(s) or sender(s) nor NAS compile time knowledge of the application object being passed. The advantages of this approach include:

- *Processing Architecture Independence.* Processing architecture is the exact configuration of hardware CPU resources (processing nodes) and the allocation of executable images (software processes corresponding to Ada main programs) to those resources. Message based design with NAS enforces processing architecture independent design by requiring the exclusive use of logical input/output (I/O).

- *Applications Software Independence.* By providing NAS components which are not compilation dependent on any applications software components, true reusability is obtained. By providing generic building blocks which are instantiated into applications components, the standard interfaces which other components rely on can be maintained and integration of these components remains simple. This independence is fundamental to the construction of a software architecture skeleton which can be incrementally augmented as required in TRW's Ada Process Model [Royce 1989]. This independence provides for the early construction of a software architecture skeleton which can be used as a demonstration testbed for early top-down interface integration and testing.

- *Software Modularity.* The top level software architecture is now definable in terms of standard software building blocks with well-defined behavior and interfaces. This eliminates one major source of errors inherent in large real time distributed system development, those in the executive control logic. It promotes a cohesive and modular software design by enforcing all task-to-task interfaces to be pure data coupling.

- *Reduced Ada Training.* Ada training costs and development risk are reduced by encapsulating Ada tasking and other difficult real time functions within NAS. Designers can implement multitasking applications with all the benefits of using Ada without needing the relatively complex tasking mechanism. The tasking mechanism is however, not precluded from being used for specific applications needs.

- *Simple Software Interfaces.* Synchronization problems among communicating application tasks are eliminated. Sending tasks need not synchronize with receiving tasks to accomplish communication. They simply send. ITC buffers the messages and delivers them to the receiving tasks. The synchronization required is accomplished using Ada tasking within NAS. In essence, NAS provides standard and reliable intertask control interfaces so that the applications designers need only concern themselves with the applications (data) interfaces.

- *Increased Productivity and Reliability.* By reusing proven components which implement the (typically) most challenging C^3 software executive capabilities, a substantial productivity increase can be realized. An even more significant reduction in life cycle development cost is realized through the early availability of a working reliable software executive with the flexibility necessary to promote incremental evolution of the product in a stable operational configuration. Furthermore, by building the software from a standard set of building blocks, uniformity in development can be exploited through automated source code generation. This topic is discussed later in this paper.

Software Architecture Skeleton

The concept of a software architecture skeleton (SAS) is fundamental to the incremental development approach prescribed by TRW's Ada Process Model [Royce 1989]. Although different applications domains may define the SAS differently, it should encompass *the declarative view of the solution which identifies all top level executable components (Ada Main Programs and Tasks), all control interfaces between these components, and all type definitions for data interfaces between these components.* Although a SAS should compile, it will not necessarily execute without software which provides data stimuli and responses. The purpose of the SAS is to provide the structure/interface baseline environment for integrating evolving components into demonstrations. The SAS represents the forum for interface evolution between components. In essence, a SAS provides only software potential energy; a framework to execute and a definition of the stimulus/response communications network. Software work is only performed when stimuli are provided along with applications components which transform stimuli into responses. If an explicit subset of stimuli and applications components are provided, a system thread can be made

visible. The incremental selection of stimuli and applications components constitutes the basis of the build a little, test a little approach of the Ada Process Model. It is important to construct a candidate SAS early, evolve it into a stable baseline, and continue to enhance, augment, and maintain the SAS as the remaining design evolves. The CCPDS-R SAS consists of all:

1. Ada Main Programs
2. Ada Tasks
3. Sockets, Socket attributes, and connections
4. Data types for objects passed across sockets, also called System Global Interfaces (SGI)
5. NAS Components

A SAS is not intended to be static. Early stability is unlikely since the information content in the SAS components is likely to evolve with use. To this end, it is important point to construct an early SAS which provides a vehicle for interface solidification, and *the flexibility* to adapt the SAS components to accommodate rapid assimilation of design interfaces. It is these SAS qualities which have driven the development and evolution of the NAS products.

There are two important aspects of SAS verification and assessment: compilation and usage. Just constructing and compiling all of the SAS objects together is an important and non-trivial task. It results in vital feedback on program structure and interface consistency. Using a SAS corresponds to constructing applications software which exercises interfaces in the SAS by creating threads of capability. This usage will provide further feedback on structural integrity, interface semantics and runtime interactions and interdependencies.

Figure 2 identifies the NAS building blocks and the applications components which must be developed to construct an operational network.

The top level NAS building blocks include:

ITC Services ITC provides network transport object definitions and operations.

Generic Applications Control (GAC) Process and Task The GAC components provide generic Ada packages which can be tailored to a specific application through instantiation. Although many generic parameters are specified for tailoring these generics, the primary customization is done by providing different "withed" packages at instantiation time. These packages provide different message processing procedures tailored to different control and applications messages.

NAS Network Management and Instrumentation The NAS Network Management components provide for interactive Network initialization, reconfiguration, termination and monitoring. The NAS Network management components (which use ITC) provide operational Ada packages that can be reused in any C^3 network of VAX nodes operating under VMS.

The NAS components are instantiated into an application network by providing the following Application unique software:

Process Executives The Application Processes provide the executable objects within the network. Processes correspond to Ada main programs which contain applications tasks. Processes are reconfigurable (i.e., transportable) software entities.

Task Executives Standard instantiations of the generic task template which define a particular applications task instance.

Task Configurations Task Configurations are logical subsets of the physical network configuration which are associated with a specific task.

Applications Task Processing Application components provide the actual "Data Transformation" procedures which process task inputs and create task outputs. Tasks are subordinate to processes and correspond to the components which perform "software work".

Applications Unique Process Control "Control procedures" can be provided to tailor the instantiated Process and Task Executives to unique control environments (e.g., redundancy management, checkpoint/restart).

Applications Network Definition The Application's Network components define the network circuits (task to task connections) and socket attributes (buffering, connections, etc.)

Global Message Objects These are the objects which are passed over circuits. These are the highest level software (data) interfaces between applications tasks.

The loosely coupled message based design concepts described in the previous section permit a very simple, standard scheme for executive control. This not only increases network reliability and understandability but also contributes substantially to increased productivity. The following sections provide an overview of the key NAS building blocks.

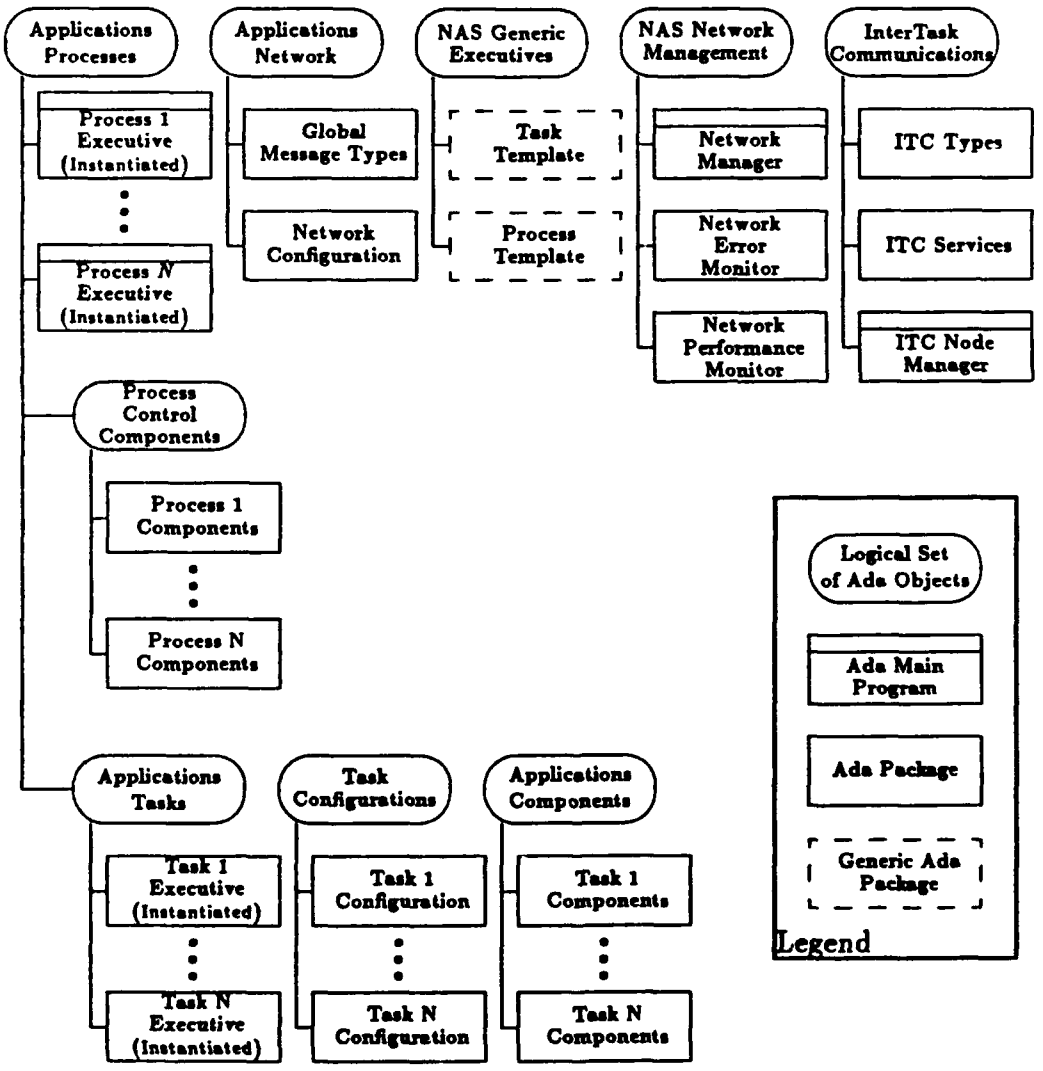


Figure 2: NAS Network Components

Intertask Communication In a C^3 network using ITC and NAS, the network is composed of tasks, processes, and nodes. Tasks are defined as an independent unit of application processing. Sets of tasks are collected into processes, and sets of processes are allocated to nodes. At the application level, the ability to communicate requires only the knowledge of logical task to task interfaces. NAS Executives resolve the logical to physical translations at runtime, while ITC resolves the target task's location and ensures proper message routing.

An Ada application task that uses ITC does so by including a pair of ITC packages in its context clause. These packages import the types and procedures required to use ITC services in an application task. The two critical objects required to enter an ITC network are sockets and circuits. A socket is a virtual I/O PORT that an application task uses to send and receive messages. An application task can communicate by establishing connections with sockets that have been created by other application tasks. The create and connect operations occur dynamically at runtime with no implied sequencing constraints. Since ITC provides buffering and synchronization, application tasks may communicate by executing the ITC procedures called Read, Get_Next_Message and Write without any other processing. ITC copies all written messages into its internally managed storage, determines the physical destination, delivers the messages, and maintains queues for the readers. ITC also has built-in features such as message sequence numbering, cyclic redundancy checks, socket prioritization and other facilities to ensure tailorable, robust operation under normal and abnormal conditions. When an external message comes into the system (e.g., from some external I/O device), its effect ripples through the task network according to the processing logic of the network. ITC implements three levels of task to task communications. A description of each level follows.

ITC Level I Level I contains the uniform user interface, and is self sufficient to service communicating applications tasks which reside in the same process. This level exists in each applications process, and is implemented via Ada tasking with local Ada objects for storage of network state information and message buffering. Applications tasks in a single process which exchange messages and have no external sources or destinations will only activate the first level of ITC. See Figure 3.

ITC Level II Level II is activated (in addition to the first level and invisibly to the caller) to service communicating applications tasks which reside in different processes on the same node. This level exists in each applications process and is implemented through Ada tasking with Operating System Services. Level II provides a means of moving local Ada objects (created at Level I) between processes. The objects are stored in Global memory thus making them available to other processes on the node. Operating System Services are used to provide notification and access to the objects in the Global memory. Receiving processes copy the data from the into local Ada objects, making it available to level I services. ITC provides all of the System Service interfaces so that applications software only need to understand the Level I interface semantics.

ITC Level III Level III is activated (in addition to the first and second levels) to service communicating applications tasks which reside in separate processes on different nodes. This level exists as a separate process on each node in the network. It was implemented through Ada tasking with System Service networking. Level III provides a means of moving Ada objects (stored in global memory by Level II) between nodes in the network. The objects are copied from global memory in one node into global memory in other nodes, making them available to Level II services.

ITC Performance Tuning

An important advantage of providing the uniform ITC interfaces at Level I is that there is a layer of Ada services between the applications programs and the underlying operating system. This layer of software is important to both the flexibility and the reusability of the NAS building blocks. The price one pays for this flexibility however, is an added software overhead of potential significance. In TRW's design of NAS, this overhead has always been a design driver and powerful tuning features were consequently added to ITC to permit the software architect to accommodate performance tradeoffs.

There are basically two dimensions of performance which tend to compete with each other in most real systems: throughput and response time. For NAS networks, throughput refers to the actual message volume which can be sustained across a circuit or set of circuits. Response time corresponds to

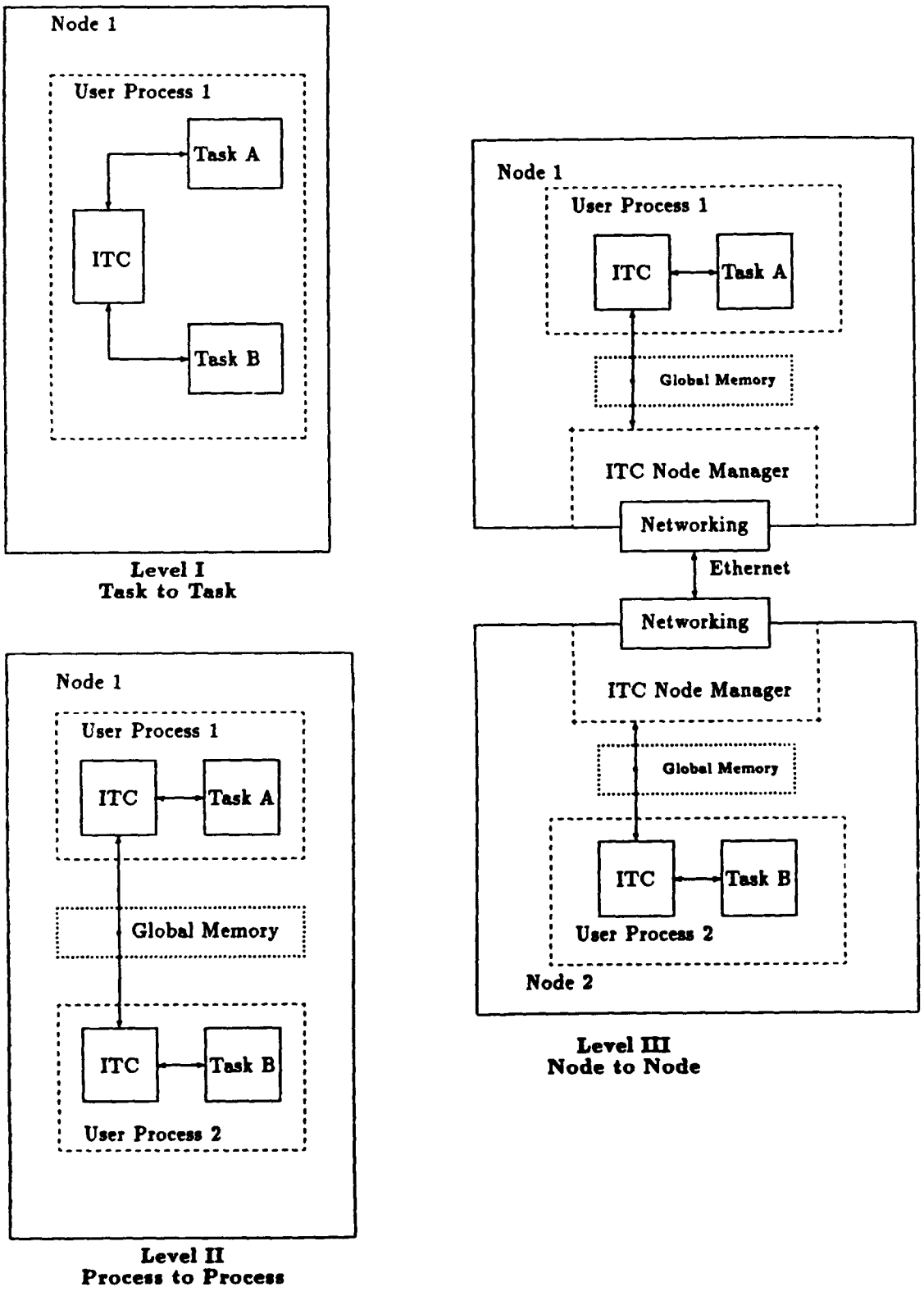


Figure 3: ITC Communication Levels

the timeliness with which NAS can deliver messages. Memory utilization, although a potentially serious concern for some systems (e.g., avionics) was treated throughout the NAS development as a "don't care". This decision stemmed from the NAS' primary target domain being C^3 applications where typical target configurations provide large virtual memory. The critical tradeoffs which confront the software architect in constructing a NAS network design are *partitioning* and *buffering*.

Partitioning corresponds to the definition of which software functions are allocated to which tasks, which tasks are allocated to which processes, and which processes are allocated to which nodes. These decisions result in determining the task-to-task communications that are Level I, Level II and Level III. The higher the level, the longer the response time, and the higher cost per transaction (CPU utilization). The two extreme situations are: 1 process with all communications Level I, or 1 task per process with all communications Level II or Level III. Typical C^3 software architectures will have a mixture of all levels to provide the best ratio of performance, reliability and flexibility.

Buffering corresponds to the definition of the runtime attributes of a given task-to-task communications circuit which optimizes the throughput or response time possible for its message traffic. If a circuit is defined as unbuffered, each message written on that circuit will be delivered as quickly as possible. Architects can also define circuits to be buffered along with a buffer length (i.e., N messages or M Bytes) and a buffer timeout. Messages written to a buffered circuit are maintained internal to NAS until either the buffer becomes full (i.e., N messages or M bytes have been written to the circuit) or the timeout elapses.

Figure 4 illustrates a simple model of ITC internal operations. The reason that buffering is useful is that $t_1 << t_2$ and t_2 is mostly insensitive to message size. Since t_1 and t_2 are both directly related to the CPU utilization (i.e., cost per transaction), the net cost per transaction can be reduced by amortizing the cost of ITC delivery (Level I, II, or III) over multiple ITC Level I accepts. The figure provides a typical relationship between buffer size and cost per transaction. The asymptote corresponds to the limiting cost per transaction imposed by the Level I ITC accept.

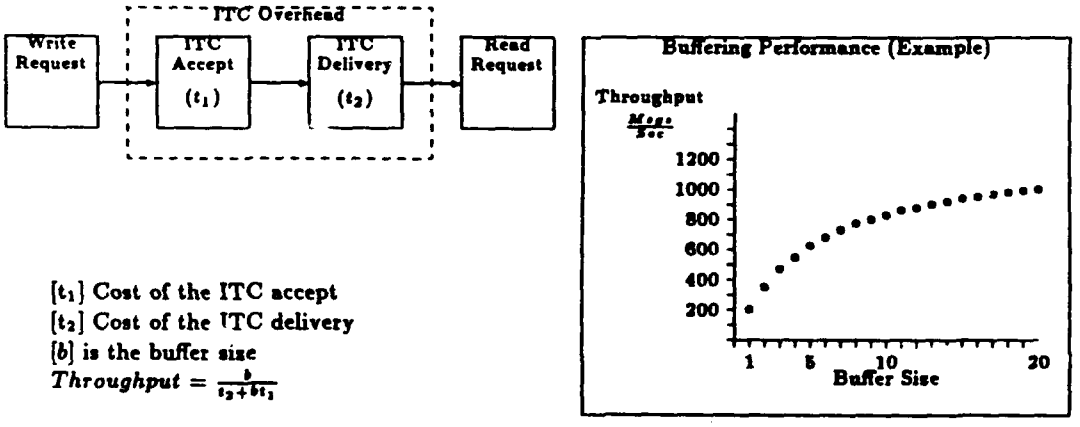


Figure 4: ITC Internal Timing

Experience with ITC tuning shows that frequently the architecture was being tuned to the worst case performance expectations on across the network circuitry. Although this meets the end requirements, we arepotentially degrading the operation at non-peak loads. To accommodate both peak and non-peak conditions, the GAC components were upgraded so that they optionally adapt themselves to current runtime conditions. If a task is busy (i.e., it has queued messages), it will continue to buffer ITC messages as defined for the task circuits thereby optimizing throughput. If a task is not busy (i.e., no queued messages), it will automatically flush its buffered sockets so that no further delay is incurred. These operational procedures dynamically adapt the overhead and the message latency times to the network traffic at hand so that peak and non-peak performance is enhanced.

Figure 5 provides the results of one of our ITC benchmarks to illustrate current performance characteristics. This benchmark shows the peak load which ITC can maintain without queuing within an otherwise unloaded VAX 8800. Since ITC performance is sensitive to multiple other parameters (message size, buffer lengths, buffer timeouts, working set size, no. of nodes in network, no. of destinations per message, etc.), these results are purely illustrative. This benchmark used 500 Byte messages, buffer timeouts of .4 seconds, and buffer lengths of 20.

	ITC Level					
	Unbuffered			Buffered		
	Level I	Level II	Level III	Level I	Level II	Level III
Current $\frac{Msgs}{Sec}$	1600	550	450	6500	4100	3700

Figure 5: Current ITC Performance

Generic Applications Control (GAC) Building Blocks

During the construction of first generation applications with ITC, it became obvious that the standardized task structure and the standardized process structure were candidates for generic implementation. In these applications, very similar components were built for each task and each process. To increase the reliability of the task and process executives, and to enforce standardisation of the executive software structures, generic packages were created for constructing processes and tasks. These generics have numerous parameters for tailoring the executive structure and performance to the application at hand. By using Ada generics to encapsulate the lessons learned in constructing ITC based networks, proven techniques are employed for executive services which are necessary for reliable C^3 networks. These services include: ITC Process Login, ITC Task Login, ITC Socket Creation, ITC Circuit Creation (Socket Connection), Process Executive Initialization, Task Executive Initialization, Task Application Initialization, ITC Input Buffer Access (get next message), Network Error Detection, Network Error Reporting, Network Error Exception Handlers, Process Shutdown and, Task Shutdown.

The top level process and task executives perform all of the processing required to provide a robust operational environment without impacting the design of the application processing packages. For example, the task executive provides status message services to support fault detection and isolation, exception handlers to trap software faults within the application processing, and top level system moding control support such as task initialisation, termination, and restart. Each application task is declared and executed as an Ada task subordinate to a process. The process is referred to as the master task.

An executable process is created by defining an Ada main program, which instantiates and calls the NAS Process Executive. The formal parameters of the generic provide information which identifies the task, processing and data interfaces. The NAS Process Executive provides a standardised framework for process initialisation, communications processing, task control flow and network management interface. Instantiation of the generic NAS Process Executive identifies data and interface information that customises the standardised process framework.

Control of the process is directed by the network manager, which communicates to the process via a control socket. The NAS Process Executive creates and connects this control socket to the network manager (identified at instantiation). It then receives control messages and initiates corresponding task control directives. The Application tasks have both control and application sockets. The task control socket receives the process' control directives and acts accordingly. Application tasks have both control and application interfaces. These are physically known as sockets, which are initialised by the NAS Task Executive at startup. The control interface receives control messages from the process control interface, while application interfaces receive application related communications.

Interactive Network Management (INM) The INM function provides the capability for a Network Management Operator (NMO) to initialise, monitor, reconfigure, and terminate a network of NAS processes from a single master terminal and the capability to monitor a network from multiple slave terminals.

The NMO interface is intended for use by knowledgeable software test or maintenance personnel for the purposes of network construction, performance tuning and detailed operational performance analysis.

The capabilities available to an NMO are:

1. Initialize a predefined (ASCII File) network configuration
2. Monitor errors at a summary network level
3. Review detailed error reports (See Figure 7)
4. Monitor performance at a summary network level
5. Tailor summary level error filtering and performance parameters
6. Perform (ASCII File) predefined network reconfigurations
7. Monitor performance for any node in the network (See Figure 6)
8. Monitor performance for any process in the network
9. Reconfigure the network
10. Reconfigure any node in the network
11. Reconfigure any process in the network

The general purpose user interface designed for the NMO provides capabilities which can be reused for other DEC VT220/VT240/VT340 user interfaces. Figures 6 and 7 provide examples of the NMO user interface format and display content.

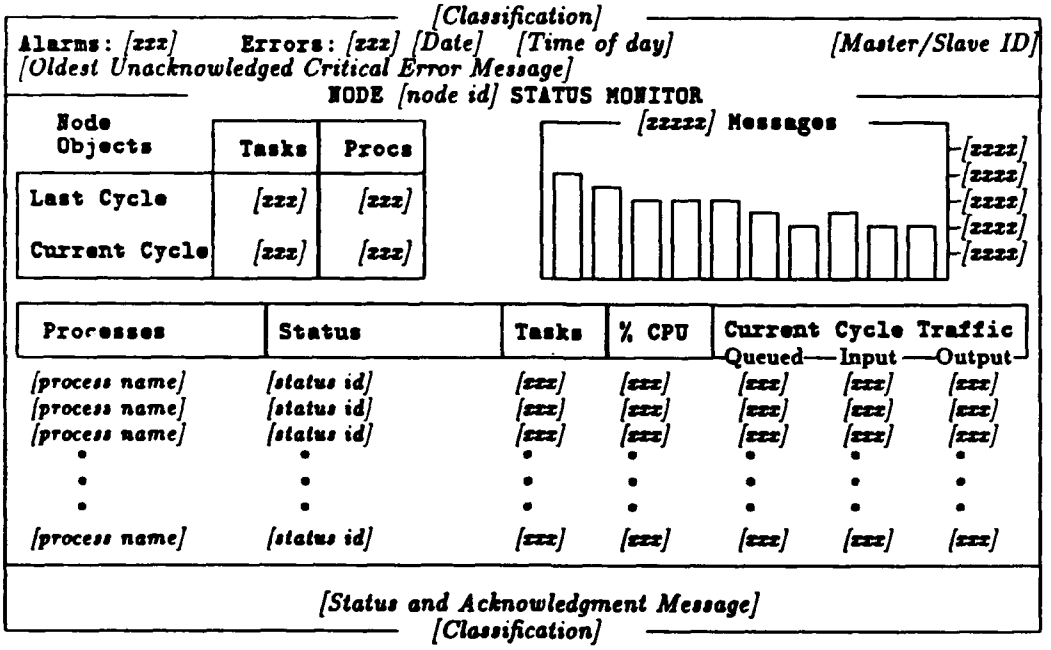


Figure 6: Node Monitoring Display

NAS Error Monitor

All errors in the system are handled through the NAS Error Monitor. The Error Monitor controls the error message flow of the system and provides a flexible, standardized error scheme for all application processes. The Error Monitor can be broken down into three separate components; the Error Code Database, Error Reporter Service, and Error Monitor Task.

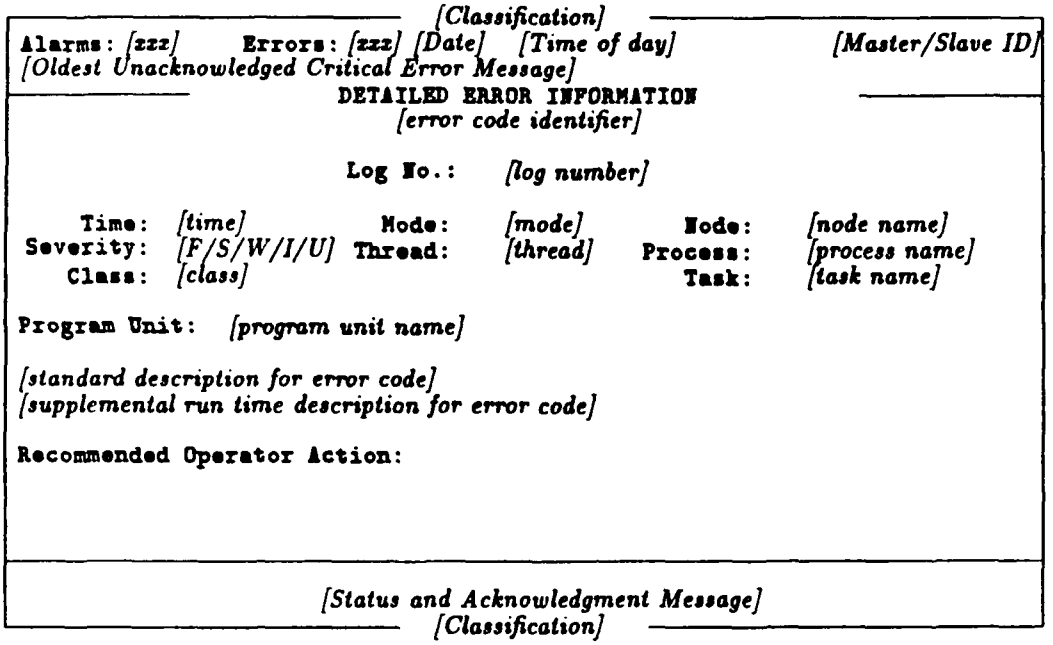


Figure 7: Detailed Error Message Display

All error messages in the system are maintained and controlled in the Error Code Database. This database is stored in an indexed sequential file. Each message can be accessed directly from the database with any of three fields, the Name, Type, or Severity of the error code. By using this database system, errors can be added, deleted, or modified in the system without recompiling code and the direct access of messages allows for quick retrieval of error information. The error code database provides a predefined set of NAS error codes and facilities for adding and maintaining application unique error codes.

The ERROR_REPORTER package is withed by all processes in the system and provides all the necessary interfaces for error logging. All processes create an output socket from the process (done automatically at process startup), using the process name for uniqueness, and connect it to the ERROR_MONITOR_TASK socket. This provides the error monitor circuit used by all tasks in the process to report errors. To report an error, a task will call the ERROR_REPORT procedure and provide an error code ID, program unit name and supplemental runtime information.

In a full up network system, the Error Monitor (ERM) Task will be executed on each of the nodes. Its function is to log all errors that occur on a particular node and to act as a gateway to the Interactive Network Manager. When an error is reported, the ERM Task tries to determine if the error exists and to gather more information on the error such as type, severity, and a description. If no such error exists in the database, an unknown error message (unknown is one enumeration of the error message type) is reported and logged. All unknown errors are subsequently added to the Error Code database. The error is then logged locally by the ERM in the data file ERROR_LOG.DAT and forwarded to the interactive Network Manager.

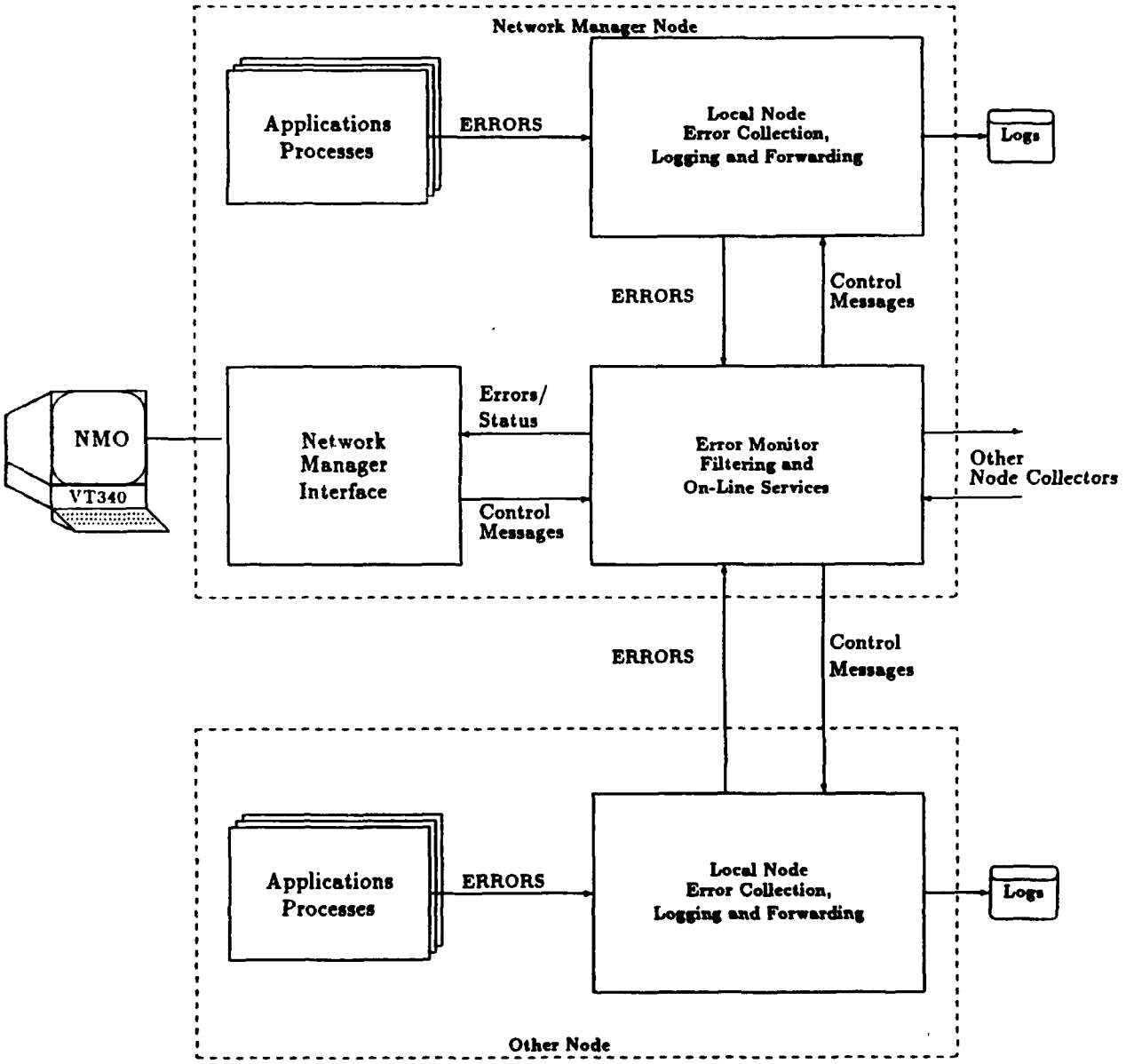


Figure 8: Error Monitor Support Network

SAS Builder Tool

The advantage that NAS provides to a software architect is a consistent set of building blocks for constructing software architecture skeletons with standard interfaces. This uniformity provides increased readability as well as standard interfaces which can be used to develop automated tools. Although other NAS based tools have been built, the most significant is the SAS Builder Tool which automates source code production for the SAS components of the network architecture.

As portrayed in Figure 9, the SAS Builder Tool accepts as input a file definition of the network, performs consistency checks and produces Ada source code which is compliant with TRW standards including headers and comments. On CCPDS-R, this amounts to approximately 30,000 SLOC. This 30,000 SLOC represents generic instantiations and relocateable process and task images which are compiled and linked into approximately 1,000,000 lines of runtime code.

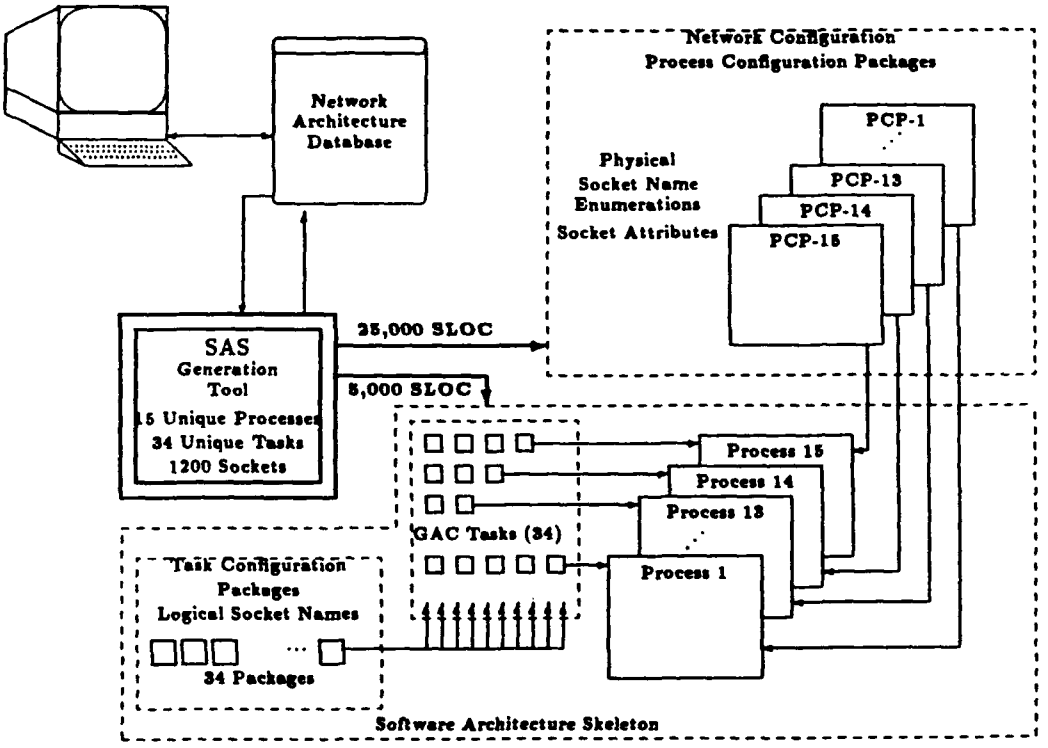


Figure 9: CCPDS-R SAS Builder Tool Production

If the top level architecture decisions are focused into the SAS components, it is imperative to provide flexibility for SAS changes to promote rapid prototyping and demonstration based design. For large networks (CCPDS-R is roughly 10 nodes, 60 Processes, 250 tasks, and 1200 sockets) the corresponding numbers of SAS unknowns and evolving design decisions further emphasizes the need for accurate and rapid SAS changes. The SAS Builder Tool produces Ada source code from a database of SAS definition data: Process names and attributes, Task names and attributes, and Socket names, connections and attributes In constructing first generation NAS networks by hand, it was found that SAS objects were generally straightforward to construct, but that the volume and consistency required resulted in some trivial human introduced errors. These errors did not always become evident until runtime which led to an inefficient network level debugging effort. ITC bookkeeping proved to be simple but voluminous and tedious. For example, 1000 sockets (each with 5-10 attributes) connected in a complex network requires approximately 20,000 source lines of Ada data declarations, many of which must be consistent with each other. For example, "in" sockets should be connected to out sockets and vice versa. This inconsistency would not be caught by a compiler since it is enforced by ITC at runtime. GAC task and process instantiations on the other hand are quite complex. However, each instantiation requires a very cookbook procedure. In early NAS networks, the primary productivity inhibitor was found to

be inconsistencies between the different designers in constructing their GACs and in interfacing to ITC. The solution to this was to isolate all of the GAC instantiations and ITC circuitry definition under the configuration management control of a single person, the chief software architect, and to automate the production of the SAS components.

The operational concept for using the SAS Builder Tool in development and maintenance is dependent on the SAS upgrade required. For example, for small SAS changes such as changing the buffering parameters of a certain socket, the change would be made manually. This type of change is simple, self-contained, and frequent, and only requires that the manual change also be reflected in the configured network database which is input into the tool. Significant SAS changes such as combining tasks, or reallocating tasks into different processes benefit from re-executing the tool and assuring all of the consistency checking therein. In either case, the control and ease of integration afforded by a single source definition of the top-level software interfaces has proven to be extremely productive, especially in the build a little, test a little development approach inherent in the Ada Process Model [Royce 1989].

The SAS Builder Tool currently provides an "application generator" system which is logically a fourth generation language. With suitable upgrades in providing a more user-friendly interface and automatically generated message based design documentation, it could be even more powerful. Currently, hierarchical diagrams are envisioned which provide representations at different levels of SAS detail:

Network Level provides network description along with summary metrics for each node (numbers of processes, tasks, sockets, SLOC, global message types, etc.).

Node Level provides node description along with summary metrics for each process (numbers of tasks, sockets, SLOC, etc.).

Process Level provides process description along with summary metrics for each task (numbers of sockets, SLOC, etc.).

Task Level provides task description along with summary metrics for each socket (attributes, message types processed, etc.).

Circuit Level provides logical task to task thread descriptions through the SAS with circuit attributes for buffering, message volume, response time, etc.

CCPDS-R Experience

NAS is currently a TRW proprietary product. TRW's NAS software was developed at TRW expense (as Independent Research and Development) prior to the award to TRW of the CCPDS-R contract and during a period when such development was not required for the performance of any Government contract or subcontract. The original version of NAS (then called Message Based Design Executive Services) was demonstrated in the CCPDS-R Software Engineering Exercise as part of the Full Scale Development Proposal for CCPDS-R. It was originally developed with the requirement for reusability, a substantial design driver. It has since been enhanced on CCPDS-R with added reliability and performance into a product baseline which is reusable on any homogeneous network of DEC VAX/VMS processors.

Figure 10 provides an Ada COCOMO assessment of the NAS product development and some SLOC counts for the major functions. NAS SLOC assessments are in terms of the Ada SLOC counting rules defined in Ada COCOMO [Boehm/Royce 1988]

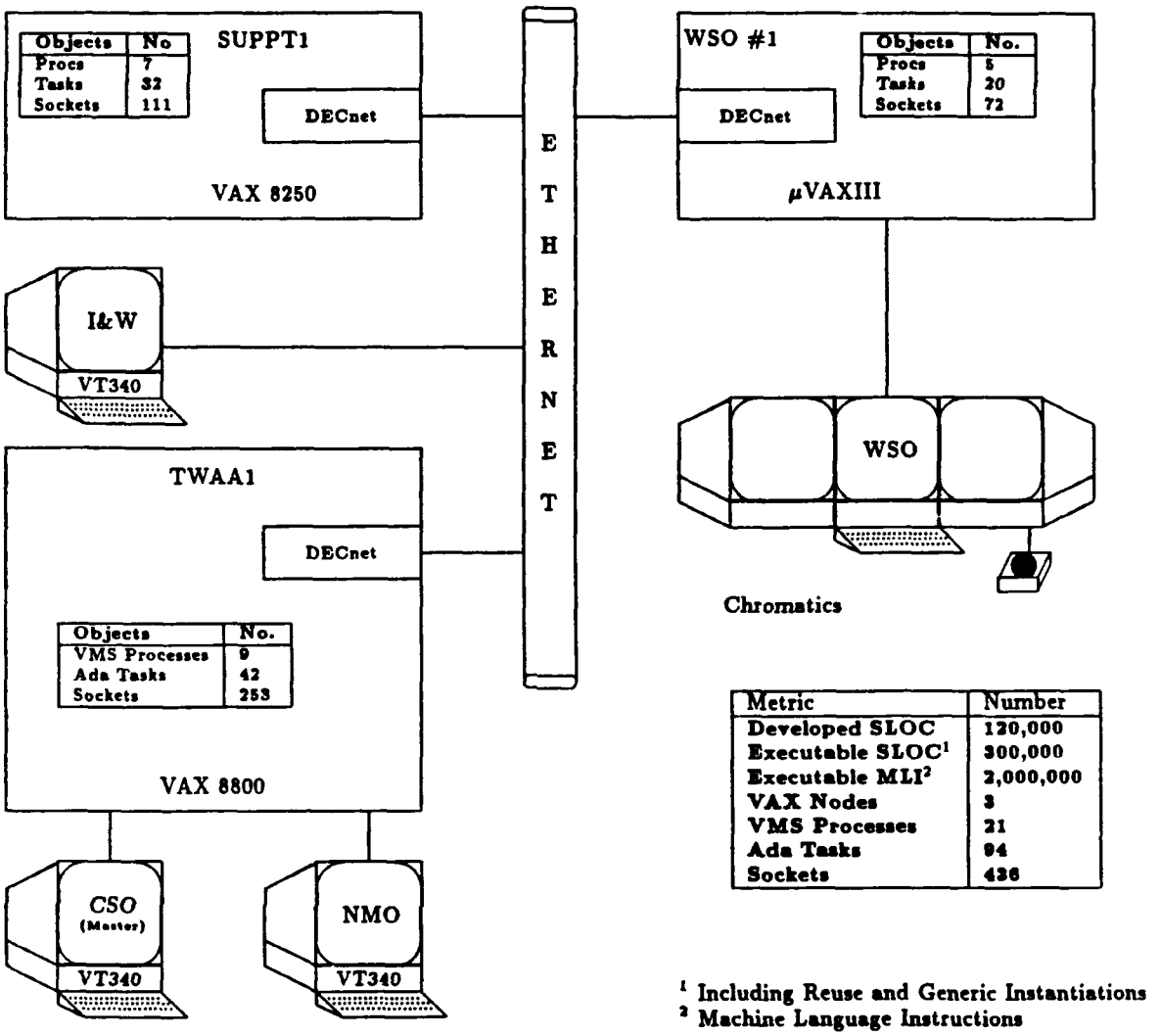
Component Group	SLOC
InterTask Communication	6100
Error Monitoring	2000
Network Performance Monitoring	1100
Generic Applications Control	1000
Interactive Network Management	7200
NAS Utilities	1300
Total NAS	18,700

Parameter	Rating	Effect
RELY	Very High	1.24
DATA	Nominal	1.0
CPLX	High	1.08
RUSE	Very High	1.3
TIME	High	1.11
STOR	Nominal	1.0
VMVH	Nominal	1.0
VMVT	Nominal	1.0
TURN	Very Low	.79
ACAP	Very High	.61
PCAP	Very High	.80
AEXP	Low	1.13
VEXP	Nominal	1.0
LEXP	Nominal	1.04
MODP	Very High	.78
TOOL	Extra High	.73
SCED	High	1.0
SECU	High	1.1
	Total EAF	.55
Exponent	ΣW_i	1.1

Figure 10: NAS Ada COCOMO Description

At the time of this writing CCPDS-R is in month 24 of development. The real proof of NAS' utility is inherent in the large CCPDS-R demonstrations which have been integrated with relative ease. The first CCPDS-R demonstration focused on quantifying the overhead of NAS in the full-up CCPDS-R SAS configuration under a peak message traffic load. This configuration (approximately 75,000 SLOC) took approximately 2 man-months to integrate including several concurrent NAS upgrades which became evident during the process. The second CCPDS-R demonstration (See Figure 11) included the critical components of the CCPDS-R application (message validation, message processing algorithms, real time data distribution, and the associated operator displays) executed under peak load. Figure 11 shows the CPU utilization measured vs our expected (modeled) utilisation. This demonstration also required approximately 2 man-months to integrate including many NAS based application components from many project personnel. The ability to rapidly construct a working system and focus on *real applications interfaces* rather than system software inconsistencies coupled with NAS' extensive support software and instrumentation resulted in an extremely successful effort.

There has been high commitment on the part of the original IR&D team, CCPDS-R project team, the CCPDS-R customer and TRW management to make sure that NAS lessons learned are available in the form of a reusable product.



Metric	Number
Developed SLOC	120,000
Executable SLOC[1]	300,000
Executable MLI[2]	2,000,000
VAX Nodes	3
VMS Processes	21
Ada Tasks	94
Sockets	436

[1] Including Reuse and Generic Instantiations
[2] Machine Language Instructions

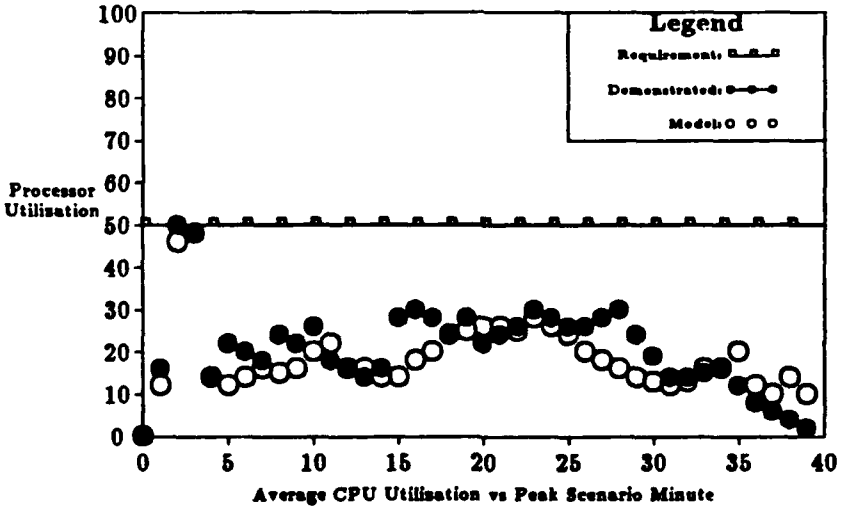


Figure 11: CCPDS-R PDR Demo Results

SUMMARY

NAS has now evolved through 4 generations from a working prototype into a truly reusable, production quality product. Its development has benefited from excellent Ada environments, extremely high quality personnel (with no critical attrition), substantial management commitment, and extensive usage under broad execution environments by many diverse users. The development productivity for the NAS software would be considered extremely low (20,000 SLOC in 300 Man-months) by most standards. The functional sophistication, performance criticality, reliability requirements and design-for-reuse all contribute to making NAS development a very complex effort. Furthermore, given NAS' usage and enhancement over the last two years on CCPDS-R, it is difficult to discriminate between development and maintenance activities. NAS usage on CCPDS-R has substantially increased productivity across the remaining 300,000 SLOC in the Common Subsystem and will provide productivity enhancements of future CCPDS-R Subsystems. This productivity enhancement across the project stems from encapsulating the hard parts of real time systems development into generic building blocks which are easy to use. This permits the large volume of software to be developed to proceed with minimal risk, simple integration, and the necessary development flexibility to react efficiently to lessons learned in an incremental development approach.

Acknowledgments

The success of NAS is a tribute to the entire CCPDS-R team as well as the following key individuals directly involved in its lifecycle development: Cyrus Choy, Russ Cinkle, Chase Dane, Carolyn Dang, Charlie Grauling, Doug Ishigaki, Cort Klein, Jeff Richardson, Ed Rusis, and Ben Willis.

BIOGRAPHY

Walker Royce is the Software Chief Engineer on the CCPDS-R Project. He received his BS in Physics at the University of California, Berkeley in 1977, MS in Computer Information and Control Engineering at the University of Michigan in 1978, and has 3 further years of post-graduate study in Computer Science at UCLA. Mr. Royce has been at TRW for 11 years, dedicating the last five years to advancing Ada technologies in support of CCPDS-R. He served as the Principal Investigator of SEDD's Ada Applicability for C^3 Systems Independent Research and Development Project from 1984-1987. This IR&D project resulted in the foundations for Ada COCOMO, the Ada Process Model and the Network Architecture Services Software, technologies which have since been been transitioned from research into practice on real projects.

REFERENCES

[Royce 1989] Royce, W. E., "TRW's Ada Process Model For Incremental Development of Large Software Systems", *TRI-Ada Proceedings*, Pittsburgh, October 1989.

[Boehm/Royce 1988] Boehm, B. W., Royce, W. E., "TRW IOC Ada COCOMO: Definition and Refinements", *Proceedings of the 4th COCOMO Users Group*, Pittsburgh, November 1988.

[Royce 1989] Royce, W. E., "Development of Reusable Ada Packages Using The VAX 8600 and the Rational R1000 Ada Environments", *Proceedings of Methodologies and Tools for Real-Time Systems Conference*, National Institute for Software Quality and Productivity", September 8, 1986.

[Booch 1986] Booch, G., "Software Engineering With Ada", Benjamin/Cummings.

[Grauling 1989] Grauling, C. G., "Requirements Analysis For Large Ada Programs: Lessons Learned on CCPDS-R", *TRI-Ada Proceedings*, Pittsburgh, October 1989.

[Springman 1989] Springman, M. C., "Incremental Software Test Methodology For A Major Government Ada Project ", *TRI-Ada Proceedings*, Pittsburgh, October 1989.